T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Verification Environment for I/O Automata Based on Formalized Meta-Theory

Olaf Müller

TUM-I9822
September 98

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-09-I9822-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

A Verification Environment for I/O Automata Based on Formalized Meta-Theory

Olaf Müller

Fakultät für Informatik
der Technischen Universität München

A Verification Environment for I/O Automata Based on Formalized Meta-Theory

Olaf Müller

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. Javier Esparza
Prüfer der Dissertation:		
	1.	Prof. Tobias Nipkow, Ph. D.
	2.	Prof. Dr. Manfred Broy

Die Dissertation wurde am 24. Juni 1998 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18. August 1998 angenommen.

Abstract

This thesis deals with the computer-assisted verification of embedded systems described as Input/Output automata. We achieve contributions in two fields: the theory of untimed I/O automata and its tool support. For the latter a combination of the theorem prover Isabelle with model checking is used.

Concerning the theory of I/O automata, we present a new temporal logic which considerably facilitates the usual implementation proofs between live I/O automata and serves as a property specification language. Furthermore, a new theory of abstraction is developed which allows us to combine theorem proving and model checking. Theorem proving is used to justify the reduction of the original system to a smaller model which is then analyzed automatically by model checking. The theorem prover reasons about first-order proof obligations only. For the remaining obligations translations to the model checkers μcke and STeP are given. In contrast to existing abstraction theories, the proof of both temporal properties and implementation relations can be abstracted, a methodology for treating liveness abstractions is proposed, and a formal relation to the dual concept of refinement is established.

Concerning tool support, we provide a verification environment that covers both standard I/O automata theory and the aforementioned extensions. The overall guideline for its construction is reliability. Therefore, meta-theoretic notions of I/O automata such as compositionality and refinement are mechanically verified in Isabelle. Nevertheless, a practicable environment for system verification is obtained. This is due to a new methodology for Isabelle's logics HOL and HOLCF, which allows the user of the framework to employ the simpler logic HOL, whereas meta-theoretic investigations gain from the more expressive HOLCF. Possibly infinite communication histories of I/O automata are formalized as lazy lists based on Scott's domain theory. This results in a generally applicable sequence model which turns out to be favorable in an elaborate comparison with alternative formalizations. In addition to the usual inductive proof principles for lazy lists we provide an infrastructure for coinduction.

The practical relevance of the resulting verification environment is proven by several case studies, in particular by an industrial helicopter alarm system.

Acknowledgments

First of all, I would like to thank Tobias Nipkow for his continuous support. His supervision has been a perfect mixture of advice, inspiration, and encouragement.

Furthermore, I am grateful to many friends and colleagues for valuable discussions and helpful comments on draft versions of the thesis. They include Frits Vaandrager, Stephan Merz, Markus Wenzel, Larry Paulson, David Griffioen, Marco Devillers, Ingolf Krüger, David von Oheimb, Jan Philipps, and Oscar Slotosch. Moreover, I wish to thank Tobias Hamberger for his efforts on carrying out a case study in order to demonstrate the developed tool set.

I am indebted to Manfred Broy and Tobias Nipkow for providing the stimulating environment that made this thesis possible. This includes many other people of the Munich research group as well, especially those who were involved with Isabelle or participated in the KorSys project. I merely mention Peter, Stephan, Wolfgang, David, Jan, Oscar, Max, Christian, Franz, and Conny. Special thanks are due to my roommates Konrad Slind and Markus Wenzel for the excellent atmosphere in our office. I really enjoyed the inspiring discussions at tea time.

I am glad that this thesis is completed, because during its writing I had less time to spend with my daughter Susanna than I would have liked to have. And I would like to thank my wife Christiane for her love and support and for reminding me of the more important things in life. Finally, there are two people who laid the basis for my current life and work with regard to many aspects: Pa & ma: thank you.

To Christiane and Susanna

Contents

Chapter 1

Introduction

This chapter provides an overview of the thesis. We start with the motivation of our work and proceed with presenting the main goals and results. Finally, we give an outline of the subsequent chapters.

1.1 Motivation

In technical applications of computer science *embedded systems* play an increasingly important rôle. Such systems are said to be embedded as they form an integral part of the physical environment they operate in. Application domains range from avionics and automotive electronics to telecommunication and process control. Characteristic of embedded systems is their *reactivity*: they continuously accept stimuli from the environment via sensors and react to them via actuators. Hence, they appear to be more complex than the usual *transformational systems* which process a given input, produce an output, and terminate. In addition, embedded systems are typically *distributed*, meaning that they consist of many separate processes that communicate with each other by message passing or shared memory. This makes the construction of embedded applications one of the most challenging tasks facing the computer science community, if not the engineering community as a whole.

This task has to deal with the entire development process, but often the *verification* phase is of particular interest, as reactive, distributed systems occur frequently in *safety-critical* areas. In such areas, errors in the control code can have disastrous consequences, including the endangering of human life. Take as an example the flight failure of the Ariane 5 launcher in 1996, which was caused by a software error that has not been detected in spite of intensive testing and other techniques of quality management [ESA96].

Here, the scientific community has proposed the use of *formal methods.* This term covers all approaches to specification and verification based on mathematical formalisms. The aim is to establish system correctness with mathematical rigour. In contrast to pure testing, this methodology increases considerably the confidence into the behaviour of the produced software. Moreover, it has the important benefit that it directly supports early error detection by revealing logical errors already in the design phase. This offers the possibility to reduce the immense costs usually spent for the testing phase.

Most approaches to formal verification can be classified according to the following two categories.

A-Priori Verification. If a system is designed hand-in-hand with the proof of its correctness we refer to this approach as *a-priori verification*. Such approaches usually support the development process in hierarchies of implementation layers, where the refinement between successive layers is shown by the use of *implementation relations*.

A-Posteriori Verification. Alternatively, a complete system may be verified *a-posteriori*. In this case, a *property specification* language is provided, which allows to express properties of particular interest, for example safety-critical requirements. Verification establishes that the system behaves properly with respect to a given property.

In practice, the two approaches may be used in combination, for example by building a system hierarchically while the correctness of the building blocks is established before combining them. Hence, our aim will be to support both approaches. In any case, formal verification is time-consuming and error prone, if done manually. Therefore, there is a demand for effective computer-assistance which facilitates and (at least partially) automates the tedious proofs on paper. Furthermore, computer-assisted proofs are in general more reliable than manual proofs.

Consequently, the overall goal of this thesis can be summarized as follows:

Tool supported verification of reactive, distributed systems

In the following we motivate the technology we will use and improve in order to achieve this goal. This comprises the system development method of *I/O automata* and the two major paradigms for tool supported verification, *theorem proving* and *model checking*.

We will see that models for reactive systems as well as the verification technology has made substantial progress in recent years. The challenge is to combine these techniques to a tool-supported methodology for embedded systems. This will turn out to be not merely an engineering problem, but a source for a number of foundational research questions.

I/O Automata. Input/Output automata are a semantic model for reactive, distributed systems together with a tailored refinement concept. The model has been originally proposed by Lynch and Tuttle [LT87], subsequent developments are mainly due to Lynch and Vaandrager [LV95, LV96, Lyn96, GSSL93, RV96, LSVW96]. The method has already been successfully applied to the verification of several non-trivial case studies, ranging from communication protocols [SLL93] and automated transit systems [DL97] to database applications [LMWF94].

Apart from I/O automata several further automaton models have been proposed [Har87, HN96, Kur94, Bro97, Kle98, Rum97]. I/O automata distinguish themselves from them by offering a compositional, but nevertheless remarkably simple semantics.

Most closely related to I/O automata are probably the automaton models developed in [Bro97, Rum97], which are specifically tailored to support the design phase of the development method Focus [Bro93b, BDD+93]. The main difference is that Foucs features a *clock synchronous* instead of *interleaving* semantics, meaning that it permits several input and output steps in one computation step, whereas I/O automata arrange all conceptually simultaneous actions in an arbitrary order. The interleaving semantics of I/O automata is less complicated because it does not need a fixpoint theory in order to deal with the feedback of simultaneous actions. Furthermore, it permits a more general form of nondeterminism, namely angelic nondeterminism [Bro85, Ded92]. Focus does allow only for erratic nondeterminism, which makes it necessary to introduce a global clock in order to be able to express notions like the fair merge function. The price for the simplicity of interleaving is a single drawback: as inputs are always enabled, outputs cannot be forced to be produced within a given time bound. The only way to ensure the occurrence of output actions is to pose fairness conditions on them. A consequence is that sometimes inputs have to be stored in unbounded buffers before the corresponding output can be produced. This contradicts verification methods that require finite-state descriptions. Later on, we will propose a methodology that solves this problem in a convenient way.

To conclude, I/O automata appear to be the appropriate choice, as they feature a concise semantics that is less complicated to be mechanically verified in a proof tool than possible alternatives.

Theorem Proving. Theorem provers construct proofs in a logical deduction system, either completely automated or under interactive guidance of the user. In recent years, especially interactive theorem provers based on higher-order logic have matured in such a way, that real-world applications have come into their reach. The most widely used verification systems of this type are probably HOL [GM93], Isabelle [Pau94], and PVS [ORR+96]. Examples for real-world applications include verifications dealing with the AAMP5 microprocessor [MS95], the type system of a subset of Java [NvO98], and various security protocols [BP98]. The reason of the success can mainly be found in the expressiveness of higher-order logic, which allows us to formulate even complex application domains in an adequate and natural way.

For our verification efforts we will use Isabelle in favour of HOL or PVS, because of the following reasons. In comparison to HOL, Isabelle is more generic, comes with more built-in theorem proving power, and appears to be more user-friendly, as it provides powerful syntactic facilities. In contrast to PVS, Isabelle is more flexible, offers several logics, and is built according to the "LCF system approach" [Pau87]. The latter means that every proof is internally broken down to a small and clear set of primitive inferences. This considerably increases the confidence in the soundness of the proof tool itself and thus in its machine-checked proofs.

A further advantage of the expressiveness of higher-order logic is that it allows us to deal with *meta-theory*, i.e. it is possible to verify not only concrete systems, but the underlying semantic model as well. In theorem provers based on first-order logic meta-theory is not directly expressible. Instead, results on paper are used to extract a set of proof obligations from the formal description of the system in question. A formal treatment of the underlying meta-theory, however, is of advantage as it leads to more reliability. This is especially true for semantic models which have not yet been verified mechanically, as it is the case for I/O automata. Apart from greater reliability, the benefits are also greater maintainability because the theorem prover keeps track of the impact the changes have on already established properties. Furthermore, a formal meta-theory reveals implicit assumptions and helps to organize them in a convenient way. Finally, it offers a greater degree of flexibility because one does not need to hardwire certain proof methods but can derive new ones at any point. Because of these considerations we will rely on verified meta-theory only, which implies that all theory extensions will be *definitional*, i.e. no externally justified axioms will be introduced.

Model Checking. In contrast to the *proof-based* theorem proving paradigm, model checking is a *model-based* verification method. This means that the system, described in a mathematical structure like a transition system, is regarded as a possible model of the property to be verified. Consequently, the property is specified as a formula in a logic that is interpreted over such structures (e.g. temporal logic). Proving correctness then boils down to checking if the formula is satisfied by the model. For finite-state systems this can in principle be done automatically by an exhaustive traversal through the reachable state space. In practice, however, there are limits in terms of the system size which can be handled. Unfortunately, systems made up of several components can easily reach these limits as the size of their global state space is, usually, exponential in the number of components.

In the last decade, several approaches have been proposed that attack this *state explosion* problem [CES86, BCM$^+$92, GW94, HP94]. The ideas are either to reduce the number of actually needed states or to work on a more efficient state representation. This resulted in a number of efficient model checking tools [CGL94, Hol91, BBC$^+$96, HR87, Bie97a], which in the meanwhile raised industrial interest as well.

The success of the approach is mainly due to the high degree of automation. However, the tools are still restricted to finite-state systems with relatively small state space. Thus, model checking is in some sense complementary to theorem proving, which can deal with arbitrary systems, but requires user interaction. This duality is the motivation for a further goal of this thesis: we will combine theorem proving with model checking in such a way that the strengths of both approaches can be exploited.

1.2 Main Goals

The discussion of the previous section allows us now to put the overall goal of this thesis in more concrete terms. We aim at building a verification environment for I/O automata using a combination of Isabelle and model checking. The overall guideline should be reliability, which in particular implies an emphasis on verified meta-theory and logical foundations. To achieve this challenging task, contributions have to be made to the following fields, which correspond to the four main parts of this thesis:

Part I: Extensions to the theory of I/O automata

Part II: Logical foundations and theorem proving methodology

Part III: I/O automata in Isabelle

Part IV: Evaluation of the approach by case studies

In the sequel we give a more detailed overview of the goals for each of the above fields.

1.2.1 Part I: Extensions to the Theory of I/O Automata

The following three enhancements to the theory of untimed I/O automata are essential in order to make formal verification more general, more convenient, and amenable to effective tool support, respectively.

A-Posteriori Verification. Up to now, I/O automata do not support a-posteriori verification. The proof techniques known so far deal only with the verification of implementation relations. Therefore, there is a need for a property specification language. We will close this gap by defining a temporal logic over executions of I/O automata. Existing temporal logics for program verification, like TLA [Lam94] or the logic by Manna/Pnueli [MP95], cannot be applied directly in this setting, as executions of I/O automata contain *explicit* actions[1]. Even in TLA, however, actions are merely *state* changes. Furthermore, existing temporal logics do not consider finite computations.

A-Priori Verification. Whereas safe implementation relations have been studied very well for I/O automata [LV95], there is no tailored proof infrastructure for live implementation relations. The standard game-theoretic treatment of liveness [GSSL93] deals only with general requirements on the automaton model, but does not provide a proof infrastructure for liveness proofs. Therefore, actual refinement proofs involving fairness are usually very low-level or rather ad-hoc [Lyn96, Rom96]. Our aim is to employ temporal logic to support

[1] During the development of our temporal logic we realized that a similar logic has been proposed in [SLL93]. However, it is closer to Manna/Pnueli's logic, whereas we are inspired by TLA.

the proof of live implementation relations. This aim is motivated by the general insight of the scientific community [GPSS80, Lam83], that for safety proofs automata appear to be most appropriate, whereas for liveness proofs temporal logic is most adequate.

Combination of Theorem Proving and Model Checking. In recent years, *abstraction techniques* have been proposed in order to integrate theorem proving and model checking in such a way, that more than just the sum of the parts is obtained [CGL92, LGS$^+$95, Kur87, DGG97, SLW95, Wol86]. The idea is to reduce the original system to a smaller model via interactive proof techniques. In a second step, the smaller system is analyzed using automatic tools. Usually, the smaller system is obtained by partitioning the original state space via a structure-preserving function between the two state spaces. The theorem prover is employed in order to guarantee the abstraction to be sound, i.e. if the abstract system satisfies a property, so does the original system.

So far, an abstraction theory for I/O automata does not exist. In addition, existing theories are tailored for either a-priori or a-posteriori verification. Our aim, however, is to provide an abstraction theory that allows us to reduce the proof of both temporal properties and implementation relations. Therefore, we have to unify the ideas of several approaches and rephrase them in the framework of I/O automata. Furthermore, existing work does not explicitly deal with liveness properties, which are known to be harder to prove under abstraction than safety properties. To our knowledge, merely the recent paper by Merz [Mer97] proposes a tailored proof rule for this case, developed in the framework of TLA. In this respect, we will adapt some of his ideas, and build on top of that a proof infrastructure and methodology that handles liveness appropriately. Our overall maxim will be to reduce the proof amount for the interactive part as much as possible.

1.2.2 Part II: Logical Foundations and Methodology

In order to verify the meta-theory of I/O automata by theorem proving, we have to provide some fundamental theories in Isabelle first. In particular, we need a theory of possibly infinite sequences, which can be used to model the communication histories of I/O automata. Furthermore, we will see that a methodology is required in order to combine two of Isabelle's object logics in a fruitful manner, namely Isabelle's version of higher-order logic (HOL [Pau94]) and its extension to Scott's domain theory (HOLCF [MNOS98, Reg95]).

Sequences. The formalization of possibly infinite sequences in theorem provers based on higher-order logic is an active research area [MN97, DG97, DGM97, CP96, HJ97a, Pau97]. In most cases, the motivation is to provide a formal foundation for particular semantic models of reactive systems. The starting point of our work is a very basic embedding of I/O automata in Isabelle by Nipkow and Slind [NS95], which models system runs by partial functions on natural numbers. During our attempts to extend this model we will, however, encounter a significant difficulty with this sequence model: it does not permit to capture the usual implementation relation between I/O automata, but merely a very restricted version of it. In addition, we will show that it is not feasible to extend the sequence model appropriately. Thus, our aim is to develop a new sequence theory which does not only solve this problem, but is powerful enough to deal easily with any application of possibly infinite sequences. Therefore, we will in addition provide an extensive comparison of our model with the aforementioned alternative solutions proposed in the literature.

We will model sequences as lazy lists using Scott's domain theory in HOLCF, which allows us to deal with lazy datatypes very easily. Thus, sequences are represented as recursive domains and functions on them as recursive functions via fixpoint constructions. The usual proof principle in this setting is structural induction. We will complement this reasoning style with the dual coinductive proof principles and give advice how to choose between the two proof alternatives.

HOL/LCF Methodology. When modeling sequences in HOLCF, it will turn out that sequence elements are easier dealt with in pure HOL. Therefore, it should be possible to mix terms of HOL and its sublogic HOLCF. So far, however, both logics were thought of as two separate worlds [Reg94]. From a methodological point of view it is, however, not at all convenient to stay always either in HOL or in its LCF extension. The reason is that HOLCF is more expressive and powerful, but reasoning in it is more complicated as well. Therefore, we will develop a methodology that allows us to choose always the logic which is the most appropriate one. A technical prerequisite for this methodology is the extension of the automatic continuity check in such a way that is covers mixed HOL and LCF terms as well.

1.2.3 Part III: I/O Automata in Isabelle

Most of the verification efforts dealing with I/O automata have been performed on paper, without computer-assistance. Nevertheless, there are several theorem-proving approaches that support the verification of I/O automata [PPG+96, AH97a, GD98]. However, they are either based on first-order logic or rely much more on unformalized meta-theory than we intend to do. Our aim is to adapt the basic model by Nipkow and Slind [NS95] to our new theory of sequences. Then we will set out to build an extensive framework for the verification of I/O automata, including the standard theory as well as the above mentioned extensions.

Meta-Theory and System Verification. The overall aim of our I/O automaton embedding is twofold. First, we want to build a tool environment for system verification that is efficient and easy to use in practice. Second, we want to verify the theory of I/O automata itself, which requires powerful and sophisticated meta-theoretic constructions. We will achieve these rather complementary goals by exploiting our HOL/LCF methodology in such a way that the user employs merely the simpler logic HOL, whereas meta-theoretic proofs may profit of the more expressive HOLCF.

Standard I/O Automata Theory. Our aim is to verify all meta-theoretic notions the user has to rely on when performing system verification. This includes in particular the correctness of refinement notions and of compositionality. Whereas existing work [AH97a, GD98, NS95] deals only with invariants or (weak) refinement mappings, we want to consider forward simulations as well, which cover the expressiveness of history variables. As far as we know, existing work does not deal with compositionality. Besides meta-theory, we aim at building a proof infrastructure for the user by providing specialized tactics.

Theory Extensions. All extensions to the standard theory of I/O automata described in §1.2.1 should be embedded and verified in Isabelle as well. This includes the temporal logic, proof support for live I/O automata, and the abstraction theory.

To conclude, we want to cover a-priori and a-posteriori verification. Both verification styles should be handable either completely in Isabelle or by a combination with model checking via abstraction. All concepts should be definitionally developed within Isabelle.

1.2.4 Part IV: Applications and Evaluation

In order to evaluate the practicability of our I/O automata framework, we will perform several case studies. Each case study will cover different aspects of the tool set. The combination of Isabelle with model checking via abstractions should be evaluated as well as the purely interactive proof of implementation relations via forward simulations. There are several assessment criteria which appear to be important. First, scalability is mandatory for any industrial-strength formal method. Therefore, we will perform a case study of a helicopter alarm system with industrial background that is no longer manageable for model checking. Second, the usability for design engineers not familiar with verification is of importance. Thus, one case study will be performed as a student project. Finally, it is interesting whether abstraction indeed allows for a considerable reduction of the required proof effort. To anticipate the result already, abstraction will turn out to be a surprisingly powerful tool in practice.

1.3 Main Results

In this section we briefly summarize the main results of this thesis.

Part I: Concerning the theory of I/O automata, the following enhancements have been achieved:

- We define a linear-time temporal logic (called Temporal Logic of Steps — TLS) as property specification language for I/O automata. In contrast to existing temporal logics, formulas are evaluated over sequences of alternating states and actions which may be finite.

- We investigate the formal relation to existing temporal logics, namely to TLA [Lam94] and the logics by Manna/Pnueli [MP95] and Kröger [Krö87].
- We show how TLS can be used to support the proof of live implementation relations.
- We develop an abstraction theory which allows us to reduce the verification of both temporal properties and implementation relations to finite-state model checking. Even for liveness proofs the theorem prover has to deal with simple first-order proof obligations only.
- For the remaining proof obligations we give translations to the model checkers μcke [Bie97a] and STeP [MBB$^+$97]. The translations lay particular emphasis on structure preservation in order to increase efficiency.
- Concerning liveness properties, we develop a method which permits to reuse the abstraction concepts of the safety part and therefore reduces the amount of intuition required.
- For safe abstractions, we show the dual character of abstraction and refinement by proving a completeness result w.r.t. a tailored notion of implementation.
- We propose a general solution to the mismatch of model checking and unbounded buffers that are due to the interleaving semantics of I/O automata (in §12).

Consequently, we yield a theory that covers a-priori and a-posteriori verification, treats safety and liveness adequately, and permits an effective combination of theorem proving and model checking.

Part II: The following contributions concern the logical foundations in Isabelle.

- We develop an interface between HOL and its LCF extension, which allows us to combine pure HOL and LCF terms without the drawback of manual continuity proofs.
- Based on this interface we propose a methodology for the combination of HOL and HOLCF that allows us to choose the adequate logic for each part of a given problem. In addition, it enables the reuse of well-established HOL theories.
- We provide a theory of possibly infinite sequences modeled as lazy lists in HOLCF. Using the HOL/LCF methodology we considerably increase the degree of automation and eliminate reasoning about nasty cases dealing with undefinedness.
- The sequence package offers both inductive and coinductive proof principles. For the latter we provide a tailored proof infrastructure and methodological guidance.
- We derive almost 200 useful theorems for this theory of sequences.

- We compare our theory with four alternative formalizations of possibly infinite sequences in higher-order logics. Our approach turns out to be the most appropriate one because of its general recursion and the diversity of its proof principles.

Consequently, we believe that our HOL/LCF methodology is of vital importance for any serious use of HOLCF. Furthermore, we are convinced that our theory of sequences will be of great value for any application involving possibly infinite sequences.

Part III: The following aspects characterize our verification environment for live I/O automata.

- Exploiting our HOL/LCF methodology we achieve two complementary goals: expressiveness for proving meta-theory (in HOLCF) and simplicity and efficiency for system verification (in HOL).
- We prove the soundness of safe and live forward simulations. The proof is very concise because of the appropriateness of the underlying sequence model.
- We prove compositionality and non-interference for safe I/O automata. The proof is significantly harder than its informal counterpart and demonstrates the insufficiencies of semi-formal paper proofs.
- We formalize the temporal logic TLS as an instantiation of a generic temporal logic on top of our theory of possibly infinite sequences. In contrast to existing embeddings of temporal logics, we incorporate finite sequences and are able to deal with stuttering.
- We embed our abstraction theory by deriving appropriate abstraction rules.
- We provide a tailored proof infrastructure for the user of the verification environment. The entire I/O automaton framework consists of more than 300 theorems.

Consequently, we think that the formalization of meta-theory is worth-while, as it offers a flexible platform, increases reliability, and provides deeper insights into the logical foundations. Furthermore, we have shown that verified meta-theory and an efficient framework for system verification need not be contradictory.

Part IV: Finally, we got the following insights due to the evaluation of case studies.

- Our verification environment scales up to case studies of industrial size.
- Abstraction techniques allow for significantly better verification results than those obtainable by pure model checking or pure theorem proving approaches.
- The framework is easy to use, even for design engineers not familiar with the verification technology.

Consequently, we have every reason to believe that machine-checking of distributed, reactive systems has become a reality, and is more than just feasible.

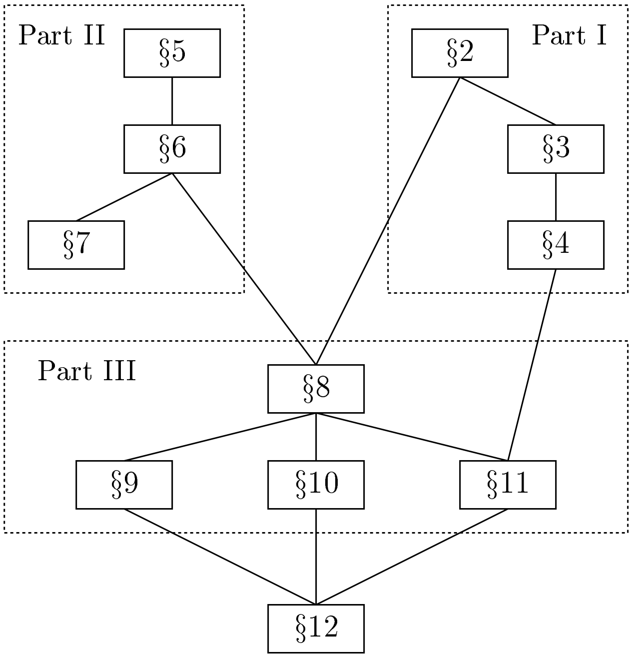

Figure 1.1: Dependency Graph of the Chapters

1.4 Outline of the Thesis

In this section we briefly sketch the structure of the thesis. The dependency graph of the chapters is depicted in Fig. 1.1. Note that related work is dealt with in detail at the end of each chapter.

Part I: In §2 we briefly survey the theory of untimed I/O automata according to Lynch and Vaandrager. This theory is extended by a temporal logic in §3, and by an abstraction theory together with model checking translations in §4.

Part II: In §5 we introduce Isabelle and propose a methodology for the combined use of its object logics HOL and HOLCF. Based on this methodology, a theory of sequences is developed in §6, which is compared with alternative formalizations in §7.

Part III: In §8 we describe the basics of our verification environment for I/O automata in Isabelle and discuss some general aspects of it. Then, we present the main meta-theoretical proofs, dealing with the correctness of refinement in §9 and compositionality and non-interference in §10. In §11 the extensions of §3 and §4 are embedded in Isabelle.

Part IV: In §12 three cases studies are described, together with the lessons learned. Finally, we conclude in §13.

Publications: The thesis contains parts of the following previously published material: [MN95, MN97, DGM97, MS97, Mül98, MNOS98].

Part I: Extensions to the Theory of I/O Automata

Chapter 2

Basic Theory of I/O Automata

In this chapter we give a concise survey of the theory of I/O automata according to Lynch and Vaandrager. This lays the foundation for the theory extensions we will present in the chapters 3 and 4. Furthermore, it roughly outlines the parts of the standard I/O automata theory we will formalize within Isabelle in the chapters 8–10.

2.1 Introduction

In the sequel a concise introduction to the theory of Input/Output (I/O) automata is given. The model has been originally developed by Lynch and Tuttle [LT87, LT89], subsequent extensions profited mainly from contributions by Lynch and Vaandrager [LV95, LV96, Lyn96, GSSL93, RV96]. The presentation in this chapter follows [Lyn96] rather closely. In addition, strong fairness is considered as in [RV96], and backward simulations and the completeness result for simulations are taken from [LV95]. Furthermore, general liveness is considered, which has been taken from [GSSL93]. Proofs of basic theorems are mostly taken from [LT87], as they are the most elaborate. Extensions of the standard theory, as for example timed or hybrid I/O automata [LSVW96, LV96] or probabilistic additions [SL95], are not taken into account.

First, we mention some basic mathematical preliminaries. A *relation* over sets X and Y is defined to be any subset of $X \times Y$. Notions like *domain*, *range*, the *composition* operator $\circ$, and the *inverse* R^{-1} are defined as usual. A relation is said to be *total*, if $domain(R) = X$. For every $x \in X$ we define $R[x] = \{y \in Y \mid (x, y) \in R\}$. Polymorphic lists, denoted by

(α)*list*, are built by the constructors : and the empty list $[\,]$. Instead of $a_1 : \ldots : a_n : [\,]$ we write $[a_1, \ldots, a_n]$ as usual. Operators on lists include *hd*, *tl*, *length*, *filter*, and $\in$. For *filter* $(\lambda x.\ P(x))\ xs$ we write $[x \in xs.\ P(x)]$. The set of booleans is denoted by *bool*.

2.2 Fair I/O Automata

In this section we introduce the model of *fair I/O automata*, which includes the definition of basic I/O automata, composition operators, and the notion of behaviors.

Definition 2.2.1 (Action Signature)
An *action signature* S is a triple of three disjoint sets: the *input* actions, $in(S)$, the *output* actions, $out(S)$, and the *internal* actions, $int(S)$. The *external* actions are defined by $ext(S) \equiv in(S) \cup out(S)$, *locally controlled* actions by $local(S) \equiv out(S) \cup int(S)$, and all actions are denoted by $acts(S) \equiv in(S) \cup out(S) \cup int(S)$. □

Definition 2.2.2 (Safe and Fair I/O Automata)
A *fair I/O automaton* A consists of the following components:

- an action signature $sig(A)$,
- a set $states(A)$ of states,
- a nonempty set $start(A) \subseteq states(A)$ of start states,
- a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$, such that for every state s and action $a \in in(A)$ there is a transition $(s, a, t) \in steps(A)$,
- sets $wfair(A)$ and $sfair(A)$ of subsets of $local(A)$, called the weak fairness sets and strong fairness sets, respectively.

Fair I/O automata with empty fairness sets are called *safe* I/O automata. □

For simplicity, we do not distinguish between fair and safe I/O automata if not explicitly needed. We let $s, t, \ldots$ range over states, and $a, b, \ldots$ over actions. We write $s \stackrel{a}{\rightarrow}_A t$, or just $s \stackrel{a}{\rightarrow} t$ if A is clear from the context, as a shorthand for $(s, a, t) \in steps(A)$. We abbreviate $acts(sig(A))$ by $acts(A)$ and similarly $in(sig(A))$, and so on. Since every input action is enabled in every state, I/O automata are said to be *input-enabled*.

Definition 2.2.3 (Parallel Composition)
Signatures $S_1, \ldots, S_n$ are *compatible* if for $1 \leq i, j \leq n$ with $i \neq j$ it holds that:

- $out(S_i) \cap out(S_j) = \emptyset$

- $int(S_i) \cap acts(S_j) = \emptyset$

I/O automata are called *compatible* if their action signatures are compatible. The composition $S = \prod_{i=1}^{n} S_i$ of compatible signatures $S_1, \ldots, S_n$ is defined by:

- $in(S) = \bigcup_{i=1}^{n} in(S_i) - \bigcup_{i=1}^{n} out(S_i)$
- $out(S) = \bigcup_{i=1}^{n} out(S_i)$
- $int(S) = \bigcup_{i=1}^{n} int(S_i)$

The parallel composition $A = A_1 \parallel \cdots \parallel A_n$ of compatible I/O automata $A_1, \ldots, A_n$ is defined by:

- $sig(A) = \prod_{i=1}^{n} sig(A_i)$
- $states(A) = \prod_{i=1}^{n} states(A_i)$
- $start(A) = \prod_{i=1}^{n} start(A_i)$
- $steps(A)$ is the set of triples $((s_1, \ldots, s_n), a, (t_1, \ldots, t_n))$ such that, for all $1 \leq i \leq n$, if $a \in acts(A_i)$, then $s_i \xrightarrow{a}_{A_i} t_i$, else $s_i = t_i$.
- $wfair(A) = \bigcup_{i=1}^{n} wfair(A_i)$ and $sfair(A) = \bigcup_{i=1}^{n} sfair(A_i)$.

□

Definition 2.2.4 (Hiding)
Let S be a signature and $\Lambda \subseteq local(S)$ a subset of its locally controlled actions. Then $hide_\Lambda(S)$ is defined to be the signature S', where $in(S') = in(S)$, $out(S') = out(S) - \Lambda$, and $int(S') = int(S) \cup \Lambda$.

Let A be an I/O automaton and $\Lambda \subseteq local(A)$ a subset of its locally controlled actions. Then $hide_\Lambda(A)$ is defined to be the I/O automaton A' obtained from A by replacing $sig(A)$ with $sig(A') = hide_\Lambda(sig(A))$. □

Definition 2.2.5 (Renaming)
Let S be a signature and σ be an injective mapping from actions to actions with $acts(A) \subseteq dom(\sigma)$. Then $rename_\sigma(S)$ is defined to be the signature S', where $in(S') = \sigma(in(S))$, $out(S') = \sigma(out(S))$, and $int(S') = \sigma(int(S))$.

Let A be an I/O automaton and σ be an injective mapping from actions to actions with $acts(A) \subseteq dom(\sigma)$. Then $rename_\sigma(A)$ is the I/O automaton A' which is defined as follows:

- $sig(A') = rename_\sigma(sig(A))$
- $states(A') = states(A)$

- $start(A') = start(A)$
- $steps(A') = \{(s, \sigma(a), t) \mid (s, a, t) \in steps(A)\}$
- $wfair(A') = \{\sigma(W) \mid W \in wfair(A)\}$ and $sfair(A') = \{\sigma(S) \mid S \in sfair(A)\}$

□

Definition 2.2.6 (Executions and Traces)
An *execution fragment* of an I/O automaton A is a finite or infinite sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of alternating states and actions of A beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \xrightarrow{a_{i+1}}_A s_{i+1}$. An *execution* is an execution fragment that begins with a start state. The set of all executions of A is denoted by $execs(A)$, the set of finite executions by $execs^*(A)$. Denote by $fstate(\alpha)$ the first state of α and, if α is finite, denote by $lstate(\alpha)$ the last state of α.

A state s of A is *reachable* if there exists a finite execution of A that ends in s.

The *trace* of an execution α, denoted by $trace(\alpha)$, is the subsequence of α consisting of all the external actions of A. We say that γ is a *trace* of A if there is an execution α of A with $\gamma = trace(\alpha)$. The set of all traces of A is denoted by $traces(A)$. Similarly, traces of a set of executions L are denoted by $traces(L)$.

Let γ be a finite or empty sequence over $ext(A)$ and s and t states of A. We say that (s, γ, t) is a *move* of A, written as $s \stackrel{\gamma}{\Rightarrow}_A t$, if A has a finite execution fragment α starting with s and ending with t, such that $trace(\alpha) = \gamma$. Empty sequences are denoted by *nil*.

Let $A = A_1 \parallel \cdots \parallel A_n$ and s be a state of A. Then, for any $1 \leq i \leq n$, define $s\lceil A_i$ to be s projected onto the i^{th} component. Now, let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be a sequence of alternating states and actions such that $s_k \in states(A)$ and $a_k \in acts(A)$, for all k, and α ends in a state if it is a finite sequence. Define $\alpha\lceil A_i$, where $1 \leq i \leq n$, to be the sequence obtained from α by projecting its states onto their i^{th} component and by removing each action not in $acts(A_i)$ together with its following state. □

Definition 2.2.7 (Fair Executions and Traces)
A set Λ of actions of an I/O automaton is said to be enabled in a state s, if for all $a \in \Lambda$ there is a state t such that $s \xrightarrow{a}_A t$.

An execution α of a fair I/O automaton A is *weakly fair* if the following conditions hold for each $W \in wfair(A)$:

- If α is finite then W is not enabled in the last state of α.
- If α is infinite then either α contains infinitely many occurrences of action from W, or α contains infinitely many occurrences of states in which W is not enabled.

An execution α of A is *strongly fair* if the following conditions hold for each $S \in sfair(A)$:

- If α is finite then S is not enabled in the last state of α.
- If α is infinite then either α contains infinitely many occurrences of actions from S, or α contains only finitely many occurrences of states in which W is enabled.

An execution α is *fair* if it is both weakly and strongly fair. The set of all fair executions of A are denoted by $\mathit{fairexecs}(A)$. The set of all traces originating from fair executions of A are denoted by $\mathit{fairtraces}(A)$. □

2.3 Live I/O Automata

Fair I/O automata may be generalized to *live* I/O automaton by specifying a liveness condition. Liveness conditions are subsets of the executions of a safe I/O automata which must not constrain its environment. This *environment-freedom* condition is ensured by setting up a game between the system and its environment, and the system is then environment-free if it can win the game no matter what moves the environment performs, i.e. if the system has a winning strategy. For more details concerning the intuition of the following liveness definitions the interested reader is referred to [GSSL93]. The user of our Isabelle environment will employ temporal logic instead of this game-theoretic treatment.

Definition 2.3.1 (Strategy)
Consider any safe I/O automaton A. A *strategy* defined on A is a pair of functions (g, f) where $g : \mathit{execs}^*(A) \times \mathit{in}(A) \rightarrow \mathit{states}(A)$ and $f : \mathit{execs}^*(A) \rightarrow (\mathit{local}(A) \times \mathit{states}(A)) \cup \{\perp\}$ such that

- $g(\alpha, a) = s$ implies $\mathit{lstate}(\alpha) \stackrel{a}{\rightarrow}_A s$
- $f(\alpha) = (a, s)$ implies $\mathit{lstate}(\alpha) \stackrel{a}{\rightarrow}_A s$

□

Definition 2.3.2 (Outcome of a Strategy)
Let A be a safe I/O automaton and (g, f) a strategy defined on A. Define an *environment sequence* for A to be any infinite sequence of symbols from $\mathit{in}(A) \cup \{\lambda\}$ with infinitely many occurrences of λ. Then define $R_{(g,f)}$, the *next-function induced by* (g, f), as follows: for any finite execution α of A and any environment sequence $\mathcal{I}$ for A,

$$R_{(g,f)}(\alpha, \mathcal{I}) \quad \equiv \quad \begin{cases} (\alpha a s, \mathcal{I}') & \text{if} \quad \mathcal{I} = \lambda\mathcal{I}',\, f(\alpha) = (a, s) \\ (\alpha, \mathcal{I}') & \text{if} \quad \mathcal{I} = \lambda\mathcal{I}',\, f(\alpha) = \perp \\ (\alpha a s, \mathcal{I}') & \text{if} \quad \mathcal{I} = a\mathcal{I}',\, g(\alpha, a) = s \end{cases}$$

Let α be any finite execution of A and $\mathcal{I}$ any environment sequence for A. The *outcome sequence* of (g, f) given α and $\mathcal{I}$ is the unique infinite sequence $(\alpha^n, \mathcal{I}^n)_{n \geq 0}$ that satisfies:

- $(\alpha^0, \mathcal{I}^0) = (\alpha, \mathcal{I})$ and
- For all $n > 0$, $(\alpha^n, \mathcal{I}^n) = R_{(g,f)}(\alpha^{n-1}, \mathcal{I}^{n-1})$.

The *outcome* $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})$ of the strategy (g, f) given α and $\mathcal{I}$ is the execution $lim_{n\rightarrow\infty}\alpha^n$, where $(\alpha^n, \mathcal{I}^n)_{n\geq 0}$ is the outcome sequence of (g, f) given α and $\mathcal{I}$ and the limit is taken under prefix ordering. □

Definition 2.3.3 (Environment-freedom)
A pair (A, L), where A is a safe I/O automaton and $L \subseteq execs(A)$, is *environment-free* if there exists a strategy (g, f) defined on A such that for any finite execution α of A and any environment sequence $\mathcal{I}$ for A, the outcome $\mathcal{O}_{(g,f)}(\alpha, \mathcal{I})$ is an element of L. The strategy (g, f) is called an *environment-free strategy* for (A, L). □

Definition 2.3.4 (Live I/O Automaton)
A *live I/O automaton* is a pair (A, L) where A is a safe I/O automaton and $L \subseteq execs(A)$ such that (A, L) is environment-free. We refer to the executions in L as the *live executions* of (A, L). Similarly, the members of $traces(L)$ are referred to as the *live traces* of (A, L). □

Definition 2.3.5 (Composition, Hiding, and Renaming of Live I/O Automata)
Live I/O automata $(A_1, L_1), \ldots, (A_n, L_n)$ are *compatible* if the safe I/O automata $A_1, \ldots, A_n$ are compatible. The *parallel composition* $(A_1, L_1) \parallel \cdots \parallel (A_n, L_n)$ of compatible live I/O automata is defined to be the pair (A, L) where $A = A_1 \parallel \cdots \parallel A_n$ and $L = \{\alpha \in execs(A) \mid \alpha\lceil A_1 \in L_1, \ldots, \alpha\lceil A_n \in L_n\}$.

Let (A, L) be a live I/O automaton and let $\Lambda \subseteq local(A)$. Then define $(A, L) - \Lambda$ to be the pair $(A - \Lambda, L)$.

Let σ be an injective mapping from actions to actions, A a live I/O automaton (A, L), and α be an execution of A. Define $\sigma(\alpha)$ to be the sequence that results from applying σ to every action in α. Then define $rename_\sigma(A, L)$ to be the pair $(\sigma(A), \sigma(L))$. □

Note that these operators are closed for live I/O automata in the sense that they produce new live I/O automata [GSSL93].

Definition 2.3.6 (Input Resistance)
Let A be a fair I/O automaton and Λ a subset of its locally controlled actions. We define Λ to be *input resistant* iff for each pair of reachable states s, t and each input action a, s enables Λ and $s \xrightarrow{a}_A t$ implies that t enables Λ. Thus, once Λ is enabled, it can only be disabled by the occurrence of a locally-controlled action. A fair I/O automaton is said to be *input resistant*, if every set in $sfair(A)$ is input resistant.

Lemma 2.3.7 (Fair I/O Automata are Live)
Every input resistant fair I/O automaton A induces a live I/O automaton $(safe(A), L)$ with $L = fairexecs(A)$, where $safe(A)$ denotes A without fairness sets.

Proof.
Essentially Theorem 1 of [RV96]. □

2.4 Refinement Notions and Compositionality

In this section we introduce the notion of *implementation relations* between I/O automata, show how to prove them via *simulations*, and mention the most important meta-theorems like compositionality and non-interference.

Definition 2.4.1 (Implementation)
Given two live I/O automata (C, L_C) and (A, L_A) with the same external actions, define the following implementation relations:

$$\begin{array}{lccl} \textit{Safe}: & C \preceq_S A & \text{iff} & \textit{traces}(C) \subseteq \textit{traces}(A) \\ \textit{Live}: & (C, L_C) \preceq_L (A, L_A) & \text{iff} & \textit{traces}(L_C) \subseteq \textit{traces}(L_A) \end{array}$$

The implementation relations are called safe and live trace inclusion, respectively, and fair trace inclusion — denoted by $\preceq_F$ —, if the liveness conditions L_C and L_A are induced by fairness sets. □

Definition 2.4.2 (Forward Simulation)
Let C and A be I/O automata with the same external actions. A *forward simulation* from C to A is a relation R over $\textit{states}(C) \times \textit{states}(A)$ that satisfies:

- If $s \in \textit{start}(C)$ then $R[s] \cap \textit{start}(A) \neq \emptyset$.
- If s is a reachable state of C, $s' \in R[s]$ a reachable state of A, and $s \overset{a}{\rightarrow}_C t$, then there is a state $t' \in R[t]$ such that $s' \overset{\gamma}{\Rightarrow}_A t'$ where $\gamma = a$ if $a \in \textit{ext}(A)$, otherwise $\gamma = \textit{nil}$.

We write $C \leq_F A$ if there is a forward simulation from C to A.

A specific and very useful form of forward simulations are *refinement mappings*.

Definition 2.4.3 (Refinement Mapping)
Let C and A be I/O automata with the same external actions. A *refinement mapping* from C to A is a function f from $\textit{states}(C)$ to $\textit{states}(A)$ that satisfies:

- If $s \in \textit{start}(C)$ then $f(s) \in \textit{start}(A)$.
- If s is a reachable state of C, and $s \overset{a}{\rightarrow}_C t$, then $f(s) \overset{\gamma}{\Rightarrow}_A f(t)$ where $\gamma = a$ if $a \in \textit{ext}(A)$, otherwise $\gamma = \textit{nil}$.

We write $C \leq_R A$ if there is a refinement mapping between C and A. □

For the following definition the notion of image-finiteness is needed: A relation R over $S_1 \times S_2$ is *image-finite* if for each $s_1 \in S_1$, $R[s_1]$ is a finite set.

Definition 2.4.4 (Backward Simulation)
Let C and A be I/O automata with the same external actions. A *backward simulation* from C to A is a total relation B over $states(C) \times states(A)$ that satisfies:

- If $s \in start(C)$ then $B[s] \subseteq start(A)$.
- If s is a reachable state of C, $s \xrightarrow{a}_C t$, and $t' \in R[t]$ a reachable state of A, then there is a state $s' \in R[s]$ such that $s' \stackrel{\gamma}{\Rightarrow}_A t'$ where $\gamma = a$ if $a \in ext(A)$, otherwise $\gamma = nil$.

We write $C \leq_B A$ if there is a backward simulation from C to A. Furthermore, if it is image-finite, we write $C \leq_{iB} A$. □

The following theorem stating the correctness of simulations is based on a specific correspondence between the executions of the low-level automaton C and the executions of the high-level automaton A, which is induced by the forward/backward simulation property. The following definition, which will be useful in other contexts as well, makes this correspondence explicit.

Definition 2.4.5 (R-Relation and Index Mappings)
Let C and A be safe I/O automata with the same external actions and let R be a relation over $states(C) \times states(A)$. Furthermore, let α and α' be executions of C and A, respectively, such that $\alpha = s_0 a_1 s_1 \cdots$ and $\alpha' = t_0 b_1 t_1 \cdots$. We say that α and α' are *R-related*, or *α' corresponds to α via R*, written $(\alpha, \alpha') \in R$, if there is a total, nondecreasing mapping[1] $m : \{0, 1, \ldots, |\alpha|\} \rightarrow \{0, 1, \ldots, |\alpha'|\}$ such that

- $m(0) = 0$,
- $(s_i, t_{m(i)}) \in R$ for all $0 \leq i \leq |\alpha|$,
- $trace(b_{m(i-1)+1} \cdots b_{m(i)}) = trace(a_i)$ for all $0 < i \leq |\alpha|$, and
- for all j, $0 \leq j \leq |\alpha'|$, there exists an i, $0 \leq i \leq |\alpha|$, such that $m(i) \geq j$.

The mapping m is referred to as an *index mapping* from α to α' with respect to R. □

[1] If, e.g., α is infinite, then the set $\{0, 1, \ldots, |\alpha|\}$ is supposed to denote the set of natural numbers (not including ∞), and $i \leq |\alpha|$ lets i range over all natural numbers but not ∞.

Theorem 2.4.6 (Correctness of Simulations)
Let A and C be I/O automata with the same external actions. Then the following holds:

- If $C \leq_F A$, then $C \preceq_S A$.
- If $C \leq_{iB} A$, then $C \preceq_S A$.

Proof.
Essentially Theorem 6.17 of [GSSL93], which makes heavy use of index mappings. □

For the next theorem the following definition is needed: An I/O automaton A has *finite invisible nondeterminism* if $start(A)$ is finite, and for any state s and any finite sequence γ over $ext(A)$, there are only finitely many states t such that $s \stackrel{\gamma}{\Rightarrow}_A t$.

Theorem 2.4.7 (Completeness of Simulations)
Let A and C be I/O automata with the same external actions, and let C have finite invisible nondeterminism. Then the following holds:

- If $C \preceq_S A$, then $\exists B : C \leq_F B \leq_{iB} A$.

Proof.
Essentially Theorem 4.22 of [LV95]. □

Note that we will not prove this completeness result in Isabelle, as the user of I/O automata does not need to rely on it. A further result from [LV95] states that the well-known history variables together with refinement mappings have the same expressiveness as forward simulations and that prophecy variables together with refinement mappings have the same expressiveness as backward simulations.

Theorem 2.4.8 (Compositionality for Executions and Traces)
Let $A = A_1 \parallel \cdots \parallel A_n$. Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be a sequence of alternating states and actions such that $s_k \in states(A)$ and $a_k \in acts(A)$, for all k, and α ends in a state if it is a finite sequence. Suppose γ is a sequence of actions in $ext(A)$. Then holds:

- $\alpha \in execs(A)$ iff for all i, $\alpha\lceil A_i \in execs(A_i)$ and $s_{j-1}\lceil A_i = s_j\lceil A_i$, if $a_j \notin acts(A_i)$.
- $\gamma \in traces(A)$ iff for all i, $\gamma\lceil A_i \in traces(A_i)$.

Proof.
Essentially Lemma 5 and Lemma 7 of [LT87]. □

Theorem 2.4.9 (Compositionality for Fair Executions and Traces)
Let $A = A_1 \parallel \cdots \parallel A_n$. Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be a sequence of alternating states and actions such that $s_k \in \mathit{states}(A)$ and $a_k \in \mathit{acts}(A)$, for all k, and α ends in a state if it is a finite sequence. Suppose γ is a sequence of actions in $\mathit{ext}(A)$. Then holds:

- $\alpha \in \mathit{fairexecs}(A)$ iff for all i, $\alpha\lceil A_i \in \mathit{fairexecs}(A_i)$ and $s_{j-1}\lceil A_i = s_j\lceil A_i$, if $a_j \notin \mathit{acts}(A_i)$.
- $\gamma \in \mathit{fairtraces}(A)$ iff for all i, $\gamma\lceil A_i \in \mathit{fairtraces}(A_i)$.

Proof.
Essentially Lemma 19 and Lemma 20 of [LT87]. □

Theorem 2.4.10 (Compositional Reasoning)
Let $(C, L) = (C_1, L_1) \parallel \cdots \parallel (C_n, L_n)$ and $(A, M) = (A_1, M_1) \parallel \cdots \parallel (A_n, M_n)$ be parallel compositions of live I/O automata, where the C_i and the A_i are compatible, respectively, and $\mathit{ext}(C_i) = \mathit{ext}(A_i)$ for every i. Then $\mathit{ext}(C) = \mathit{ext}(A)$ and the following holds:

- If $C_i \preceq_S A_i$ for every i, then $C \preceq_S A$.
- If $(C_i, L_i) \preceq_L (A_i, M_i)$ for every i, then $(C, L) \preceq_L (A, M)$.

Proof.
Essentially Theorem 8.10 of [Lyn96] and Theorem 3.33 of [GSSL93]. □

Theorem 2.4.11 (Non-Interference)
Let $A = A_1 \parallel \cdots \parallel A_n$ be a parallel composition of compatible I/O automata, and let $a \in \mathit{local}(A_i)$ be a locally-controlled action of A_i for some i. Suppose $\gamma = a_1 \cdots a_n$ is the subsequence of actions of an execution of A and $\gamma' = a_1 \cdots a_n a$. Then the following holds:

- If $\gamma'\lceil A_i$ is the subsequence of actions of an execution of A_i, then γ' is the subsequence of actions of an execution of A.

Proof.
Essentially Corollary 3 of [LT87]. □

2.5 Syntax for Automata Descriptions

Until now the state space of an I/O automaton merely consisted of an unstructured set of states. In order to be able to refer to specific state components explicitly, we will from now on assume, that every state space is described by a mapping σ from state variables to their values. This means that every I/O automaton A is equipped with a set of state variables $\mathcal{V}$ which yield a concrete state $s \in \mathit{states}(A)$ if being evaluated under σ. For any x in $\mathcal{V}$ we write $\sigma[\![x]\!]$ instead of $\sigma(x)$. The set of all state mappings σ is called **St**. Thus, $\mathit{states}(A) = \mathbf{St}(\mathcal{V})$.

I/O automata are specified using the *precondition/effect style* [Lyn96]. This means that the behaviour caused by actions is given by pre- and postconditions on the state space. Let us illustrate this format with the simple example of a buffer. The buffer *Buf* is modeled by a variable *queue* of type $(\mathit{bool})\mathit{list}$, which is initially empty. Then, actions and transitions are given in the precondition/effect style as:

input $S(m), m \in \mathit{bool}$	**output** $R(m), m \in \mathit{bool}$
post: $\mathit{queue} := \mathit{queue}@[m]$	**pre**: $\mathit{queue} = m : \mathit{rst}$
	post: $\mathit{queue} := \mathit{rst}$

Chapter 3

Temporal Logic and Live I/O Automata

We define a linear-time temporal logic (Temporal Logic of Steps – TLS) as property specification language for I/O automata and show how it can be used of advantage to support proofs of live implementation relations. In contrast to existing temporal logics, formulas are evaluated over sequences of alternating states and actions, which in addition may be finite.

3.1 Introduction

In §3.2 we will define a linear-time temporal logic, called Temporal Logic of Steps (TLS), that allows to express properties over executions of I/O automata. The motivation for this is twofold:

1. Temporal formulas can be used to specify liveness conditions for safe I/O automata. In standard I/O automata theory [GSSL93], liveness conditions are just sets of executions. The game-theoretic treatment of liveness in [GSSL93] deals only with meta theory, and gives no advice for how to describe such sets. In particular, there is no methodology that supports liveness proofs. Even for the restricted notion of fairness, refinement proofs do not follow a common scheme, but are often performed in an ad hoc fashion (see e.g. [Rom96, Lyn96]). Temporal logic, however, has been proven to be an adequate methodology for liveness proofs [GPSS80, Lam83].

2. Temporal logic can serve as a property specification language for I/O automata. Thus, property verification may also be applied as opposed to the usual techniques dealing with implementation relations. This is especially advantageous for model checking, which will be treated together with abstraction techniques in the subsequent chapter.

A number of temporal logics for program verification have already been proposed, most notable are Lamport's Temporal Logic of Actions (TLA) [Lam94] and the temporal logic developed by Manna and Pnueli [MP95]. Neither of these approaches, however, can be applied directly in our setting, because of two reasons. First, both logics do not treat actions explicitly, but evaluate temporal formulas over *state* sequences only. Even in TLA actions are only state changes. Second, both logics do not consider finite computations.

Thus, the following modifications are essential:

- To handle actions explicitly, we introduce step predicates, which depend on a state s, the corresponding action a, and the successor state s'. Thus, it is possible to evaluate temporal formulas over sequences of alternating states and actions.
- To handle finite computations, we introduce a stuttering action $\surd$, which is disjoint from all relevant action signatures, and add a single stuttering step $s_n \surd s_n$ at the end of all finite executions. Intuitively, we thereby ensure that finite executions may possibly be continued to infinity. In fact, we will show that under this semantics the evaluation of temporal formulas does not change when extending executions by infinite stuttering.

Subsequently, TLS is used to describe live I/O automata (§3.3). In particular, weak and strong fairness can now be specified by temporal formulas. As a result, fairness can now be defined with a uniform formula, without the former distinction between finite and infinite executions. Furthermore, we provide theorems which support simulation proofs between fair I/O automata.

3.2 A Temporal Logic of Steps

In this section we define the temporal logic TLS whose formulas are evaluated over finite and infinite sequences of alternating states and actions. Such sequences need not necessarily be generated by I/O automata. In particular, they are allowed to contain the stuttering action $\surd$. Thus, they are called *execution schemes* rather than executions.

Definition 3.2.1 (Execution Schemes)
Let $\mathcal{S}$ be a set of states and $\mathcal{A}$ be a set of actions with $\surd \notin \mathcal{A}$. A finite or infinite sequence of alternating states and actions $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ that begins with a state s_0 and, if it is finite, ends with a state s_n, is called an *execution scheme* over $(\mathcal{S}, \mathcal{A})$, iff $s_i \in \mathcal{S}$, $a_i \in \mathcal{A} \cup \{\surd\}$, and $s_{i+1} = s_i$, if $a_{i+1} = \surd$, for all i.

The *length* $|\alpha|$ of an execution scheme α is the number of actions occurring in α. Thus,

$$|\alpha| \equiv \begin{cases} n & \text{if } \alpha \text{ is finite and ends in } s_n \\ \infty & \text{if } \alpha \text{ is infinite} \end{cases}$$

Define the i-th *prefix* $\alpha|_i$ and the i-th *suffix* $_i|\alpha$ of α, for $0 \leq i \leq |\alpha|$[1], as

$$\begin{array}{rcl} \alpha|_i & \equiv & s_0\, a_1\, s_1 \ldots a_i\, s_i \\ _i|\alpha & \equiv & \begin{cases} s_i\, a_{i+1} s_{i+1} \ldots & \text{if } i < |\alpha| \\ s_{|\alpha|} & \text{if } \alpha \text{ is finite and } i = |\alpha| \end{cases} \end{array}$$

□

Execution schemes contain at least one state, namely s_0, and contain always the succeeding state s_i of a given action a_i. Furthermore, every execution of an I/O automaton is a specific execution scheme, which contains no stuttering steps $s_i \surd s_i$.

The basic ingredients of our temporal logic will be *step predicates* that refer to a variable v in a given state, its value in the successor state (refered to by v'), and to a further variable a reserved for actions. Let us illustrate step predicates with the simple example of a buffer. A buffer *Buf* may be modeled by a variable *queue* of type $(bool)list$, which is initially empty. Then, actions and transitions are given in the precondition/effect style as:

input $S(m), m \in bool$	**output** $R(m), m \in bool$
post: $queue := queue@[m]$	**pre**: $queue = m : rst$
	post: $queue := rst$

A possible step predicate over $\{queue\}$ would be $queue = [] \wedge a = S(true) \wedge queue' = [true]$ which states that the value *true* is added by the action S to the empty queue.

In the following formal definition we assume an underlying first-order assertion language $\mathcal{L}$ like the one defined in [Lam94].

Definition 3.2.2 (State, Action, and Step Predicates)
Let $\mathcal{V}$ be a set of variables. Any formula p of $\mathcal{L}$ over $\mathcal{V}$ is called a *state predicate* over $\mathcal{V}$. Its semantics is given in a postfix notation, letting $\sigma[\![p]\!]$ denote the value of p in σ. Formally:

$$\sigma[\![p]\!] \quad \equiv \quad p(\forall v.\, \sigma[\![v]\!]/v)$$

where $p(\forall v.\, \sigma[\![v]\!]/v)$ denotes the value obtained from p by substituting $\sigma[\![x]\!]$ for v, for all variables v in $\mathcal{V}$. Furthermore, we define

$$\sigma \models p \quad \equiv \quad \sigma[\![p]\!] = true$$

Let $\mathcal{A}$ be a set of actions. A *step predicate* p over $(\mathcal{V}, \mathcal{A})$ is a state predicate over $\mathcal{V} \uplus \mathcal{V}' \uplus \{a\}$ where $\mathcal{V}'$ is the set of primed variables in $\mathcal{V}$ and a is a reserved variable name for actions whose values stem from the set $\mathcal{A} \uplus \{\surd\}$. Its semantics is given in a postfix notation,

[1] The index ranges over the natural numbers including zero: if $|\alpha| = \infty$, then $i \leq |\alpha|$ is the same as $i < |\alpha|$.

letting $(\sigma, act, \tau)[\![p]\!]$ denote the value of p for a given action $act \in \mathcal{A} \uplus \{\surd\}$ in σ and its "successor" state τ. Formally:

$$(\sigma, act, \tau)[\![p]\!] \quad \equiv \quad p(\forall v.\, \sigma[\![v]\!]/v, \tau[\![v]\!]/v', act/a)$$

which means that $(\sigma, act, \tau)[\![p]\!]$ is obtained from p by replacing each unprimed variable v by $\sigma[\![v]\!]$, each primed variable v' by $\tau[\![v]\!]$, and the action variable a by act. We define:

$$(\sigma, act, \tau) \models p \quad \equiv \quad (\sigma, act, \tau)[\![p]\!] = \mathit{true}$$

A state predicate p is said to be *valid*, written as $\models p$, if p holds in every state. Thus,

$$\models p \quad \equiv \quad \forall \sigma \in \mathbf{St}.\, \sigma \models p$$

Similarly, a step predicate p is said to be *valid*, if every step is a p-step.

$$\models p \quad \equiv \quad \forall \sigma \in \mathbf{St}, \tau \in \mathbf{St}, act \in \mathcal{A} \uplus \{\surd\}.\, (\sigma, act, \tau) \models p$$

A step predicate, which contains only the action variable a, but no variables from $\mathcal{V} \cup \mathcal{V}'$, is called an *action predicate.* □

Action predicates will in most cases have the form $a \in \Lambda$, where Λ is a subset of the actions of an I/O automaton. As the stuttering action $\surd$ is not contained in any action signature, this means that those predicates are false for $\surd$. In the Isabelle implementation (§11.3) this constraint will be handled explicitly.

We will now define the temporal formulas of TLS. Note that these formulas will not merely refer to external actions, but to arbitrary actions, as they will be employed to reason about liveness and fairness, which affects internal actions as well.

Definition 3.2.3 (Temporal Formulas of TLS)
Let $\mathcal{V}$ be a set of state variables, $\mathcal{A}$ a set of actions, and $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ an execution scheme over $(\mathbf{St}(\mathcal{V}), \mathcal{A})$. Every step predicate S over $(\mathcal{V}, \mathcal{A})$ is a temporal formula. Furthermore, given any temporal formulas P and Q of TLS, the formulas $P \wedge Q$, $\neg P$, $\Box P$, and $\bigcirc P$ are temporal formulas of TLS as well, and the evaluation $\models$ is defined inductively as follows:

Step Predicate:	$\alpha \models S$	$\equiv$	if $\lvert\alpha\rvert \neq 0$ then $(s_0, a_1, s_1) \models S$ else $(s_0, \surd, s_0) \models S$
Conjunction:	$\alpha \models P \wedge Q$	$\equiv$	$\alpha \models P$ and $\alpha \models Q$
Negation:	$\alpha \models \neg P$	$\equiv$	$\neg \alpha \models P$
Always:	$\alpha \models \Box P$	$\equiv$	$\forall i \leq \lvert\alpha\rvert.\, {}_i\lvert\alpha \models P$
Next:	$\alpha \models \bigcirc P$	$\equiv$	if $\lvert\alpha\rvert \leq 1$ then $\alpha \models P$ else ${}_1\lvert\alpha \models P$

□

The peculiarities of a temporal logic incorporating finite computations manifest themselves in this context in the asymmetry of execution schemes: they contain one more state than actions in the finite case. We deal with this by adding a stuttering step that stutters the last step — thereby intuitively ensuring a possibly infinite continuation of the execution. The definition of $\models$ for step predicates reflects this intuition directly, the $\Box$ and $\bigcirc$ operators boil down to this case: $\Box P$ requires P to hold for all suffixes of a finite α, thus in particular for the last state s_n, which implies that the desired stuttering step $s_n \surd s_n$ is added as soon as the structural definition reaches a step predicate contained in P. The same holds for $\bigcirc P$, where in addition an explicit stuttering step is introduced, if the execution consists of less then two actions.

Definition 3.2.4 (Derived Temporal Formulas)
Given temporal formulas P and Q the following are temporal formulas as well.

Boolean Operators:	$P \Rightarrow Q$	$\equiv$	$\neg(P \wedge \neg Q)$
	$P \vee Q$	$\equiv$	$(\neg P) \Rightarrow Q$
Eventually:	$\Diamond P$	$\equiv$	$\neg\Box\neg P$
Leads To:	$P \rightsquigarrow Q$	$\equiv$	$\Box(P \Rightarrow (\Diamond Q))$

$\Box$

The formula $\Diamond P$ expresses that the temporal formula P holds now or at some time in the future. Note that for finite executions $\alpha = s_o a_1 s_1 \ldots s_n$ the formula $\Diamond P$ holds in particular, if P holds only for the final stuttering step $s_n \surd s_n$. The formula $P \rightsquigarrow Q$ asserts that any time P is true, Q is true then or at some later time.

Frequently used formulas are $\Box\Diamond P$ and $\Diamond\Box P$, which assert for infinite execution schemes that P is true infinitely often and that there is some time such that P is always true from that point on, respectively. Note, however, that their meaning for *finite* executions coincides: both express that P is true for the last state s_n.

Definition 3.2.5 (Validity)
A temporal formula P over $(\mathcal{V}, \mathcal{A})$ is said to be *valid*, written $\models P$, iff P holds for every execution scheme α over $(\mathbf{St}(\mathcal{V}), A)$, thus

$$\models P \;\equiv\; \forall \alpha.\, \alpha \models P$$

Let A be any safe I/O automaton. Then, a temporal formula P is said to be *valid* for A, written $A \models P$, iff P holds for every execution of A, thus

$$A \models P \;\equiv\; \forall \alpha \in \mathit{exec}(A).\, \alpha \models P$$

Let (A, L) be any live I/O automaton. Then, a temporal formula P is said to be *valid for (A, L), written $(A, L) \models P$, iff P holds for every execution in L, thus*

$$(A, L) \models P \;\equiv\; \forall \alpha \in L.\, \alpha \models P$$

$\Box$

In the sequel we formally capture the intuition that the evaluation of TLS formulas remains unchanged when adding infinite stuttering at the end.

Definition 3.2.6 (Adding Stuttering Steps)
Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ be an execution scheme. Then

$$\nabla\alpha \quad \equiv \quad \begin{cases} \alpha\surd s_n \surd s_n \ldots & \text{if } \alpha = s_0 a_1 s_1 \ldots a_n s_n \text{ is finite} \\ \alpha & \text{if } \alpha \text{ is infinite} \end{cases}$$

□

Lemma 3.2.7 (Single versus Infinite Stuttering)
Let P be a TLS formula and α an execution-scheme. Then,

$$\alpha \models P \qquad \text{iff} \qquad \nabla\alpha \models P$$

Proof.
By induction on the structure of P. As base case assume P is a step predicate. Assume further that $|\alpha| = 0$. This is the important case: $\alpha \models P$ equals $(s_0, \surd, s_0) \models P$, as a stuttering step is added. At the same time it equals $\nabla\alpha \models P$, as $\nabla\alpha$ contains already this stuttering step, which shows the desired property. If $|\alpha| \neq 0$, we get directly $(s_0, a_1, s_1) \models P$ for both sides. Now, as induction hypothesis assume the desired property holds for Q and R for every execution α_1. We have to distinguish whether P equals $\Box Q$, $\bigcirc Q$, $Q \wedge R$, or $\neg Q$. The first two cases reduce to the induction hypothesis, as $\nabla_i|\alpha_1 = {}_i|\nabla\alpha_1$ for all $i \leq |\alpha_1|$. The remaining cases are trivial. □

Notation. Since boolean operators appear twice in step predicates and temporal formulas, we determine that every step formula extends as far as possible to the right to avoid syntactical ambiguities. Furthermore, the operators $\Box$, $\bigcirc$, $\Diamond$, and $\neg$ have higher priority than $\wedge$ and $\vee$, which in turn precede $\Rightarrow$ and $\leadsto$.

3.3 Live I/O Automata

Having the temporal logic TLS at our disposal, we are now able to describe live I/O automata in a way that allows to argue about them via temporal reasoning, the tailored proof methodology for liveness proofs. The idea is to capture the subset of live executions of an I/O automaton by a temporal formula. In particular, fairness may be expressed by temporal formulas which is more convenient than the fairness sets of traditional I/O automata theory [Lyn96].

3.3.1 Liveness Conditions as Temporal Formulas

For the following definitions basic notions concerning live I/O automata are needed, for which the reader is referred back to the introductory notes in §2.

Definition 3.3.1 (Fair and Live I/O Automata via TLS)
A live I/O automaton (A, L), i.e. a safe I/O automaton A and a subset of its executions L, is described by specifying L indirectly in terms of a temporal formula F as follows:

$$L = \{\alpha \in \mathit{execs}(A) \mid \alpha \models F\}$$

That is, L consists of all the executions of A for which a certain temporal property F holds. We say that L is *induced* by F. If the context is clear we write (A, F) instead of $(A, \{\alpha \in \mathit{execs}(A) \mid \alpha \models F\})$.

Let A be any safe I/O automaton and $\Lambda \subseteq \mathit{local}(A)$ a subset of its locally-controlled actions. Then, weak fairness w.r.t. Λ may be expressed by means of the following temporal formula

$$\mathit{WF}_A(\Lambda) \quad \equiv \quad \Diamond\Box \mathit{Enabled}_A(\Lambda) \Rightarrow \Box\Diamond\; a \in \Lambda$$

where $\mathit{Enabled}_A(\Lambda)$ denotes the state predicate over A that holds in exactly those states of A where an action in Λ is enabled. Similarly, strong fairness w.r.t. Λ may be expressed by the formula $\mathit{SF}_A(\Lambda)$:

$$\mathit{SF}_A(\Lambda) \quad \equiv \quad \Box\Diamond \mathit{Enabled}_A(\Lambda) \Rightarrow \Box\Diamond\; a \in \Lambda$$

If obvious, we omit the subscript A and write $\mathit{WF}(\Lambda)$, $\mathit{SF}(\Lambda)$, and $\mathit{Enabled}(\Lambda)$. □

Of course, it has always to be ensured, that the temporal formula F used to describe the liveness behavior L for A really induces a live I/O automaton, i.e. that (A, L) is environment-free. In particular, this implies that L has to be a *liveness condition* [GSSL93], which means that any finite execution of A can be extended to an execution in L. Thus, no further safety property should be introduced by L.

For the fairness formulas $\mathit{WF}_A(\Lambda)$ and $\mathit{SF}_A(\Lambda)$ these conditions are always fulfilled, provided that A is input resistant. It is relatively easy to see that fairness formulas do not introduce any further safety, i.e. they represent liveness conditions. Furthermore, the proof that for input resistant A $\mathit{WF}_A(\Lambda)$ and $\mathit{SF}_A(\Lambda)$ induce a set of executions L such that (A, L) is environment-free, is the same as for traditional fair I/O automata (see [RV96] and Lemma 2.1.14). This is due to the fact that the "old" and "new" fairness notions coincide, which is shown below.

Note that the two requirements for L to be a liveness condition and for (A, L) to be environment-free have their analogous counterparts in TLA: the former corresponds to machine-closureness and the latter to receptiveness (see [AL93, GSSL93]).

Lemma 3.3.2 (Correspondence of Fairness Notions)
Let A be a safe I/O automaton and L a liveness condition which is induced by the TLS formula $F = \bigwedge_{i=1}^{n} WF_A(\Lambda_i) \wedge \bigwedge_{j=1}^{m} SF_A(\Lambda'_j)$. Furthermore, define B to equal A for the safety part, i.e. $X(B) = X(A)$ for every component $X \in \{sig, states, start, steps\}$ of A, and assume that B is fair with $wfair(B) = \bigcup_{i=1}^{n} \Lambda_i$ and $sfair(B) = \bigcup_{j=1}^{m} \Lambda'_j$. Then the fair executions of B induced by the fairness sets $wfair(B)$ and $sfair(B)$ equal those induced by the fairness formula F of A:

$$\{\alpha \in execs(A) \mid \alpha \models F\} = fairexecs(B)$$

Proof.
"$\subseteq$": Assume $\alpha \in execs(A)$ with $\alpha \models F$. If α is infinite, obviously $\alpha \in fairexecs(B)$, as desired. Otherwise, for every $1 \leq i \leq n$ the formula $\alpha \models WF_A(\Lambda_i)$ asserts, that if any action $a \in \Lambda_i$ is enabled in the last state s_n of α, then $a_{n+1} \in \Lambda_i$. By definition of TLS we get $a_{n+1} = \surd$ and $\surd \notin acts(A)$. As $\Lambda_i \subseteq acts(A)$, we have $a_{n+1} \notin \Lambda_i$, which implies that s_n cannot be enabled for any $a \in \Lambda_i$, as desired. The same holds for $SF_A(\Lambda'_j)$ for every $1 \leq j \leq m$. Thus, $\alpha \in fairexecs(B)$, as required.

"$\supseteq$": Assume $\alpha \in fairexecs(B)$. If α is infinite, obviously $\alpha \in execs(A)$ and $\alpha \models F$, as desired. Otherwise, the last state s_n of α is not enabled for any Λ_i or Λ'_j, with $1 \leq i \leq n$ and $1 \leq j \leq m$. Thus, the assumptions of all formulas $WF_A(\Lambda_i)$ and $SF_A(\Lambda'_j)$ are false, which makes them and thus the entire F true, as required. □

Note that for any live automaton (A, F) with F being given as a temporal formula the equation $(A, F) \models P = A \models (F \Rightarrow P)$ holds for any temporal formula P.

3.3.2 Live Implementation

In the sequel we show how temporal reasoning can be used of advantage to support refinement proofs which aim at live implementation. First, we extend refinement mappings and simulations (Def. 2.4.2 – 2.4.4) to the corresponding live notions, which means that they in addition transfer liveness from the implementation to the specification automaton.

Definition 3.3.3 (Live Simulations)
Let (C, F_C) and (A, F_A) be live I/O automata with the same external actions, whose liveness conditions are given via temporal formulas F_C and F_A. Then, for every $X \in \{F, R, iB\}$, we say that $(C, F_C) \leq_X^L (A, F_A)$ holds iff there is relation S such that if $C \leq_X A$ via S then for all $\alpha \in exec(C)$ and $\alpha' \in exec(A)$ with $(\alpha, \alpha') \in S$ holds: $\alpha \models F_C$ implies $\alpha' \models F_A$. □

Theorem 3.3.4 (Soundness of Live Simulations)
Let (C, F_C) and (A, F_A) be live I/O automata with the same external actions, whose liveness conditions are induced by the temporal formulas F_C and F_A. Then,

$$\text{if } (C, F_C) \leq_X^L (A, F_A) \text{ for some } X \in \{F, R, iB\} \text{ then } (C, F_C) \preceq_L (A, F_A).$$

Proof.
Essentially Lemma 6.18 in [GSSL93] except for the fact that liveness conditions are specified indirectly in terms of temporal formulas, together with Lemma 3.3.2. □

With this lemma we get the following proof method for showing that a live I/O automaton (C, F_C) implements another live I/O automaton (A, F_A) using temporal reasoning:

- Prove that S is a simulation from C to A, i.e. prove the safety part.
- Assume that α and α' are arbitrary executions of C and A, respectively, which fulfill $(\alpha, \alpha') \in S$ and $\alpha \models F_C$. Thus, α is assumed to be live.
- Prove that $\alpha' \models F_A$. Thus, the corresponding execution is proved to be live as well.

In this proof method, F_C is evaluated over executions of C, whereas F_A is evaluated over executions of A. Thus, it is not sufficient to merely apply temporal tautologies for the liveness proof. Rather there have to be means to switch between properties over an execution α of C and those of the corresponding execution α' of A.

For refinement proofs restricted to fairness, these switches can be minimized by employing them at only two distinct places in the proof. Moreover, the alternations always follow a common scheme, as they have to be performed only for particular patterns of temporal properties. For these patterns the following lemmas apply, the use of which will be demonstrated by a schematic example below showing their general applicability for the switches of any fairness proof.

Lemma 3.3.5 (Transforming Fairness from C to A)
Let C and A be safe I/O automata with the same external actions and let R be a relation between $states(C)$ and $states(A)$. Suppose α and α' are executions of C and A, respectively, with $(\alpha, \alpha') \in R$. Then the following holds for every subset $\Lambda \subseteq ext(C)$ of the common external actions:

$$\alpha \models \Box\Diamond\, a \in \Lambda \quad \text{iff} \quad \alpha' \models \Box\Diamond\, a \in \Lambda$$

Proof.
We prove only one direction by contradiction, the other is analogous. Let $\alpha' \models \Box\Diamond a \in \Lambda$ and let m be an arbitrary index mapping from α to α' w.r.t. R. Furthermore, assume $\alpha \not\models \Box\Diamond a \in \Lambda$. Then, $\alpha \models \Diamond\Box a \notin \Lambda$, which means that there is an index i with ${}_i|\alpha \models \Box a \notin \Lambda$.

$$\begin{array}{r|ccc} \alpha' \in execs(A) & \Diamond\Box Enabled_A(\Lambda) & \longrightarrow & \Box\Diamond\ a \in \Lambda \\ & \downarrow & & \uparrow \\ & \text{(Lemma 3.3.6)} & & \text{(Lemma 3.3.5)} \\ & \downarrow & & \uparrow \\ \alpha \in execs(C) & \Diamond\Box Enabled_C(\Lambda) & \longrightarrow & \Box\Diamond\ a \in \Lambda \end{array}$$

Figure 3.1: Schematic example of a simulation proof involving fairness

Thus, in $trace(_i|\alpha)$ no actions of Λ appear. By the trace correspondence which holds for all executions that correspond via an index mapping (Lemma 6.14 in [GSSL93]) and the fact that Λ contains only external actions, no actions of Λ occur in $_{m(i)}|\alpha$ either. Thus, $_{m(i)}|\alpha \models \Box\ a \notin \Lambda$, which implies by the definition of $\Diamond$, that $\alpha' \models \Box\Diamond\ a \notin \Lambda$. This gives the desired contradiction to the initial assumption. $\Box$

Lemma 3.3.6 (Transforming Fairness from A to C)
Let C and A be safe I/O automata with the same external actions and let R be a relation between $states(C)$ and $states(A)$. Suppose α and α' are executions of C and A, respectively, with $(\alpha, \alpha') \in R$. Furthermore, let P and P' be state predicates over C and A, respectively, such that for all reachable states s of C and reachable states t of A with $(s,t) \in R$ it holds that: if $t \models P'$, then $s \models P$. Then,

$$\text{if} \quad \alpha' \models \Diamond\Box P' \quad \text{then} \quad \alpha \models \Diamond\Box P.$$

Proof.
Let m be an arbitrary index mapping from α to α' w.r.t. R. Assume $\alpha' \models \Diamond\Box P'$, which means that there is an index j such that $_j|\alpha' \models \Box P'$. Thus, for each state t in $_j|\alpha'$ we have $t \models P'$. Then, by condition 4 of the definition of an index mapping (Def. 2.4.5) and the fact that index mappings are nondecreasing we get the existence of an index i such that for all $i \leq k \leq |\alpha|$ holds $m(k) \geq j$. By condition 2 of the definition of an index mapping we get that for each state s of α with index k $s \models P$ holds. Thus, $_i|\alpha \models \Box P$, as $i \leq k$, which finally gives $\alpha \models \Diamond\Box P$ by definition. $\Box$

Example 3.3.7
Let $CL = (C, WF_C(\Lambda))$ and $AM = (A, WF_A(\Lambda))$ be two live I/O automata. In order to show $CL \preceq_L AM$ according to the proof method above, we select a relation R and prove it to be a simulation. For the liveness part (see Figure 3.1) we choose arbitrary α and α' with $(\alpha, \alpha') \in R$ and assume $\alpha \models WF_C(\Lambda)$. In order to show $\alpha' \models WF_A(\Lambda)$ we can assume $\alpha' \models \Diamond\Box Enabled_A(\Lambda)$. Here, Lemma 3.3.6 allows us to switch from α' to α, thus we get $\alpha \models \Diamond\Box Enabled_C(\Lambda)$, provided that the state predicate $Enabled_A(\Lambda)$ implies $Enabled_C(\Lambda)$ for all $(s,t) \in R$. From the assumption $\alpha \models WF_C(\Lambda)$ we get $\alpha \models \Box\Diamond a \in \Lambda$. Here, Lemma 3.3.5 allows us to switch back from α to α', thus we get $\alpha \models \Box\Diamond\ a \in \Lambda$, as desired. The

readers may easily convince themselves that this methodology applies to any fairness proof in general. □

Note, however, that these switching lemmas do not generalize to arbitrary temporal properties. For example, Lemma 3.3.6 does not hold for $\Box\Diamond$ instead of $\Diamond\Box$ as P' in α' could hold infinitely often, but only at internal states of A added by the simulation R, so that the desired property does not transform to the execution α of C. However, for specific refinement mappings these lemmas hold for arbitrary temporal properties. This will be the key idea for temporal abstractions which will be introduced in §4.

3.4 Conclusion and Related Work

We defined TLS as a property specification language for I/O automata. Thus, a framework is provided, in which proofs of live implementation relations can be treated adequately. In particular, for the restricted, but important case of fairness, we proved theorems which support the transfer of temporal formulas from the specification to the implementation automaton and vice versa (Thm. 3.3.5 – 3.3.6). Previously, this has been done without temporal logic either by complicated reasoning about index mappings (Definition 2.4.5) [Lyn96] or in a more informal way [Rom96].

As a result we get a verification framework which is, we claim, especially well suited to handle proofs of implementation relations. It treats both safety and liveness aspects in a tailored way, namely by automata theory and temporal logic, respectively.

Related Work. Despite of the differences already mentioned in the introduction, TLA [Lam94] is of course closely related to TLS. In fact, [Lam94] guided us during the development of TLS. However, there are some further remarkable differences[2].

TLA incorporates arbitrary stuttering for every action directly in the basic model, which is therefore more complicated than the basic I/O automata model. On the other hand, the refinement concept of TLA is simpler than that for I/O automata, as both specification and implementation can be evaluated over the same sequences of states, whereas in refinements of I/O automata stuttering has to be added explicitly by internal actions.

Concerning our purposes this in particular implies that TLS sequences generated by I/O automata executions contain at most one stuttering step at the end. This makes it more reasonable to employ a next-time operator $\bigcirc$, which in TLA would not make sense because of arbitrary stuttering. Note, however, that $\bigcirc$ should be used carefully in combination with parallel composition, as the interleaving model of I/O automata does not guarantee that formulas containing $\bigcirc$ carry over from components to their composition. Thus, $\bigcirc$ has limitations concerning the intuition of specifications. Concerning proof rules, however, it turned out to be advantageous (cf. the experiences in §11.3).

[2]The relation to [Krö87, MP95] will be discussed in §11.3 in detail.

Chapter 4

Abstraction and Model Checking

We develop an abstraction theory for I/O automata which allows us to integrate deductive verification and model checking. Abstraction rules are provided for properties expressed both as temporal formulas (proof obligation $(C, F_C) \models P$*) and as I/O automata (proof obligation* $(C, F_C) \preceq_L (P, F_P)$*). A major advantage is that the theorem prover, verifying the abstraction's correctness, reasons about steps only, whereas reasoning about entire system runs is left to the model checker. This applies to both the simpler safety and the inherently difficult liveness proofs, whereas for the latter the availability of temporal logic is of vital importance. For the safety part, a completeness result for relational abstractions w.r.t. a restricted notion of implementation is shown. Furthermore, we give tailor-made translations into logics of model checkers selected for the two kinds of proof obligations. A running example of an abstract cockpit alarm system illustrates the method.*

4.1 Introduction

The purpose of this chapter is to combine the two major paradigms for the verification of reactive, distributed systems: *theorem proving* and *model checking*. The advantages of each approach are well known: model checking is automatic but limited to systems of relatively small finite state space, theorem proving requires user interaction but can deal with arbitrary systems.

Abstraction techniques [CGL92, LGS+95, GL93, Mer97, Kur87, DGG97, SLW95, Wol86, MN95, HS96] promise to integrate the two approaches: in a first step, the original system C is reduced to a smaller model A. If C is large or infinite, this step will in general require interactive proof techniques. In a second step, the smaller system A is analyzed using automatic tools.

Usually, the smaller system A is obtained by partitioning the state space of C via a function h between the two state spaces [CGL92]. If h is a homomorphism (*abstraction function*),

$$\begin{array}{l|cccccc}
\alpha' \in execs(A) & h(s_0) & a_1 & h(s_1) & a_2 & h(s_2) & \ldots \\
 & \uparrow & & \uparrow & & \uparrow & \\
\alpha \in execs(C) & s_0 & a_1 & s_1 & a_2 & s_2 & \ldots
\end{array}$$

Figure 4.1: Functional correspondence

the abstraction is guaranteed to be sound, i.e. if the abstract system satisfies a property so does the original system.

In I/O automata theory, an alternative intuition for h is to regard it as a special refinement mapping. Therefore, abstraction and implementation are in some sense just two different sides of the same medal: if A is an abstraction of C, then C is a safe implementation of A. The other direction does not hold in general, as the property of being a homomorphism means that h neglects the difference between internal and external actions. However, we could define a stronger implementation notion by postulating schedule inclusion instead of trace inclusion, where schedules denote the action subsequence of executions. In fact, we will show in §4.3 that the slightly more general forward/backward abstraction relations are complete w.r.t. this notion of implementation.

The key concept of proof techniques for I/O automata is the possibility to derive trace inclusion (a property about *executions*) from some kind of simulation (properties about *steps*). Therefore, the desired global behaviour can be reduced to simpler local proof obligations.

Of course, this applies to abstraction functions as well, as they are specific simulations[1]. However, an important feature of our abstraction theory is, that this "localizing of properties" applies to liveness properties as well, which are known to be harder to prove under abstraction than safety properties. Here we profit in particular from treating liveness conditions as temporal formulas, as they provide two levels of description: the step level and the temporal level. We will show in §4.2.1 that implications between temporal formulas expressing properties about different automata C and A can be reduced to implications between the step predicates occurring in them. The reason is the close correspondence between those executions of C and A which are related by the homomorphism h: the abstract execution α' is always a direct mapping of the concrete execution α under h (see Fig. 4.1), which means that α and α' always proceed in lock-step.

As a result, the simpler step level is associated with interactive theorem proving, whereas automatic verification tools can be employed to reason about entire system runs. Especially for liveness proofs this represents a significant proof facilitation, which goes beyond the support for live simulations provided already in the previous chapter. In addition, further

[1]This is the key idea of the usual abstractions e.g. in [CGL92].

techniques are developed, which allow to strengthen liveness of the abstract model or to weaken liveness of the concrete model. Therefore, much of the tedium that is usually associated with proving liveness properties may be left to the model checker.

In §4.2.2 and §4.2.3 abstraction rules are provided for properties expressed both as temporal formulas (proof obligation $(C, F_C) \models P$) and as I/O automata (proof obligation $(C, F_C) \preceq_L (P, F_P)$). Therefore, we may always choose the more adequate description technique to describe system properties (see Examples 4.2.16 and 4.2.12).

Assuming that Isabelle and the formalization of I/O automata in it (which we will present in §8 – §11) are correct, the only remaining source of errors is the model checker which is treated like an oracle by Isabelle. Note that this includes the interface between the model checker and Isabelle, which is particularly critical because we need to ensure that the theorem prover formalizes exactly the logic the model checker is based on. Therefore, in §4.4 we furthermore investigate interfaces to the model checkers STeP [BBC+96] and μcke [Bie97a], tailored for the proof obligations $(C, F_C) \models P$ and $(C, F_C) \preceq_L (P, F_P)$, respectively. Concerning these translations emphasis is laid on structure preservation, which facilitates the soundness and completeness proofs for the translations and ensures a maximum of efficiency, which is of vital importance for the practical applicability of model checkers.

An example of an abstract cockpit alarm system illustrates the abstraction method as well as the model checker translations.

The chapter is organized as follows: the abstraction theory for functions is presented in §4.2, its extension to relations in §4.3, and the model checker translations in §4.4.

4.2 A Theory of Abstraction

In this section we investigate abstractions which are caused by homomorphic functions. In §4.2.1 we will first show how properties about executions can be reduced to properties about steps, if such a function exists. Then, abstraction rules will be derived for properties both expressed as temporal formulas and as automata. This is done in in §4.2.2 and §4.2.3, respectively. We shall see that the desired correctness of abstractions boils down to implications between execution properties.

Note that in this chapter we always assume that liveness conditions of I/O automata are given as TLS formulas.

4.2.1 Basic Theory

As mentioned in the introduction, the key concept of the abstraction theory is a specific correspondence between abstract and concrete executions, which is induced by a function between the two state spaces (cf. Fig. 4.1).

Definition 4.2.1 (Functional Correspondence)
Let C and A be two I/O automata with the same set of actions. Let $\alpha = s_0 a_1 s_1 \ldots$ be an execution scheme over $states(C)$ and $acts(C)$ and $\alpha' = t_0 b_1 t_1 \ldots$ be an execution scheme over $states(A)$ and $acts(A)$. Furthermore let h be a function from $states(C)$ to $states(A)$. Then α and α' are said to *functionally correspond* under h, if the index mapping m from α to α' w.r.t. h is the identity, or explicitly, if $t_i = h(s_i)$ and $b_i = a_i$ for all i. □

Note that for every execution scheme α over $states(C)$ and $acts(C)$ and a given function h there is exactly one corresponding execution scheme α' over $states(A)$ and $acts(A)$ under h. Therefore, α' is denoted by $c_h(\alpha)$ in the sequel.

The following definition formalizes implications between execution properties of the abstract and the concrete system. The executions have to functionally correspond to one another, which lays the foundation to "localize" these implications later on.

Definition 4.2.2 (Global Weakening and Strengthening)
Let C and A be safe I/O automata with the same set of actions and h be a function from $states(C)$ to $states(A)$. A is said to a be an automaton weakening of C w.r.t. h if for all execution schemes of C holds:

$$\alpha \in exec(C) \Rightarrow c_h(\alpha) \in exec(A)$$

Similarly, let P and Q be properties of execution schemes of C and A, respectively. Then Q is said to be a temporal weakening (resp., strengthening) of P w.r.t. h, iff for all executions $\alpha \in exec(C)$ holds:

$$\alpha \models P \Rightarrow c_h(\alpha) \models Q \qquad (\text{resp., } \; c_h(\alpha) \models Q \Rightarrow \alpha \models P)$$

We collectively refer to temporal and automaton weakenings/strengthenings as *global* weakenings/strengthenings. □

Note that $\neg Q$ is a temporal weakening of $\neg P$ w.r.t. h, if Q is a temporal strengthening of P w.r.t. h. Further note that there is no principal difference between temporal and automaton weakenings. The temporal property of an automaton weakening is just the property of being an execution of C, respectively A.

In the following we show how global weakenings and strengthenings can be reduced from temporal formulas to step predicates and from automata to transitions. First, we define weakening and strengthening on the step level as well.

Definition 4.2.3 (Local Weakening and Strengthening)
Let $\mathcal{S}$ and $\mathcal{T}$ be sets of states, and $\mathcal{A}$ be a set of actions, and h be a function from $\mathcal{S}$ to $\mathcal{T}$. Let Q be a transition predicate over $\mathcal{T} \times \mathcal{A} \times \mathcal{T}$ and P be a transition predicate over $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$. Then,

- Q is a *step weakening* of P, iff $\forall s, t \in \mathcal{S}, a \in \mathcal{A}.\, (s, a, t) \models P \Rightarrow (h(s), a, h(t)) \models Q$.

- Q is a *step strengthening* of P, iff $\forall s\ t \in \mathcal{S},\ a \in \mathcal{A}.(h(s), a, h(t)) \models Q \Rightarrow (s, a, t) \models P$.

Step weakenings/strengthenings are also called *local* weakenings/strengthenings. $\square$

The following important theorem permits the desired "localizing" of temporal weakenings and strengthenings.

Theorem 4.2.4 (Localizing Temporal Weakenings and Strengthenings)
Let P be a temporal formula built from transition predicates P_i by the connectives of TLS, and assume given local weakenings and strengthenings P_i^- and P_i^+ for every P_i. Then,

- P^- is a temporal weakening of P, where P^- denotes the formula obtained from P by replacing every positive (resp., negative) occurrence of P_i by P_i^- (resp.,P_i^+).
- P^+ is a temporal strengthening of P, where P^+ denotes the formula obtained from P by replacing every positive (resp., negative) occurrence of P_i by P_i^+ (resp.,P_i^-).

Proof.
By induction on the structure of the formulas. See §11.5 for the proof in Isabelle. $\square$

Therefore, temporal formulas F_C and F_A, which have the same structure w.r.t. the temporal operators, can be shown to be weakenings/strengthenings of each other simply by showing weakening/strengthening for the different transition predicates occurring in them. If all the transition predicates occurring in F_C and F_A are both weakenings *and* strengthenings of each other, we say, that F_A is the *corresponding abstract property* to F_C.

Note that this theorem represents a significant improvement for the case of functionally corresponding executions in comparison to the Theorems 3.3.5 and 3.3.6 in the preceding chapter. Whereas the theorems of the previous chapter allowed localizations only for two specific patterns of temporal formulas, the theorem above applies to any temporal formula. This is, of course, only possible, because we require a neater correspondence between the executions than that established by general simulations or refinement mappings.

We proceed with reducing automaton weakenings to implications on the step level.

Definition 4.2.5 (Abstraction Functions)
Let C and A be safe I/O automata with the same set of actions and h a function from $states(C)$ to $states(A)$. Then, h is an *abstraction function*, iff it satisfies the following conditions:

- If $s \in start(C)$ then $h(s) \in start(A)$.
- If s is a reachable state of C, and $s \xrightarrow{a}_C t$, then $h(s) \xrightarrow{a}_A h(t)$.

We write $C \leq_{AF} A$, if there is an abstraction function from C to A. $\square$

Lemma 4.2.6 (Localizing Automaton Weakenings)
Let C and A be safe I/O automata with the same set of actions and h a function from $\mathit{states}(C)$ to $\mathit{states}(A)$. Then,

$$\text{if } C \leq_{AF} A \text{ via } h, \text{ then } A \text{ is an automaton weakening of } C \text{ w.r.t. } h.$$

Proof.
Simple consequence of the Execution Correspondence Theorem in [GSSL93]. Explicitly, let $\alpha = s_0 a_1 s_1 \ldots$ be an execution of C. Then $c_h(\alpha) = h(s_0)\, a_1\, h(s_1) \ldots$ is an execution of A, as h is an abstraction function, which guarantees that $h(s_0) \in \mathit{start}(A)$ and $h(s_i) \xrightarrow{a}_A h(t_i)$ for all $i \geq 0$. This gives the desired weakening property immediately. $\square$

Abstraction functions may be extended to liveness in the same way as done for simulations and refinement mappings in the previous chapter.

Definition 4.2.7 (Live Abstraction Functions)
Let (C, F_C) and (A, F_A) be live I/O automata with the same set of actions. Then,

$$\begin{array}{lll} (C, F_C) \leq^L_{AF} (A, F_A) & \text{iff} & \text{there is an } h \text{ such that } C \leq_{AF} A \text{ via } h \text{ and} \\ & & F_A \text{ is a global weakening of } F_C \text{ w.r.t. } h. \end{array}$$

$\square$

Theorem 4.2.8 (Soundness of Live Abstraction Functions)
Let (C, F_C) and (A, F_A) be live I/O automata with the same set of actions. Then,

$$\text{if } (C, F_C) \leq^L_{AF} (A, F_A), \quad \text{then } (C, F_C) \preceq_L (A, F_A).$$

Proof.
Analogous to the proof of Lemma 3.3.4. Explicitly, let $\alpha \in \mathit{exec}(C)$ and $\alpha \models F_C$. Then we have to show that $c_h(\alpha) \in \mathit{exec}(A)$, $c_h(\alpha) \models F_A$, and $\mathit{trace}(c_h(\alpha)) = \mathit{trace}(\alpha)$. The first is ensured by Lemma 4.2.6, the second is due to the weakening relation between F_A and F_C, and the last is true, as α and $c_h(\alpha)$ coincide on all actions, even the internal ones. $\square$

This soundness result will be a prerequisite for the abstraction rules in §4.2.3. A completeness result will be shown for the more general concept of forward/backward abstraction relations in §4.3.

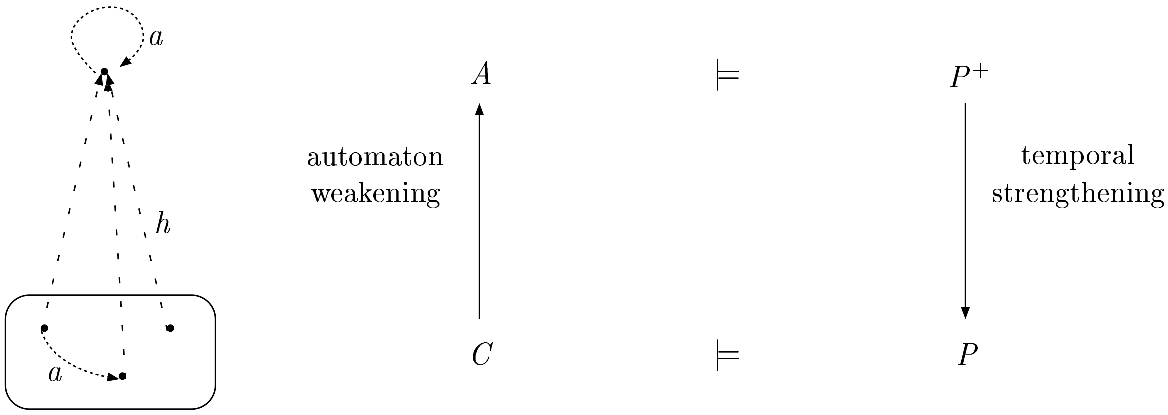

Figure 4.2: Abstraction Rule (Abs-P_1)

4.2.2 Abstraction Rules for Temporal Formulas

We will now present abstraction rules for properties which are expressed as temporal formulas. Note that we always take *temporal* weakenings/strengthenings as assumptions for these rules. Therefore, after applying an abstraction rule, Lemma 4.2.4 has to be used to reduce those weakenings/strengthenings to local ones.

Theorem 4.2.9 (Abstraction for Safe I/O Automata)
Let C and A be safe I/O automata with the same set of actions, and let h be a function from $states(C)$ to $states(A)$. Suppose $C \leq_{AF} A$ via h, and let P^+ be a temporal strengthening of P w.r.t. h. Then:

$$\frac{A \models P^+}{C \models P}\ (Abs\text{-}P_1)$$

Proof.
Let $\alpha \in exec(C)$. Then it has to be shown that P holds for α. As $C \leq_{AF} A$ via h, we get by Lemma 4.2.6 that A is an automaton weakening of A w.r.t. h. Therefore, $c_h(\alpha) \in exec(A)$. Because of $A \models P^+$ every execution of A satisfies P^+, therefore in particular $c_h(\alpha) \models P^+$. Finally, as P^+ is a temporal strengthening of P w.r.t. h, we get $\alpha \models P$. □

We illustrate the abstraction rule by a simple example, which will serve as a "running example" during the entire chapter. It represents a very rough simplification of the cockpit alarm system which will be handled in §12.2 to its full extent. Nevertheless the simplified version is interesting enough to demonstrate all ideas to be developed. See Fig. 4.2 as well.

Example 4.2.10
The alarm management of a cockpit control system can be described in a very abstract way by a stack. Alarms, which are initiated by the physical environment, are stored, then

handled by the pilot and finally acknowledged, which means that the respective alarm is removed from the stack. When adding a new alarm to the stack, any older occurrences of this particular alarm are removed, such that only the most urgent instance of an alarm has to be treated by the pilot. There are quite a number of alarms

$$Alarms = \{PonR, Fuel, Eng, \ldots\}$$

which in the original specification made it impossible to verify the system via model checking. Note that the order of alarms in the stack has to be respected. However, for proving properties which concern merely a single alarm, for example the important alarm *PonR*, (Point of no Return), abstraction may be applied.

The I/O automaton $Cockpit_C$ is modeled by a list, called *stack*, which is initially empty.

Transitions of $Cockpit_C$:

input $Alarm(a), a \in Alarms$	**output** $Ack(a), a \in Alarms$
post: $stack := a : [x \in stack.\ x \neq a]$	**pre**: $stack = a : stack'$
	post: $stack := stack'$

The properties we want to prove about $Cockpit_C$ are the following:

$$P_1 \equiv \Box(a = Alarm(PonR) \Rightarrow \bigcirc PonR \in stack)$$

"Whenever PonR arrives, it is immediately stored in the stack"

$$P_2 \equiv \Box(a \neq Alarm(PonR)) \Rightarrow \Box\neg(PonR \in stack)$$

"If PonR never arrives, the system will never pretend this"

For both properties it is merely relevant whether *PonR* is in the stack or not. Therefore, we construct the abstract I/O automaton $Cockpit_A$ which replaces the alarm stack by the boolean variable *PonR-in*, initially *false*, which indicates if *PonR* is stored (cf. Fig. 4.3).

Transitions of $Cockpit_A$:

input $Alarm(a), a \in Alarms$
post: **if** $a = PonR$ **then** $PonR\text{-}in := true$

---

output $Ack(a), a \in Alarms$
pre: $a = PonR \Rightarrow PonR\text{-}in$
post: **if** $a = PonR$ **then** $PonR\text{-}in := false$

The abstraction function is obviously defined as follows:

$$\begin{aligned} h_1 &: (\alpha)list \to bool \\ h_1(s) &\equiv PonR \in s \end{aligned}$$

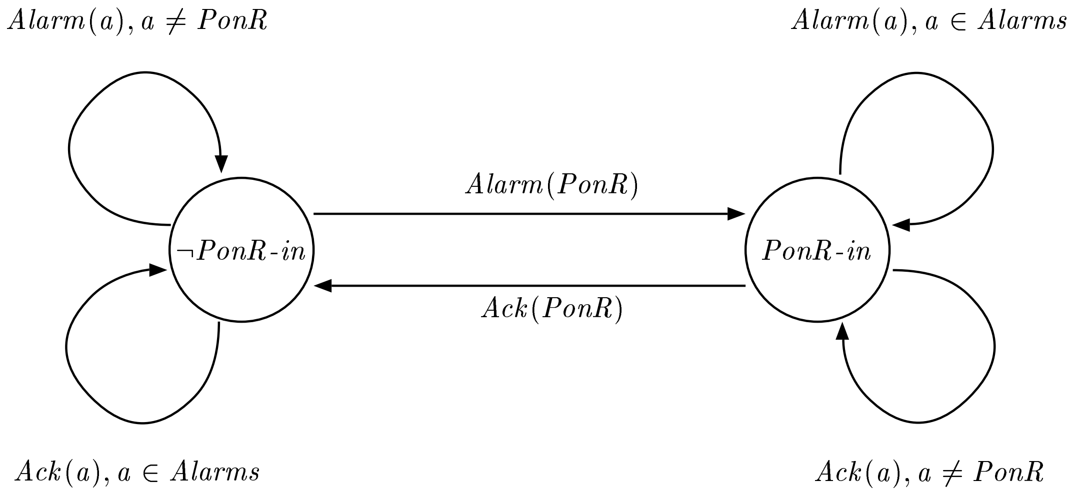

Figure 4.3: Transitions of $Cockpit_A$

As $Cockpit_A$ has been designed already with h_1 in mind, the property $Cockpit_C \leq_{AF} Cockpit_A$ is easily established[2]. Therefore it remains to show that the corresponding abstract properties P_i^+ for $Cockpit_A$, defined as

$$\begin{array}{lcl} P_1^+ & \equiv & \Box a = Alarm(PonR) \Rightarrow \bigcirc PonR\text{-}in \\ P_2^+ & \equiv & \Box a \neq Alarm(PonR) \Rightarrow \Box \neg PonR\text{-}in \end{array}$$

are temporal strengthenings of the concrete P_i. We restrict our attention to the first strengthening proposition, which reduces by the localizing lemma for abstractions (Lemma 4.2.4) to the following two obligations

$$\begin{array}{llll} c_h(\alpha) \models a = Alarm(PonR) & \Rightarrow & \alpha \models a = Alarm(PonR) & (1) \\ \alpha \models PonR \in stack & \Rightarrow & c_h(\alpha) \models PonR\text{-}in & (2) \end{array}$$

for any execution α of $Cockpit_C$. The first obligation is trivially fulfilled, as $c_h(\alpha)$ distinguishes itself from α only in the state but not in the action components. Assuming $\alpha = s_0 a_1 s_1 \cdots$ and $|\alpha| \neq 0$ proof obligation (2) reduces to

$$(s_0, act, s_1) \models PonR \in stack \quad \Rightarrow \quad (h(s_o), act, h(s_1)) \models PonR\text{-}in$$

which holds as $h(s_0) = PonR \in s_0$. The case $|\alpha| = 0$ holds analogously. Note that this reduction to a first-order property follows the schemes given in Lemma 4.2.4 and can therefore be completely automated by the theorem prover.

Therefore, the initial goals $Cockpit_C \models P_i$ have been reduced to $Cockpit_A \models P_i^+$ which can be verified by a model checker, e.g. the one of STeP (see §4.4.2). $\Box$

[2]Note that the fact that there are no duplicates in the stack is needed to show that $Ack(PonR)$ causes the transition from $PonR\text{-}in$ to $\neg PonR\text{-}in$.

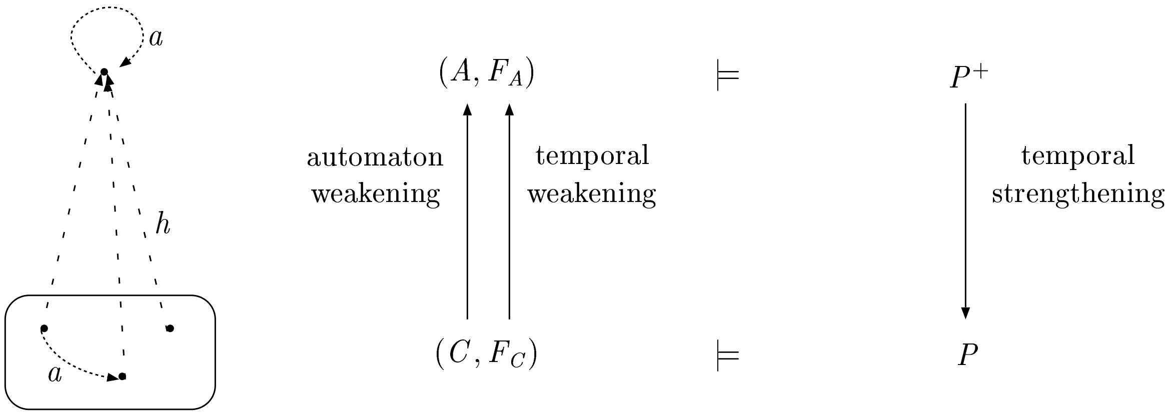

Figure 4.4: Abstraction Rule (Abs-P_2)

The abstraction theorem can easily be extended to live I/O automata.

Theorem 4.2.11 (Abstraction for Live I/O Automata)
Let (C, F_C) and (A, F_A) be live I/O automata with the same set of actions, and let h be a function from $states(C)$ to $states(A)$. Suppose $(C, F_C) \leq^L_{AF} (A, F_A)$ via h, and let P^+ be a temporal strengthening of P w.r.t. h. Then:

$$\frac{(A, F_A) \models P^+}{(C, F_C) \models P} \; (Abs\text{-}P_2)$$

Proof.
Let $\alpha \in exec(C)$ with $\alpha \models F_C$. Then we have to show that P holds for α. Because of $(C, F_C) \leq^L_{AF} (A, F_A)$ we get by Lemma 4.2.6 that A is an automaton weakening of C. Therefore, $c_h(\alpha) \in exec(A)$. By definition of $\leq^L_{AF}$ we get additionally that F_A is a temporal weakening of F_C. Therefore $c_h(\alpha) \models F_A$. Because of $(A, F_A) \models P^+$ every execution of A that satisfies F_A, satisfies P^+ as well. As $c_h(\alpha)$ fulfills both conditions, we get $c_h(\alpha) \models P^+$. Finally, as P^+ is a temporal strengthening of P, we get $\alpha \models P$. $\Box$

This theorem, which is illustrated in Fig. 4.4, permits to push inherently difficult liveness proofs to an automatic proof checker. As one might expect, this will not always be possible without further interactive intervention. In fact, in a lot of cases some of the theorem's assumptions are not fulfilled: either F_A is not a temporal weakening of F_C or $(A, F_A) \models P^+$ cannot be proven in the abstract system. This is illustrated by the following example.

Example 4.2.12
Let us try to prove the following property about $Cockpit_C$ from Example 4.2.10.

$$P_3 \equiv \Box(PonR \in stack \wedge \Diamond\Box(\forall a'.\ a \neq Alarm(a')) \Rightarrow \Diamond\neg(PonR \in stack))$$

"If PonR is stored and only finitely many further alarms arrive,
then sometimes PonR will be removed from the stack."

Note first, that P_3 holds only if the pilot treats the acknowledgments in a fair way. This is necessary, even if all other actions are excluded, as done by the assumption $\Diamond\Box(\forall a'.\ a \neq Alarm(a'))$. Otherwise there would be finite executions which do not satisfy the property, as the aforementioned assumption in the finite case merely means that the last action has to be an *Ack* action. Therefore, we have to ensure the infinity of all executions, hereby excluding "stuttering" which is allowed at the end of finite executions. Therefore, we define

$$F_C \quad \equiv \quad WF_C\,(\bigcup_{a \in Alarms} Ack(a))$$

and try to prove $(Cockpit_C, F_C) \models P_3$ with the same abstraction h_1 as in Example 4.2.10. First, we define the corresponding abstract properties for P_3 and F_C:

$$\begin{array}{lcl} P_3^+ & \equiv & \Box(PonR\text{-}in \wedge \Diamond\Box(\forall a'.\ a \neq Alarm(a')) \Rightarrow \Diamond\neg PonR\text{-}in) \\ F_A & \equiv & WF_A\,(\bigcup_{a \in Alarms} Ack(a)) \end{array}$$

In fact, P_3^+ is a temporal strengthening of P_3. However, we cannot apply the liveness abstraction rule (Abs-P_2) because of the following reasons:

- $(Cockpit_A, F_A) \models P_3^+$ is not satisfied.
  This is due to the loop in $Cockpit_A$ which allows to acknowledge alarms $a \neq PonR$ infinitely often without ever leaving the state $PonR\text{-}in$ (see Fig. 4.3).

- F_A is not a temporal weakening of F_C.
  To explain this we use the structural abstraction rules of Lemma 4.2.4. Then it remains to show that

  $$Enabled_A\ (\bigcup_{a \in Alarms} Ack(a))\ (h_1\ stack) \Rightarrow Enabled_C\ (\bigcup_{a \in Alarms} Ack(a))\ stack$$

  This is, however, not satisfied, as $Ack(a), a \neq PonR$ is enabled for $\neg PonR\text{-}in$ in $Cockpit_A$ but not for $stack = [\,]$ in $Cockpit_C$, although $h_1([\,]) = false$. $\Box$

The example showed the two main problems when abstracting liveness properties. In the following we will summarize them and propose adequate solutions for each of them. See also Fig. 4.5 and Fig. 4.6 which demonstrate the two problems (indicated by big arrows) and their solutions (indicated by dotted, big arrows).

- First, the abstract liveness assumption F_A of A may be too weak because of loops which have been introduced in A, but did not exist in C. Then $(A, F_A) \models P^+$ cannot be proved. The solution is to *strengthen the abstract liveness assumption* by deducing further temporal properties H_1 from C whose weakening abstractions H_1^- can be used as a further assumption for A.

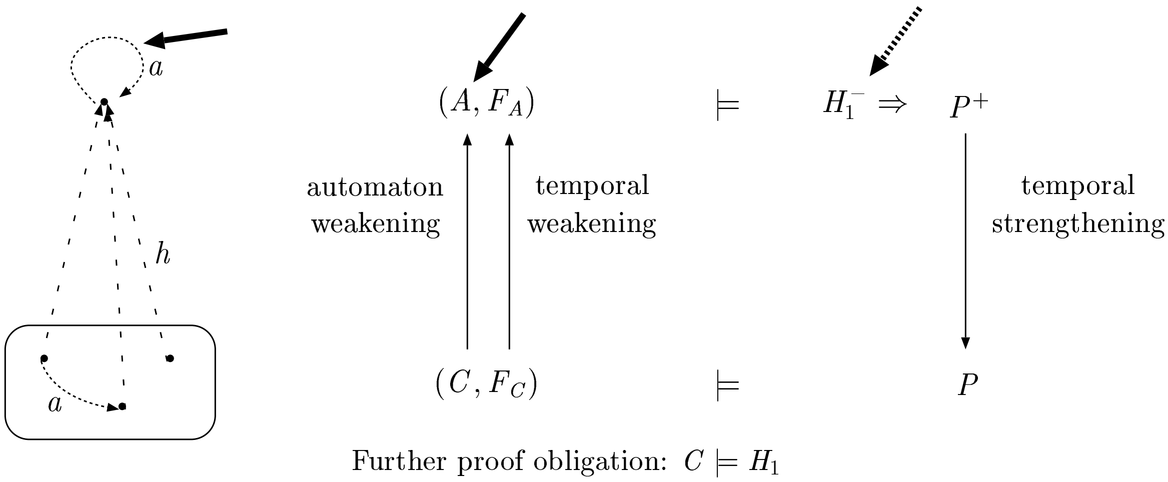

Figure 4.5: Motivation for H_1 in Abstraction Rule (Abs-P_3)

- Second, the concrete liveness assumption F_C of C may be too strong, which means that the corresponding abstract F_A is not a temporal weakening of F_C. This is often the case if F_C and F_A are fairness properties, as weakening of fairness means at the same time to *strengthen* the enabledness of a set of actions Λ and to *weaken* the occurrence of Λ. Therefore, there is not much elbowroom for a reasonable abstraction. The solution is to *weaken the concrete liveness assumption* in order to get a form H_2 which is better amenable to weakening.

Both solutions are handled jointly by the following rule.

Theorem 4.2.13 (Improving Abstractions)
Let (C, F_C) and (A, F_A) be live I/O automata with the same set of actions, and let h be a function from $states(C)$ to $states(A)$. Suppose $(C, H_2) \leq^L_{AF} (A, H_2^-)$ via h, and let P^+ be a temporal strengthening of P w.r.t. h and H_1^- be a temporal weakening of H_1. Then:

$$\frac{(A, H_2^-) \models H_1^- \Rightarrow P^+ \qquad (C, F_C) \models H_1 \wedge H_2}{(C, F_C) \models P} \; (Abs\text{-}P_3)$$

Proof.
Let $\alpha \in exec(C)$ with $\alpha \models F_C \wedge H_1 \wedge H_2$. Then we have to show that P satisfies α. Because of $(C, H_2) \leq^L_{AF} (A, H_2^-)$ we get by Lemma 4.2.6 that A is a weaker automaton than C. Therefore, $c_h(\alpha) \in exec(A)$. By definition of $\leq^L_{AF}$ we get additionally that H_2 is a temporal weakening of H_2^-. Moreover, we use that H_1^- is a temporal weakening of H_1. Therefore $c_h(\alpha) \models H_1^- \wedge H_2^-$. Because of $(A, H_2^-) \models H_1^- \Rightarrow P^+$ every execution of A that satisfies $H_1^- \wedge H_2^-$, satisfies P^+ as well. As $c_h(\alpha)$ fulfills the two formulae, we get $c_h(\alpha) \models P^+$. Finally, as P^+ is a temporal strengthening of P, we get $\alpha \models P$. $\square$

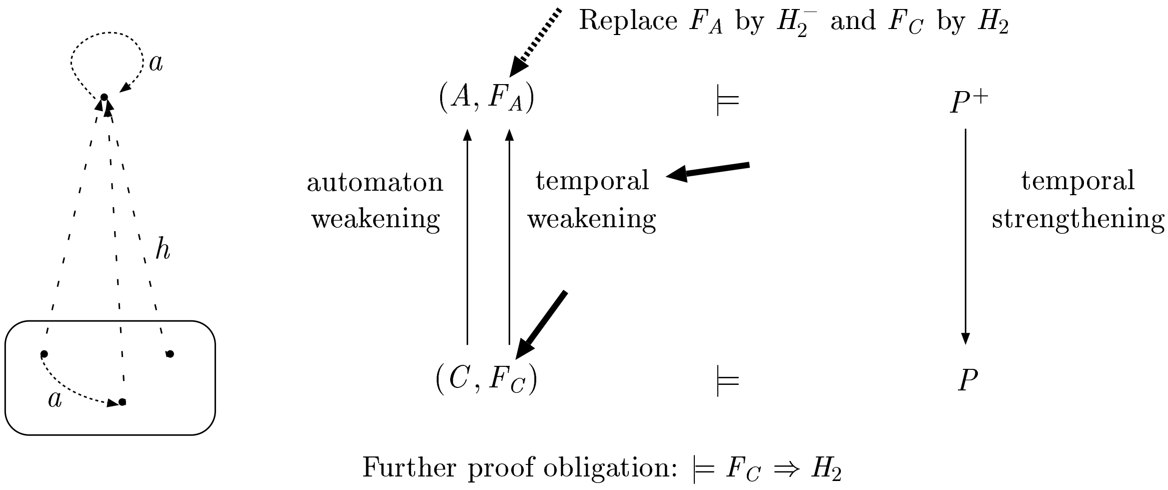

Figure 4.6: Motivation for H_2 in Abstraction Rule (Abs-P_3)

Note that for the solution to the first problem the assumption $(C, F_C) \models H_1 \wedge H_2$ is needed to get $C \models H_1$ in order to yield a stronger abstract H_1^-, whereas for the solution to the second problem it is needed to express $\models F_C \Rightarrow H_2$ in order to yield a weaker concrete H_2.

The careful reader may have noticed that the rule is similar to Theorem 4.2.11, but by using auxiliary formulas H_1 and H_2 it in addition allows to use only temporal weakenings and strengthenings of naturally *corresponding* properties instead of arbitrary ones.

Example 4.2.14
With the extended rule (Abs-P_3) we are now able to prove $(Cockpit_C, F_C) \models P_3$ from Example 4.2.12 with the abstraction function h_1 from Example 4.2.10. First, we define

$$H_1 \quad \equiv \quad \Box\Diamond(\exists a'.\ a = Ack(a')) \Rightarrow \Box\Diamond(\exists a'.\ a = Alarm(a'))$$

which falsifies the assumption $\Diamond\Box(\forall a'.\ a \neq Alarm(a'))$ of P_3 and therefore makes P_3 true. Intuitively, the property H_1 guarantees that there cannot be only *Ack* actions from some time on, as there have to be infinitely many *Alarm* actions as well. This property is an instance of the more general *well-founded properties* which can be used to tackle loops appearing in abstractions (see [Mer97]). The general idea is to define an order on the transitions and then derive a property of the shape: if transitions decrease infinitely often, then they also have to increase infinitely often. This allows to generate contradictions to loops, which are introduced in the abstract model and make liveness properties fail. Returning to our example, it is obvious that $H_1^- \equiv H_1$ is a temporal weakening of H_1 as it does not reference states. Furthermore, $Cockpit_C \models H_1$ is easily established deductively. Therefore, we may take H_1^- to strengthen the abstract liveness assumption.

Second, we define

$$\begin{aligned} H_2 \quad &\equiv \quad \Diamond\Box(PonR \in stack) \Rightarrow \Box\Diamond a \in \textstyle\bigcup_{a' \in Alarms} Ack(a') \\ H_2^- \quad &\equiv \quad \Diamond\Box PonR\text{-}in \Rightarrow \Box\Diamond a \in \textstyle\bigcup_{a' \in Alarms} Ack(a') \end{aligned}$$

where H_2 is weaker than the former fairness property, i.e. $F_C \Rightarrow H_2$. Furthermore, H_2^- is in fact a temporal weakening of H_2 now, as the enabledness of *PonR-in* in A implies that of $(PonR \in stack)$ in C. Notice that this weakened fairness H_2^- is still strong enough to show the desired property P_3^+ as we only need fair treatment of *Ack* actions, if a *PonR* alarm is stored.

Therefore, the extended abstraction rule $(Abs\text{-}P_3)$ allows us to reduce $(Cockpit_C, F_C) \models P_3$ to $(Cockpit_A, H_2^-) \models H_1^- \Rightarrow P_3^+$, which is indeed satisfied and can be proved by the STeP model checker, for example. This finishes the abstraction proof.

Let us consider an alternative solution to the introduction of H_2. Instead of weakening the fairness F_C one could also employ a more complicated abstraction function h_2 which ensures that F_A is indeed a temporal weakening of F_C. Therefore, define

$$\begin{array}{lcl} h_2 & : & (\alpha)list \rightarrow (bool \times bool) \\ h_2(stack) & \equiv & (PonR \in stack, \exists a \neq PonR.\ a \in stack) \end{array}$$

with the following corresponding abstract automaton $Cockpit'_A$, which is augmented w.r.t. $Cockpit_A$ by a boolean variable *Other-in*, initially false, indicating whether an alarm $a \neq PonR$ is stored:

Transitions of $Cockpit'_A$:

input $Alarm(a)$, $a \in Alarms$	**output** $Ack(a)$, $a \in Alarms$
post: **if** $a = PonR$	**pre**: **if** $a = PonR$
then $PonR\text{-}in := true$	**then** $PonR\text{-}in$
else $Other\text{-}in := true$	**else** $Other\text{-}in$
	post: **if** $a = PonR$
	then $PonR\text{-}in := false$
	else $Other\text{-}in := false$
	or $Other\text{-}in := Other\text{-}in$

This abstraction function h_2 is discerning enough to transfer the enabledness of *Ack* actions from $Cockpit'_A$ to $Cockpit_C$, as it respects the case when the stack is empty. Therefore, F_A is a temporal weakening of F_C, which makes it sufficient to use only the aforementioned H_1 to apply $Abs\text{-}P_3$, whereas $H_2 = F_C$ and $H_2^- = F_A$. □

This example shows that the abstraction rule $(Abs\text{-}P_3)$ can be reagrded as proof support for the following methodological decision every proof designer must take if abstraction fails: either one chooses a more complicated abstraction function — togehter with a new associated abstract model — or one stays with a simpler function and has most likely to improve the behaviour of one of the models instead. According to our proof rule this improvement often simply means to presume a further assumption about the environment. In our experience it requires less intuition to rule out such undesired behaviour of the environment than to invent a new abstraction function. Therefore, we propose the following methodology, that allows to generate appropriate abstractions incrementally:

- Start with a primitive abstraction function for a safety property preferably generated by an heuristics.
- If the model checker fails, the reason may be either that the desired property is refuted already or that the abstraction is not sophisticated enough. This question should be answered by testing the generated counterexample w.r.t. the original automaton. If the abstraction is not sophisticated enough, the necessary improvement of the abstraction can be accomplished by the extended abstraction rule (Abs-P_3). This results in an incremental process that finally reaches an appropiate abstraction.
- For further safety or even liveness properties, the same abstraction function should be reused. Once more, rule (Abs-P_3) allows to improve the behaviour of the abstraction incrementally.

This methodology enables the reuse of simple abstraction functions even for the liveness part and thus decreases the required intuition.

4.2.3 Abstraction Rules for Automata

In classical I/O automata theory properties about automata are expressed by automata as well. This section will demonstrate how abstraction rules can be proved for this setting, which are rather analogous to the rules for properties specified in temporal logic.

For safe I/O automata the basic rule looks as follows:

Theorem 4.2.15 (Abstraction for Safe I/O Automata)
Let C, A, P^+, and P be safe I/O automata. Suppose that C and A have the same actions, and P^+ and P have the same actions, whereas A and P^+ have the same external actions. Assume that $C \leq_{AF} A$ and $P^+ \leq_{AF} P$. Then:

$$\frac{A \preceq_S P^+}{C \preceq_S P}\ (Abs\text{-}A_1)$$

Proof.
As every abstraction function represents a particular refinement mapping, we get $C \leq_R A$ and $P^+ \leq_R P$. The correctness result for refinement mappings yields $C \preceq_S A$ and $P^+ \preceq_S P$. Then, the result follows immediately from the transitivity of $\preceq_S$. □

Often the rule is used in a simpler fashion with $P^+ = P$. Then, only the automaton C is weakened, but the "property automaton" P is not strengthened.

Example 4.2.16
Consider once more the alarm system $Cockpit_C$ and its abstract counterpart $Cockpit_A$ under the abstraction h_1 of Example 4.2.10. We enhance the two automata by an output action *Info* which signals the presence of newly arrived alarms to the pilot.

Extension to $Cockpit_C$	Extension to $Cockpit_A$
output $Info(a), a \in Alarms$	**output** $Info(a), a \in Alarms$
pre: $stack = a : stack'$	**pre**: $a = PonR \Rightarrow PonR\text{-}in$

It is easily shown that $Cockpit_C \leq_{AF} Cockpit_A$ w.r.t. h_1 still holds with the additional *Info* action.

This abstraction can be used to prove property P_4, expressed by the following I/O automaton: the state is the same as for $Cockpit_A$, i.e. a boolean variable $PonR\text{-}in$, which is initially false. The actions are:

Transitions of P_4:

input $Alarm(a), a \in Alarms$	**output** $Info(a), a \in Alarms$
post: **if** $a = PonR$ **then** $PonR\text{-}in := true$	**pre**: $a = PonR \Rightarrow PonR\text{-}in$

The I/O automaton P_4 expresses that every $PonR$ alarm is not signaled to the pilot before it has arrived, i.e. *Alam* and *Info* actions appear always in the desired order. Note that P_4 cannot be expressed in TLS, as TLS does not feature an *unless* operator in analogy to TLA.

Using the abstraction rule (Abs-A_1) in the simpler fashion with $P = P^+$ we may conclude the desired refinement $Cockpit_C \preceq_S P_4$ from the simpler $Cockpit_A \preceq_S P_4$, which is easily established by a model checker like μcke (see §4.4.3). □

In the sequel we present extensions of this rule, which look rather analogous to the rules for temporal properties, although the proofs are different.

Theorem 4.2.17 (Abstraction for Live I/O Automta)
Let (C, F_C), (A, F_A), (P^+, F_{P^+}), and (P, F_P) be live I/O automata. Suppose that C and A have the same actions, and P^+ and P have the same actions, whereas A and P^+ have only the same external actions. Assume that $(C, F_C) \leq^L_{AF} (A, F_A)$ and $(P^+, F_{P^+}) \leq^L_{AF} (P, F_P)$. Then:

$$\frac{(A, F_A) \preceq_L (P^+, F_{P^+})}{(C, F_C) \preceq_L (P, F_P)} \; (Abs\text{-}A_2)$$

Proof.
By Theorem 4.2.8 we get $(C, F_C) \preceq_L (A, F_A)$ and $(P^+, F_{P^+}) \preceq_L (P, F_P)$ from the assumptions $(C, F_C) \leq^L_{AF} (A, F_A)$ and $(P^+, F_{P^+}) \leq^L_{AF} (P, F_P)$. Then, the result follows immediately from the transitivity of $\preceq_L$. □

Theorem 4.2.18 (Improving Abstractions)
Let (C, F_C), (A, F_A), (P^+, F_{P^+}), and (P, F_P) be live I/O automata. Suppose that C and A have the same actions, and P^+ and P have the same actions, whereas A and P^+ have only the same external actions. Assume that $(C, H_2) \leq^L_{AF} (A, H_2^-)$ and $(P^+, F_{P^+}) \leq^L_{AF} (P, F_P)$. Let H_1^- be a temporal weakening of H_1. Then:

$$\frac{(A, H_1^- \wedge H_2^-) \preceq_L (P^+, F_{P^+}) \qquad (C, F_C) \models H_1 \wedge H_2}{(C, F_C) \preceq_L (P, F_P)} \; (Abs\text{-}A_3)$$

Proof.
As the H_i^- are temporal weakenings of H_i for $i \in \{1, 2\}$, we get $(A, F_A) \models H_1^- \wedge H_2^-$. Therefore, $A \models F_A \Rightarrow H_1^- \wedge H_2^-$, which implies $(A, F_A) \preceq_L (A, H_1^- \wedge H_2^-)$. Now we apply Theorem 4.2.17 with the assumption $(A, H_1^- \wedge H_2^-) \preceq_L (P^+, F_{P^+})$ and get $(C, H_1 \wedge H_2) \preceq_L (P, F_P)$. As $(C, F_C) \models H_1 \wedge H_2$, we get $A \models F_C \Rightarrow H_1 \wedge H_2$ and therefore $(C, F_C) \preceq_L (C, H_1 \wedge H_2)$. The rest follows easily from the transitivity of $\preceq_L$. $\Box$

4.3 Extension to Relational Abstractions

It has been argued in [LGS+95, GL93] that relations can provide more powerful abstractions, as they are more general than functions. As we shall see in the following, it is straightforward to extend our abstraction theory to relations. We will sketch only the necessary changes. In addition, we introduce the notion of *internal implementation* and show that relational abstractions are complete w.r.t. this notion.

Definition 4.3.1 (Relational Correspondence)
Let C and A be two I/O automata with the same set of actions. Let $\alpha = s_0 a_1 s_1 \ldots$ be an execution scheme over $states(C)$ and $acts(C)$ and $\alpha' = t_0 b_1 t_1 \ldots$ be an execution scheme over $states(A)$ and $acts(A)$. Furthermore, let R be a total relation between $states(C)$ and $states(A)$. Then α and α' are said to *relationally correspond* w.r.t. R , if the index mapping m from α to α' w.r.t. R is the identity, or explicitly, if $t_i \in R[s_i]$ and $a_i = b_i$ for all i. $\Box$

Note that for every execution scheme α and total relation R there is always at least one relationally corresponding execution scheme α'.

Global weakenings and strengthenings are defined as before, except that the desired implication is not required for $c_h(\alpha)$, but for all relationally corresponding α'. Similarly, local weakenings and strengthenings are defined as before, except that $Q(s', a, t')$ has to hold for all $s' \in R[s]$ and $t' \in R[s]$ instead for $s' = h(s)$ and $t' = h(t)$ only. Note that it would not suffice to consider only an exemplary relationally corresponding α' or some $s' \in R[s], t' \in R[t]$, respectively, as otherwise the definition of weakenings and strengthenings would not be dual to one another. This, however, is necessary, as otherwise Lemma 4.2.4 would not hold anymore in this context, which allows to reduce global weakenings/strengthenings to local ones.

Below, we introduce forward and backward abstraction relations, which are specific forward and backward simulations, respectively.

Definition 4.3.2 (Abstraction Relations)
Let C and A be safe I/O automata with the same set of actions and R be a total relation between $states(C)$ and $states(A)$. Then, R is said to be an *forward abstraction relation*, iff it satisfies the following conditions:

- If $s \in start(C)$ then $R[s] \cap start(A) \neq \emptyset$.
- If s is a reachable state of C, $s \xrightarrow{a}_C t$, and $s' \in R[s]$, then $\exists t' \in R[t].\ s' \xrightarrow{a}_A t'$.

We write $C \leq_{FAR} A$, if there is a forward abstraction relation from C to A. Furthermore, R is said to be an *backward abstraction relation*, iff it satisfies the following conditions:

- If $s \in start(C)$ then $R[s] \subseteq start(A)$.
- If s is a reachable state of C, $s \xrightarrow{a}_C t$, and $t' \in R[t]$, then $\exists s' \in R[s].\ s' \xrightarrow{a}_A t'$.

We write $C \leq_{BAR} A$, if there is a backward abstraction relation from C to A. □

As before, abstraction relations R between $states(C)$ and $states(A)$ induce an automaton weakening between C and A.

Definition 4.3.3 (Live Abstraction Relations)
Let (C, F_C) and (A, F_A) be live I/O automata with the same set of actions. Then,

- $(C, F_C) \leq^L_{FAR} (A, F_A)$ iff $C \leq_{FAR} A$ via h and F_A is a global weakening of F_C w.r.t. h for some h.
- $(C, F_C) \leq^L_{BAR} (A, F_A)$ iff $C \leq_{BAR} A$ via h and F_A is a global weakening of F_C w.r.t. h for some h. □

Theorem 4.3.4 (Soundness of Live Abstraction Relations)
Let (C, F_C) and (A, F_A) be live I/O automata with the same set of actions. Then,

- if $(C, F_C) \leq^L_{FAR} (A, F_A)$, then $(C, F_C) \preceq_L (A, F_A)$.
- if $(C, F_C) \leq^L_{BAR} (A, F_A)$, then $(C, F_C) \preceq_L (A, F_A)$.

Proof.
Analogous to the proof of Theorem 4.2.8. □

Using this modified infrastructure, the six abstraction rules for abstraction functions directly carry over to relations. Simply replace everywhere $\leq_{AF}$ by $\leq_{FAR}$ or $\leq_{BAR}$.

In the sequel we present a restricted kind of implementation notion, with respect to which abstraction relations are complete. The implementation notion differs from the standard one by demanding schedule inclusion instead of trace inclusion, where a schedule means the action subsequence of an execution.

Definition 4.3.5 (Internal Implementation)
Let A be any safe I/O automaton. Define $\mathcal{E}(A)$ to be A except for turning its internal actions into output actions, i.e. $out(\mathcal{E}(A)) = out(A) \cup int(A)$, $int(\mathcal{E}(A)) = \{\}$, and $X(\mathcal{E}(A)) = X(A)$ for every other component $X \in \{in, states, start, steps\}$ of A.

Let (C, F_C) and (A, F_A) be live I/O automata with the same set of actions, whose liveness conditions are induced by the temporal formulas F_C and F_A. Then, define

$$(C, F_C) \preceq_A (A, F_A) \;\;\equiv\;\; (\mathcal{E}(C), F_C) \preceq_L (\mathcal{E}(A), F_A)$$

For the special case of safe I/O automata C and A the definition means:

$$C \preceq_A A \;\;\equiv\;\; \mathcal{E}(C) \preceq_S \mathcal{E}(A)$$

If $(C, F_C) \preceq_A (A, F_A)$, we say that (C, F_C) internally implements (A, F_A). □

Theorem 4.3.6 (Completeness of Abstraction Relations)
Let C and A be safe I/O automata with the same set of actions, and let C have finite invisible nondeterminism. Then,

$$(\exists B.\ C \leq_{FAR} B \leq_{BAR} A) \;\;\Leftrightarrow\;\; C \preceq_A A$$

Proof.

"$\Rightarrow$": Assume $C \leq_{FAR} B \leq_{BAR} A$ via R and S. As every abstraction relation is a specific forward or backward simulation, the soundness of simulations (Theorem 2.4.6) gives $C \preceq_S A$. Therefore, the result $C \preceq_A A$ follows together with the fact, that for all α, α' with $(\alpha, \alpha') \in R$ and for all β, β' with $(\beta, \beta') \in S$ all actions, even the internal ones, coincide.

"$\Leftarrow$": As $C \preceq_A A = \mathcal{E}(C) \preceq_S \mathcal{E}(A)$, the completeness result for simulations (Theorem 2.4.7) ensures the existence of an I/O automaton B with $\mathcal{E}(C) \leq_F B \leq_{iB} \mathcal{E}(A)$. Therefore, there is a backward simulation R_1 between B and $\mathcal{E}(A)$. As $\mathcal{E}(A)$ has only external actions, every move (s, γ, t) of it degenerates to a step (s, a, t) for some action a of $\mathcal{E}(A)$. Therefore, the backward simulation R_1 degenerates to a backward abstraction relation. The same applies to the forward simulation R_2 which exists between B and $\mathcal{E}(C)$. It degenerates ot a forward abstraction relation. Therefore, $C \leq_{FAR} B \leq_{BAR} A$ via R_1 and R_2, as desired. □

4.4 Model Checking

The abstraction rules of the previous section generate proof obligations, which should be discharged by a model checker. This is in principle possible if the involved I/O automata are finite state and feature only finitely many actions. The challenge is to make the interface between these proof obligations and the model checker efficient and practically applicable. In §4.4.1 we identify criteria for this aim and choose adequate model checkers and translations. This choice depends on whether properties are expressed as temporal formulas or as automata, as this determines whether the generated proof obligations have the form $(A, F_A) \models P$ (Section 4.2.2) or $(A, F_A) \preceq_L (P, F_P)$ (Section 4.2.3). In the subsequent sections §4.4.2 and §4.4.3 for each type of obligation tailored translations are given, which are demonstrated by the abstract cockpit alarm system.

4.4.1 General Discussion

It is a well known result that every ω-automaton can be translated to a linear temporal formula and vice versa [VW86]. This could be adapted to turn both types of proof obligations into a linear temporal formula, which could then be evaluated by a tautology checker for propositional temporal logic. However, this approach would not satisfy the following two important criteria for a reasonable model checker interface:

- **Efficiency:** As the crucial restriction of model checkers is their scalability, the translation should not add further unnecessary complexity in terms of the input size.
- **Structure Preservation:** An important feature of model checkers is their ability to generate counterexamples. Therefore, the system structure should be preserved during the translation in such a way that generated counterexamples can be reinterpreted in the original specification. In addition, structure preservation increases confidence into the soundness of the actual compilers. Last but not least, loss of structure is usually interlinked with loss of efficiency. For example, variable orderings, which are crucial for the success of BDDs, depend often on the structure of the specification[3].

[3]For example in μcke, in the case of transition predicates $P(s, a, t)$ the bit representations of the states s and t should be interleaved, as they have the same type, but the action a should be handled separately. Otherwise, even small transition predicates blow up tremendously.

These considerations suggest to handle both types of proof obligations in a separate and tailored way.

Checking $(A, F_A) \models P$. As TLS is a linear time temporal logic, the interface criteria dissuade from employing a CTL model checker like SMV [CGL94], although it is in principle possible to encode an LTL formula into a CTL formula plus some additional fairness constraints [CGH94]. It seems preferable to select an LTL model checker like Spin [Hol91] or STeP [MP95, BBC$^+$96]. Both tools do not handle actions explicitly. Therefore, there is no distinction between internal, output, and input actions, and temporal formulas express only properties about states. Spin, however, has some further drawbacks. First, it does not support the next-time operator in verification, as this would encumber partial order techniques which are used to attack state explosion [GW94, HP94]. Furthermore, specifications in the guarded-command like language Promela do not match the precondition/effect style of I/O automata immediately.

Therefore, in §4.4.2 we take the STeP model checker and rectify the deficiencies concerning non-explicit actions.

Checking $(A, F_A) \preceq_L (P, F_P)$. For the restricted version of fair trace inclusion this problem has already been adequately solved by a translation of fair I/O automata to ω-automata [KMOS95]. In that paper it is shown that fair trace inclusion of I/O automata coincides with language containment of ω-automata being generated by the translation. The method considers all particularities of I/O automata (interleaving, internal actions, finite executions), and the proof obligation is treated by the COSPAN model checker [HR87, Kur94]. COSPAN checks language containment of A and P by checking language emptiness of the intersection of A and the complement of P. Such an algorithm could alternatively be encoded in a sufficiently expressive logic. For Büchi automata for example, language containment can be encoded into Park's μ-calculus [BCM$^+$92] or second order monadic logic [Tho90][4]. The interface criteria, however, do not suggest to adapt those translations to I/O automata. In §4.4.3 we rather propose a solution which completely preserves the structure of the proof obligations. The idea is to encode forward simulations directly into the μ-calculus [Par76]. A further, practical advantage of this approach is the fact that there is already a tactic in Isabelle which invokes the μ-calculus model checker μcke [Bie97a, Bie97b] as an external oracle. However, this approach only yields a complete decision procedure w.r.t. trace inclusion of two I/O automata A and P if P is deterministic and both A and P are safe. At least the first restriction is not that restrictive in practice, because in the context of abstraction (as opposed to a context of layered implementations), P represents a property about A which is usually deterministic[5].

[4]Park's μ-calculus and second order monadic logic have exactly the same expressivity [Eme90].

[5]Note that in this context A will most likely be nondeterministic as it represents the abstraction of some automaton C, but P merely describes a specific property about the external actions of A. Therefore, it is often deterministic.

4.4.2 Checking Temporal Formulas with STeP

Below we give a translation of I/O automata into transition systems [MP95], which represent the computational model of the STeP model checker [BBC$^+$96, MBB$^+$97]. Furthermore, we translate TLS formulas into Manna/Pnueli's LTL [MP95]. Then, the correctness and completeness of this translation is investigated, and finally, we demonstrate how it is adapted to the concrete STeP model checker, using the running example of the cockpit alarm system. Concerning an introduction to the notations of LTL and transition systems the reader is refered to [BBC$^+$96, MBB$^+$97].

The idea of the translation is to encode the explicit actions of I/O automata into the state space and to add a further idling action. Liveness conditions are not coded into fairness sets, as this would not be general enough[6]. Instead it will be considered as a further assumption of the temporal property to be proved.

Translating I/O Automata. Let (A, F_A) be a live I/O automaton. We assume that states of A are given as a mapping σ from a set of variable names $\mathcal{V}$ to a set of values Val. Furthermore, let the start states $start(A)$ be characterized by a predicate $P^{st}(\mathcal{V})$, and every transition (s, a, t) in $steps(A)$ by a predicate $P^{tr}_a(\mathcal{V}, \mathcal{V}')$, which may refer to both primed and unprimed versions of the state variables. It is obvious how the predicates P^{st} and P^{tr} for an I/O automaton A are obtained from its components, if A is defined by means of the operators $\|$, $hide_\Lambda$ or $rename_\sigma$.

Then we define $\Psi(A) = (\mathcal{V}_{ext}, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$, where

- $\mathcal{V}_{ext} = \mathcal{V} \uplus \{Act\}$ with Act ranging over $acts(A) \uplus \{idle\}$,
- $\Theta(\mathcal{V}) = P^{st}(\mathcal{V})$,
- $\mathcal{T} = \{\tau_a \mid a \in acts(A)\} \uplus \{\tau_{idle}\}$, where every τ_a, $a \in acts(A) \uplus \{idle\}$ is characterized by $\sigma_{\tau_a}(\mathcal{V}_{ext}, \mathcal{V}'_{ext}) = P^{tr}_a(\mathcal{V}, \mathcal{V}') \wedge Act' = \tau_a$, and $\sigma_{\tau_{idle}}(\mathcal{V}_{ext}, \mathcal{V}'_{ext}) = \forall v.\ v = v'$,
- $\mathcal{J} = \emptyset$ and $\mathcal{C} = \emptyset$.

Translating Temporal Formulas. Let P be any TLS formula over $\mathcal{V}_{ext}$ and $\mathcal{V}'_{ext}$, where actions a appear in transition predicates only in the form $a \in \Lambda$ for action sets Λ[7]. Then $\Psi(P)$ has to transform every transition predicate $p(s, a, s')$ occurring in P into a state predicate $p^*(s)$. This is done by replacing every subformula $a \in \Lambda$ by $\bigcirc \exists a' \in \Lambda.\ Act = a'$, and by replacing every transition predicate $p(s, a, s')$ involving primed variables s' by $s = c \Rightarrow \bigcirc \Psi(p(c, a, s))$, where c is a constant (which is stable over time).

[6] Besides, the two fairness notions do not coincide exactly as I/O automata require fair turns to some element of a fairness set, whereas transition systems require them to every element.

[7] These are the only interesting properties about actions in practice.

Theorem 4.4.1 (Correctness and Completeness)
Let (A, F_A) be a live I/O automaton and P an TLS formula. Suppose that actions a appear in transition predicates of F_A and P only in the form $a \in \Lambda$. Denote by $\models_{MP}$ the notion of A-validity by Manna and Pnueli. Then,

$$(A, F_A) \models P \qquad \text{iff} \qquad \Psi(A) \models_{MP} \Psi(F_A \Rightarrow P)$$

Proof.
The translation Ψ is extended to executions by defining $\Psi := \nabla' \circ \zeta$ where $\zeta(s_0 a_1 s_1 \ldots) := s_0\ ([Act \mapsto a_1] \cup s_1) \ldots$ and ∇' denotes the operator defined in §11.2 which adds infinite stuttering at the end of every finite execution. With this definition we first get

$$\alpha \models P \quad = \quad \Psi(\alpha) \models_{MP} \Psi(P) \quad (1)$$

which follows easily from Lemma 3.2.7 and the special encoding of temporal formulas, namely that action predicates (resp. primed variables s') refer to the Act variable (resp. the variable s) in the successor state.

"$\Rightarrow$": Suppose $\sigma \in Comp(\Psi(A))$ with $\sigma \models_{MP} \Psi(F_A)$, where $Comp(T)$ denotes the set of behaviours of a transition system T. Then there is an execution α with $\sigma = \Psi(\alpha)$ and $\alpha \in execs(A)$ because of the specific encoding of I/O automata: the variable Act always reflects the action performed by A in the previous step. Therefore, with (1) we get $\alpha \models F_A$. The assumption $(A, F_A) \models P$ then implies $\alpha \models P$, which gives the desired $\sigma \models_{MP} \Psi(P)$ by applying (1) again.

"$\Leftarrow$": Suppose $\alpha \models F_A$ and $\alpha \in execs(A)$. Then we get $\Psi(\alpha) \models_{MP} \Psi(F_A)$ with (1). Furthermore, we get $\Psi(\alpha) \in Comp(\Psi(A))$, again because of the specific encoding of actions and primed variables. Therefore, the assumption $\Psi(A) \models_{MP} \Psi(F_A \Rightarrow P)$ implies $\Psi(\alpha) \models_{MP} \Psi(P)$, which gives with (1) the desired $\alpha \models P$. $\square$

Example 4.4.2
The Examples 4.2.10 and 4.2.14 of the cockpit alarm system generated the proof obligations $Cockpit_A \models P_i^+, i \in \{1, 2\}$ and $(Cockpit_A, H_2^-) \models H_1^- \Rightarrow P_3^+$, respectively. The I/O automaton $Cockpit_A$ can easily be encoded into a STeP transition system.

```
Transition System

type actions = {AlarmP, AlarmNP, AckP, AckNP}
local PonRin: bool
local Act: actions
Initially PonRin = false
Transition AlarmP NoFairness:
  enable true
  assign PonRin := true, Act := AlarmP
Transition AlarmNP NoFairness:
```

```
  enable true
  assign PonRin := PonRin, Act := AlarmNP
Transition AckP NoFairness:
  enable PonRin
  assign PonRin := false, Act := AckP
Transition AckNP NoFairness:
  enable true
  assign PonRin := PonRin, Act := AckNP
```

The variable Act encodes the actions into the state. An idling transition is automatically added by STeP. The following properties represent $\Psi(P_1^+)$, $\Psi(P_2^+)$, and $\Psi(H_1^- \wedge H_2^- \Rightarrow P_3^+)$, respectively.

```
SPEC

PROPERTY P1:  []()Act=AlarmP --> ()PonRin
PROPERTY P2:  []()Act!=AlarmP --> []!PonRin
PROPERTY P3:
   (([]<>()(Act=AckP \/ Act=AckNP)) --> []<>()(Act=AlarmP \/ Act=AlarmNP))
    /\ (<>[] PonRin --> []<>()(Act=AckP \/ Act=AckNP))
    --> [](PonRin /\ <>[]()(Act!=AlarmP /\ Act!=AlarmNP) --> <>!PonRin)
```

All properties are easily verified by STeP. □

4.4.3 Checking Trace Inclusion with μcke

In the sequel we give a translation of forward simulations between I/O automata into Park's μ-calculus[8], investigate the correctness and completeness of this translation, and finally demonstrate how it is adapted to the concrete tool μcke [Bie97a, Bie97b], using once more the example of the cockpit alarm system.

The translation of a safe and finite-state I/O automaton with only finitely many actions into a μ-calculus formula is given inductively. Assume that a set of states $\mathcal{S}_A$ for each I/O automaton A and a global set of actions $\mathcal{A}$ is given.

[8] For an introduction into syntax and semantics of Park's μ-calculus see [Par76, BCM$^+$92]. We assume a typed version with finite carrier sets (see [Bie97b]). Park's μ-calculus can express any property of the modal μ-calculus [Koz83].

Basic I/O Automata. Every basic I/O automaton A is characterized by the following boolean predicates.

$$
\begin{array}{llcl}
\mathit{In}_A : \mathcal{A} \to \mathit{bool} & \mathit{In}_A(a) & \equiv & a \in \mathit{in}(A) \\
\mathit{Out}_A : \mathcal{A} \to \mathit{bool} & \mathit{Out}_A(a) & \equiv & a \in \mathit{out}(A) \\
\mathit{Int}_A : \mathcal{A} \to \mathit{bool} & \mathit{Int}_A(a) & \equiv & a \in \mathit{int}(A) \\
\mathit{Start}_A : \mathcal{S}_A \to \mathit{bool} & \mathit{Start}_A(s) & \equiv & s \in \mathit{start}(A) \\
\mathit{Trans}_A : \mathcal{S}_A \times \mathcal{A} \times \mathcal{S}_A \to \mathit{bool} & \mathit{Trans}_A(s, a, t) & \equiv & s \xrightarrow{a}_A t
\end{array}
$$

For basic as well as non-basic automata these elementary predicates are used to define the following predicates, which describe the set of external actions and the set of all actions of an I/O automaton A.

$$
\mathit{Ext}_A(a) \equiv \mathit{Inp}_A(a) \vee \mathit{Out}_A(a) \qquad \mathit{Act}_A(a) \equiv \mathit{Ext}_A(a) \vee \mathit{Int}_A(a)
$$

Parallel Composition. The actions, initial states, and transition relation of a parallel composition $A = A_1 \parallel A_2$ are defined as follows, where $S_A = S_{A_1} \times S_{A_2}$:

$$
\begin{array}{lcl}
\mathit{Inp}_A, \mathit{Out}_A, \mathit{Int}_A & : & \mathcal{A} \to \mathit{Bool} \\
\mathit{Inp}_A(a) & \equiv & \mathit{Inp}_{A_1}(a) \vee \mathit{Inp}_{A_2}(a) \wedge \neg(\mathit{Out}_{A_1}(a) \vee \mathit{Out}_{A_2}(a)) \\
\mathit{Out}_A(a) & \equiv & \mathit{Out}_{A_1}(a) \vee \mathit{Out}_{A_2}(a) \\
\mathit{Int}_A(a) & \equiv & \mathit{Int}_{A_1}(a) \vee \mathit{Int}_{A_2}(a) \\
\mathit{Start}_A & : & \mathcal{S}_{A_1} \times \mathcal{S}_{A_2} \to \mathit{Bool} \\
\mathit{Start}_A((s_1, s_2)) & \equiv & \mathit{Start}_{A_1}(s_1) \wedge \mathit{Start}_{A_2}(s_2) \\
\mathit{Trans}_A & : & (\mathcal{S}_{A_1} \times \mathcal{S}_{A_2}) \times \mathcal{A} \times (\mathcal{S}_{A_1} \times \mathcal{S}_{A_2}) \to \mathit{Bool} \\
\mathit{Trans}_A((s_1, s_2), a, (t_1, t_2)) & \equiv & \textbf{if } \mathit{Act}_{A_1}(a) \textbf{ then } \mathit{Trans}_{A_1}(s_1, a, t_1) \textbf{ else } s_1 = t_1 \wedge \\
 & & \textbf{if } \mathit{Act}_{A_2}(a) \textbf{ then } \mathit{Trans}_{A_2}(s_2, a, t_2) \textbf{ else } s_2 = t_2
\end{array}
$$

Hiding. The I/O automaton $A = A_1 \setminus \Lambda$ is characterized by $S_A = S_{A_1}$, the predicates $\mathit{Start}_A \equiv \mathit{Start}_{A_1}$, $\mathit{Trans}_A \equiv \mathit{Trans}_{A_1}$, and the following action predicates, where $L(a) \equiv a \in \Lambda$ describes the set of actions to be hidden.

$$
\begin{array}{lcl}
\mathit{Inp}_A(a) & \equiv & \mathit{Inp}_{A_1}(a) \\
\mathit{Out}_A(a) & \equiv & \mathit{Out}_{A_1}(a) \wedge \neg L(a) \\
\mathit{Int}_A(a) & \equiv & \mathit{Int}_{A_1}(a) \vee L(a)
\end{array}
$$

Renaming. The I/O automaton $A = \mathit{rename}_\sigma(A_1)$ is characterized by $S_A = S_{A_1}$, $\mathit{Start}_A \equiv \mathit{Start}_{A_1}$ and the following predicates, where $\mathit{Ren}(a_1, a_2) \equiv a_2 = \sigma(a_1)$ describes the renaming function σ.

$$\begin{array}{lcl} Inp_A(a_2) & \equiv & \exists a_1.\, Ren(a_1, a_2) \wedge Inp_{A_1}(a_1) \\ Out_A(a_2) & \equiv & \exists a_1.\, Ren(a_1, a_2) \wedge Out_{A_1}(a_1) \\ Int_A(a_2) & \equiv & \exists a_1.\, Ren(a_1, a_2) \wedge Int_{A_1}(a_1) \\ Trans_A(s, a_2, t) & \equiv & \exists a_1.\, Ren(a_1, a_2) \wedge Trans_{A_1}(s, a_1, t) \end{array}$$

Forward Simulation. Given translations for two I/O automata C and A a forward simulation between them can be encoded directly. Internal steps, a finite sequence of internal steps, and a move are described by the following predicates:

$$\begin{array}{lcl} IntStep_A(s, t) & \equiv & \exists a.\, Int_A(a) \wedge Trans_A(s, a, t) \\ IntStep^*_A & \equiv & \mu P.\, \lambda s, t.\, (s = t) \vee \exists u.\, IntStep_A(s, u) \wedge P(u, t) \\ Move_A(s, a, t) & \equiv & (Int_C(a) \wedge IntStep^*_A(s, t)) \vee \\ & & \exists u_1, u_2.\, IntStep^*_A(s, u_1) \wedge Trans_A(u_1, a, u_2) \wedge IntStep^*_A(u_2, t) \end{array}$$

Then, the existence of a forward simulation can be expressed by

$$\begin{array}{lcl} isSim_{CA} & \equiv & \nu P.\, \lambda s_1, t_1.\, \forall a, s_2.\, Trans_C(s_1, a, s_2) \Rightarrow \exists t_2.\, Move_A(t_1, a, t_2) \wedge P(s_2, t_2) \\ SimExists_{CA} & \equiv & \forall a.\, Inp_C(a) \leftrightarrow Inp_A(a) \wedge Out_C(a) \leftrightarrow Out_A(a) \wedge \\ & & \forall s, t.\, Start_C(s) \wedge Start_A(t) \Rightarrow isSim_{CA}(s, t) \end{array}$$

Theorem 4.4.3 (Correctness and Completeness)
Let C and A be finite-state I/O automata with finitely many actions, and let A be deterministic. Then,

$$C \preceq_S A \qquad \text{iff} \qquad SimExists_{CA} = true$$

Proof.
First, it is obvious that the translation of I/O automata preserves the semantics, as the definitions for $\|$, $hide_\Lambda$, and $rename_\sigma$ are encoded in the μ-calculus without any modification. Second, the predicate $SimExists_{CA}$ indeed expresses the existence of a forward simulation between C and A. The only uncommon definitions are those concerning executions, but their correctness is obvious as well: the least fixpoint is used to express a finite execution of internal steps, and the greatest fixpoint is used to express the correspondence of (possibly) infinite executions. The actual forward simulation R can easily be constructed from $SimExists_{CA}$ — provided it is true — by defining $t_i \in R[s_i]$ for all s_i, t_i occuring in the corresponding executions. Therefore, one direction of the theorem holds because of the correctness of simulations (Theorem 2.4.6). The other one holds because of the partial completeness result for forward simulations w.r.t. deterministic specifications (Theorem 4.11 of [LV95]). $\Box$

Example 4.4.4
The abstraction example 4.2.16 generated the model checker proof obligation $Cockpit_A \preceq_S P_4$, where P_4 expresses the property that *Alarm* and *Info* actions always appear in the right order. As P_4 is a deterministic automaton, we may use the suggested translation to discharge the proof obligation. Actions and the automaton P_4 are encoded in μcke as follows.

```
enum Alarms {PonR, Fuel, Eng};
enum Actions {Alarm, Ack, Info};
class Action {Actions act; Alarms arg;};
class StateP4 {bool PonRin;};

bool IntP4(Action a) a.act = Ack;
bool StartP4(StateP4 s) s.PonRin = 0;
bool TransP4(StateP4 s, Action a, StateP4 t)
  (case
    a.act = Alarm :
          (if(a.arg = PonR)
             (t.PonRin = 1)
          else
             t.PonRin = s.PonRin);
    a.act = Ack : 0;
    a.act = Info :
          (a.arg = PonR -> s.PonRin = 1) &
          t.PonRin = s.PonRin;
   esac);
```

Note that the representation of I/O automata is very close to that in Isabelle (see §8.2), a translation is straight forward. Encoding $Cockpit_A$ in an analogous way with state type `StateCo`, start condition `StartCo`, and transition relation `TransCo` we can encode the desired trace inclusion as follows.

```
bool IntStepP4(StateP4 s, StateP4 t)
  exists Action a. IntP4(a) & TransP4(s,a,t);

mu bool IntStepStarP4(StateP4 s, StateP4 t)
  s = t |  (exists StateP4 u.
              IntStepP4(s, u) & IntStepStarP4(u, t));

bool MoveP4(Action a, StateP4 x, StateP4 y)
  IntP4(a) & IntStepStarP4(x, y) |
  (exists StateP4 u1. IntStepStarP4(x, u1) &
    (exists StateP4 u2. TransP4(u1, a, u2) & IntStepStarP4(u2, y)));

nu bool isSim(StateCo s1, StateP4 t1)
```

```
 forall Action a, StateCo s2.
   TransCo(s1, a, s2) ->
     (exists StateP4 t2. MoveP4(a, t1, t2) & isSim(s2, t2));

bool SimExists
   forall StateCo s, StateP4 t. StartCo(s) & StartP4(t) -> isSim(s,t);
```

□

4.5 Conclusion and Related Work

In this chapter we laid the foundation for an effective verification of I/O automata via a combination of theorem proving and model checking. We developed abstraction rules, which will be derived in Isabelle in §11. For the remaining proof obligations we presented translations to μcke and the STeP model checker.

The main characteristics of the abstraction theory is of practical importance: the interactive prover has to reason about steps only, whereas reasoning about entire system runs is left to the automatic prover.

In addition, a further rule allows to improve abstractions without modifying the chosen abstraction function. This can be done by strengthening the abstract model, if that appears to be too weak to prove the desired property, or by weakening the concrete model, if that includes elements which are not needed to prove the desired property, but hinder the intended abstraction.

Based on this further rule a methodology is proposed which allows to reuse simple abstraction functions even for sophisticated liveness properties and therefore decreases the required intuition.

In addition to the usual correctness considerations, we provided a completeness theorem for the safety part, which characterizes abstraction relations as means to treat implementations between systems featuring merely external actions.

Related Work. We first sketch the field of related abstraction theories. Closely related to this work are the abstraction rules for TLA by Merz [Mer97]. In fact, our abstraction theory can be regarded as a combination of the classical automata-theoretic framework [CGL92, Sif83, Kur87] with the ideas of [Mer97] to tackle the problems of liveness abstractions, which has then been adapted to I/O automata. From the theoretical point of view the theory by Merz is very attractive, as it is formulated in the uniform setting of temporal logic. Therefore, the rules make no distinction between system and property specifications. From a practical point of view, however, we argue that it is more adequate to treat safety aspects with explicit automata and use temporal formulas merely for liveness conditions. In particular, our theory features rules for porperties both expressed as automata or as temporal formulas. Another distinguishing feature between [Mer97] and our work is that the reduction from temporal formulas to step predicates is more complicated in our context. This is again due to the theory of I/O automata, which in comparison to TLA features a simpler basic model, but in return gets a more sophisticated refinement (resp. abstraction) concept. Recall the discussion of this topic in §3.4.

As mentioned above, the main ideas from the standard literature on abstraction [CGL92, Sif83, Kur87] are covered by our theory, but we have to deal with a different setting. For example we employ a linear-time instead of a branching-time temporal logic and work with explicit actions instead of standard Kripke structures.

Orthogonal to our theory are approaches that deal with data abstraction and data independence [SLW95, Wol86]. They would correspond to a notion of action refinement, just like our abstraction theory corresponds to state refinement. Up to our knowledge, however, such notions do not exist for I/O automata. Further work should investigate how interface refinement available for other formalisms, e.g. for FOCUS [BDD$^+$93], carries over to I/O automata.

Further approaches aim at transfering the ideas of abstract interpretation to abstraction [DGG97, Kel95]. They propose specific abstraction relations which are remarkably powerful, but usually require more intuition from the proof designer. Until now it is an open question whether such abstractions can easily be found in practice.

Abstraction theories brought up the idea to combine theorem proving and model checking. At first this has been done for case studies where the interactive and automatic provers were not actually integrated [Hun93, KL93, MN95]. Shortly after, several theorem provers have been combined with model checkers as external decision procedures [ORR$^+$96, MBB$^+$97]. In contrast to our framework for I/O automata, however, usuall y only simple abstraction theories are supported.

Furthermore, some nontrivial case studies have been performed concerning the verification of abstractions in theorem provers [DF95, HS96]. Interestingly, the experiences from these studies are quite different. In [HS96] the soundness proof of the abstraction required almost all invariants that would have been necessary for the classical refinement proof without model checker as well. In [DF95], however, a procedure similar to our methodology has been employed, though only imformally justified and not tailored for liveness. The result was encouraging: the interactive part could considerably be reduced.

Part II: Logical Foundations and Methodology

Chapter 5

A Methodology for HOL and HOLCF

In this chapter we first describe our presentation style for theories and proofs, that have been performed in Isabelle. The second part develops a methodology which allows the combined use of Isabelle's logics HOL and HOLCF. In the chapters 8–11 this methodology will be of vital importance for the formalization of I/O automata in Isabelle.

5.1 Introduction to Isabelle

Isabelle [Pau94] is a generic theorem proving environment that supports a number of object logics. For the reader of this thesis it is not necessary to be familiar with Isabelle's technicalities, with its theory syntax or proof tactics. Both object logics used, Isabelle/HOL and Isabelle/HOLCF, provide together with Isabelle's syntax facilities an expressive and readable specification formalism. To improve readability even further the syntax has been slightly tuned in this thesis. Furthermore, proofs are reproduced in a natural language style. For the Isabelle expert, however, it should be straightforward to identify the correspondence between the text proofs and concrete tactics.

The aim of the following subsections is to explain how definitions and proofs, which have been performed in Isabelle, are presented in this text. In addition, we give an impression of the basic ideas of Isabelle.

5.1.1 The HOL Logic

Isabelle/HOL is Isabelle's instantiation of Higher Order Logic and is very close to Gordon's HOL system [GM93]. In this thesis HOL is short for Isabelle/HOL. HOL is based on Church's formulation of simple type theory [Chu40], which has been augmented by polymorphism, type classes like in Haskell, and extension mechanisms for defining new constants and types. The syntax is that of simply typed λ-calculus with an ML-style first order language of types.

Types. The syntax of types in HOL is given by

$$\tau \quad ::= \quad v \mid (\tau_1, ..., \tau_n)t$$

where τ, $\tau_1, ..., \tau_n$ range over types, v ranges over type *variables*, and t ranges over n-ary type *constructors* ($n \geq 0$). We usually drop the parentheses if $n \in \{0, 1\}$. Greek letters (e.g. α, β) are generally used for type variables, and sans serif identifiers (e.g. list, set) are used for type operators. Binary constructors are often written infix, e.g. function types as $\alpha \to \beta$, which associate to the right. Type abbreviations $\tau = (\alpha_1, \ldots, \alpha_n)t$ simply introduce aliases for already existing types.

Further type constants used in this thesis are the basic types nat and bool, and the polymorphic types $(\alpha)\,\mathsf{set}$, $\alpha \times \beta$ and $(\alpha)\,\mathsf{list}$.

Terms. The syntax of terms in HOL is given by

$$M \quad ::= \quad c \mid v \mid (t\ u) \mid \lambda v.\, t$$

where c ranges over constants, v ranges over variables, and t and u range over terms. Sans serif identifiers (e.g. a, b, c) and non-alphabetical symbols (e.g. $\Rightarrow$, $=$, $\forall$) are generally used for constants, and italic identifiers (e.g. x, y, z) are used for variables. As usual, application associates to the left and binds most tightly. If there is no nested application, we sometimes rather write $t(u)$ instead of $t\ u$. An abstraction body ranges from the dot as far to the right as possible. Nested abstractions like $\lambda x.\, \lambda y.\, t$ are abbreviated to $\lambda x\, y.\, t$. Abstractions over pairs are written as $\lambda(p_1, p_2).\, t$.

Every term in HOL denotes a *total* function and has to be *well-typed*. Writing $t :: \tau$ indicates explicitly that the term t is of type τ. Each constant c has a a type *scheme* σ associated with it and is well typed with any substitution instance of σ. Typing rules for abstractions and applications require that $\lambda x.\, t :: \tau_1 \to \tau_2$, provided that $x :: \tau_1$ and $t :: \tau_2$, and that $(t\, u) :: \tau_2$, provided that $t :: \tau_1 \to \tau_2$ and $t :: \tau_1$. It follows that the type of a term is uniquely determined by the types of constants and variables it contains. Note that type inference is decidable in that setting.

We proceed by introducing syntax for specific constants. Lists are constructed using the "cons" operator : and the empty list $[\,]$. Instead of $a_1 : \ldots : a_n : [\,]$ we write $[a_1, \ldots, a_n]$ as usual. Operators on lists include @ (concatenation), hd, tl, length, map, $\in$, and filter. For $\mathsf{filter}\ (\lambda x.\, P(x))\ xs$ we write $[x \in xs.\, P(x)]$. Tuples are pairs (with projections fst and snd) nested to the right, e.g. $(a, b, c) = (a, (b, c))$. Natural numbers are equipped with the operators 0, Suc, $+$, and $\times$. Operators on sets include $\in$, $\cap$, $\cup$, $\setminus$. Set comprehension has the shape $\{x \mid P\}$ where x is a variable and P a predicate. More generally we write $\{t \mid_{x_1,\ldots,x_n} P\}$ where t is an arbitrary term and $x_1, \ldots, x_n$ are those variables which determine the set of values $t\ x_1 \ldots x_n$ that satisfy P. For relations R the notation $R[a]$ means $\{b.\,(a, b) \in R\}$. Function composition is defined as $(f \circ g)(x) = f(g(x))$. Conditional expressions are written $\mathbf{if}\, A\, \mathbf{then}\, B\, \mathbf{else}\, C$. For local definitions a let construct is available, and for case distinctions the syntax ($\mathbf{case}\ e\ \mathbf{of}\ c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n$) is used.

Formulae. Any term of type bool is called a *formula*. The standard logical connectives and quantifiers can be defined as λ-terms and have the normal syntax ($\forall$, $\exists$, $\neg$, $=$, $\wedge$, $\vee$, $\Rightarrow$, True, False), where earlier infixes in this list have stronger binding power than later ones. Hilbert's choice operator ε is a primitive constant of HOL, its behavior is characterized by the following rule: $P(t) \Rightarrow P(\varepsilon\, x.\, P(x))$. The term $\varepsilon\, x.\, P(x)$ denotes a fixed, but unknown value satisfying P. Such a description is always meaningful, but is only useful in proofs, if the existence of a value satisfying P can be guaranteed.

Nested $\forall$'s are abbreviated like nested λ's. For pairs and sets in conjunction with quantifiers there are extended notations, e.g. $\forall (p_1, p_2) \in S.\, P$.

Theories. A *theory* consists of a signature (type and constant declarations) and a set of axioms. A type constructor t of arity n is declared by writing $(\alpha_1, \ldots, \alpha_n)t$ and a constant c of type scheme σ is introduced by writing $c :: \sigma$.

Axioms are introduced via two canonical ways, concerning type and constant definitions. First, constants c are defined by $c \equiv t$, where t has to be well-formed w.r.t. the signature, c must not appear in t, and all type variables on the right-hand side have also to appear on the left-hand side. In practice, there is a less restrictive version of this definition scheme. First, it allows arguments of c to appear on the left, thereby abbreviating a string of λ-abstractions. Second, arguments of a pair type may explicitly written as pairs. For example, the projection function fst could be defined as follows:

$$\mathsf{fst}\ (p_1, p_2) \equiv p_1$$

Isabelle allows also inductive definitions using the keyword **inductive**, which specify the least set $\mathcal{R}$ closed under given rules. Such definitions are broken down to canonical constant definitions via a fixpoint construction.

Furthermore, types τ may be defined by demanding them to be isomorphic to some non-empty subset of an already existing type. For the reader of this thesis this definition mechanism will not appear explicitly, but merely under the surface of inductive datatype definitions, supported by HOL's datatype package. The datatype package allows to define recursive datatypes in an ML-style syntax, and generates freeness theorems and induction rules. For example, lists could be defined as

$$\textbf{datatype} \quad (\alpha)\,\mathsf{list} = [\,] \mid \alpha : (\alpha)\,\mathsf{list}$$

It is possible to define primitive recursion on these datatypes using the keyword **primrec**.

Another type, which will be frequently used, is defined as

$$\textbf{datatype} \quad (\alpha)\,\mathsf{option} = \mathsf{None} \mid \mathsf{Some}(\alpha)$$

It has an unpacking function $\mathsf{the} : (\alpha)\ \mathsf{option} \to \alpha$ such that $\mathsf{the}\,(\mathsf{Some}\ x) = x$ and $\mathsf{the}\ \mathsf{None} = \mathit{arbitrary}$ where *arbitrary* is an unknown value.

If only these two definition schemes, concerning types and constants definitions, are employed to introduce new axioms for a theory, the theory extension is guaranteed to be *definitional*. This ensures that no semantic inconsistencies can be introduced. It is one of the main characteristics of this thesis that all the theories developed are based on definitional theory extensions.

5.1.2 The HOLCF Logic

Isabelle/HOLCF [MNOS98, Reg95, Reg94] (for short HOLCF) conservatively extends HOL with concepts of domain theory such as complete partial orders, continuous functions and a fixpoint operator. By this means it supports reasoning in Scott's *Logic for Computable Functions* [SG90, Win93]. The logic of the Cambridge LCF prover [Pau87] constitutes a proper sublanguage of HOLCF. Whereas HOL is restricted to total functions, HOLCF allows arbitrary recursive function definitions and is therefore especially useful for handling infinite or partial objects.

Partial Orders as Type Classes. HOLCF uses Isabelle's *type classes* [Wen97] to distinguish HOL and LCF types. Type classes classify types just like types classify values. Given a type τ and a class C, the notation $\tau :: C$ or simply τ_C means that τ is of class C. Classes are partially ordered. The class of all HOL types is called term; it is the greatest element in the class hierarchy and the default type class in HOL.

For example, there is a type class po of partial orders which is defined as follows given a polymorphic constant $\sqsubseteq ::\ \alpha \to \alpha \to \mathsf{bool}$ for every HOL type α:

$$\begin{array}{ll} \textbf{class} & \mathsf{po} < \mathsf{term} \\ \quad \mathit{reflexive} & x \sqsubseteq x \\ \quad \mathit{transitive} & x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z \\ \quad \mathit{antisymmetric} & x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \end{array}$$

Starting from these axioms, we may derive further theorems about $\sqsubseteq$ which hold in any type of class po.

Just like in Haskell there is an **instance** declaration which tells Isabelle that some type is of a certain class. In contrast to Haskell, this comes with the obligation to prove that the axioms of the class hold for that type. As an illustrative example, let us show that the set of functions from an arbitrary type into a partial order is again a partial order. First, we make use of the overloading mechanism which allows the symbol $\sqsubseteq$ to represent different orderings on different domains. On functions $\sqsubseteq$ is defined as the pointwise extension of $\sqsubseteq$ on the range type of the function:

$$f \sqsubseteq g \quad \equiv \quad \forall x.\, f(x) \sqsubseteq g(x)$$

Given this definition, it is easy to derive the above axioms for po as theorems about pointwise ordered functions; call those theorems `refl_fun`, `trans_fun` and `antisym_fun`. Then, Isabelle can be convinced that the pointwise extension of a partial order is again a partial order by declaring

$$\textbf{instance} \;\; \rightarrow \, :: \, (\mathsf{term}, \mathsf{po})\mathsf{po} \qquad (\mathtt{refl\_fun}, \mathtt{trans\_fun}, \mathtt{antisym\_fun})$$

From this point onwards all axioms and theorems involving the generic $\sqsubseteq$ can be used for functions whose range type is of class po because the type checker now knows that the function space itself is of class po.

HOLCF provides an entire hierarchy of type classes for specific partial orders. Next after po is the class cpo (complete partial order) which demands that every ω-chain has a least upper bound. The default type class of HOLCF is pcpo (pointed complete partial order), which is a subclass of of po, equipped with a least element $\bot$. Furthermore, there is a class chfin that demands every chain to be finite. Finally, there is the class flat, which is a subclass of pcpo and chfin and postulates the axiom $x \sqsubseteq y \Rightarrow x = \bot \lor x = y$.

Continuity and Fixpoints. The notions of monotonicity and continuity are introduced according to standard domain theory [Win93] as constants of the following type:

$$\begin{array}{lcl} \mathsf{monofun} & :: & \alpha_{\mathsf{po}} \rightarrow \beta_{\mathsf{po}} \rightarrow \mathsf{bool} \\ \mathsf{cont} & :: & \alpha_{\mathsf{cpo}} \rightarrow \beta_{\mathsf{cpo}} \rightarrow \mathsf{bool} \end{array}$$

For continuous functions between pcpos there is a special type constructor, which is denoted by $\rightarrow_c$ in contrast to the standard HOL constructor $\rightarrow$. Furthermore, there are abstraction and application mechanisms for this new type with a special syntax (see Fig. 5.1).
Terms built with these continuous abstractions and applications are called *operations* or *LCF terms*. The idea of introducing this LCF sublanguage is to use type checking for determining continuity as much as possible. The only points where continuity still has to

function space	type constructor	abstraction	application
full	$\rightarrow$	$\lambda x.\, t$	$f\ t$
continuous	$\rightarrow_c$	$\Lambda x.\, t$	$f\ `t$

Figure 5.1: Function spaces in HOLCF

be handled on the proof level instead of the type level are due to the special β-reduction rule for continuous functions:

$$\mathsf{cont}(f) \Rightarrow (\Lambda x.\, f\ x)\ `t = f\ t$$

For this case, however, continuity lemmas have been proved which are able to discharge every LCF term, and a corresponding proof procedure has been implemented. Thus, for the LCF sublanguage reasoning about continuity is completely automated and hidden from the user.

The fixpoint operator

$$\mathsf{fix}\ ::\ (\alpha_{\mathsf{pcpo}} \rightarrow_c \alpha_{\mathsf{pcpo}}) \rightarrow_c \alpha_{\mathsf{pcpo}}$$

is defined as usual and enjoys the fixpoint property $\mathsf{fix}\ `f = f\ `(\mathsf{fix}\ `f)$. Note that the assumption of continuity, which is required for this equation, is incorporated in the type of fix by using $\rightarrow_c$ instead of $\rightarrow$.

From that definition the fixpoint induction rule is derived:

$$\frac{\mathsf{adm}(P) \quad P(\bot) \quad (\forall x.\, P(x) \Rightarrow P(f\ `x))}{P(\mathsf{fix}\ `f)}$$

As known from the literature [Win93] it includes the admissibility of P as an assumption. Admissibility of a predicate P is defined by the constant $\mathsf{adm}\ ::\ (\alpha_{\mathsf{cpo}} \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool}$ and expresses that P holds for the least upper bound of every chain satisfying P.

Admissibility Check. In practice, it is of vital importance that admissibility proof obligations are discharged automatically. For this reason an admissibility check has been implemented in HOLCF, which is based on the following theorems.

$$\begin{array}{lll}
\mathsf{adm}(P) \wedge \mathsf{adm}(Q) & \Rightarrow & \mathsf{adm}(\lambda x.\, P(x) \wedge Q(x)) \quad (1)\\
\mathsf{adm}(P) \wedge \mathsf{adm}(Q) & \Rightarrow & \mathsf{adm}(\lambda x.\, P(x) \vee Q(x)) \quad (2)\\
\forall y.\, \mathsf{adm}(P(y)) & \Rightarrow & \mathsf{adm}(\lambda x.\, \forall y.\, P\ y\ x) \quad (3)\\
\mathsf{cont}(f) \wedge \mathsf{cont}(g) & \Rightarrow & \mathsf{adm}(\lambda x.\, f(x) \sqsubseteq g(x)) \quad (4)\\
\mathsf{cont}(f) \wedge \mathsf{cont}(g) & \Rightarrow & \mathsf{adm}(\lambda x.\, f(x) = g(x)) \quad (5)\\
\mathsf{cont}(f) & \Rightarrow & \mathsf{adm}(\lambda x.\, f(x) \not\sqsubseteq c) \quad (6)\\
\mathsf{cont}(f) & \Rightarrow & \mathsf{adm}(\lambda x.\, f(x) \neq \bot) \quad (7)
\end{array}$$

The first five theorems reduce admissibility obligations for a predicate P to continuity obligations for the functions occurring in P by exploiting the syntactical formula structure. Note that there are no structural rules for negation or existential quantification, which is often a handicap in practice. In the case of negation there is sometimes a way out, namely for the specific cases involving constants c or $\perp$ given by the last two theorems in the list above.

This check has been automated in Isabelle/HOLCF, such that every predicate P is automatically proved to be admissible, if in prenex normal form it does not contain any $\exists$ or $\neg$ (except for the special cases described above) and if every function occurring in it stems from the (continuous) LCF sublanguage.

If this syntactic check fails, there is an extension, which exploits the following two rules, namely the substitution theorem and the fact, that every predicate with a chain-finite argument type is admissible:

$$\begin{array}{l} \mathsf{cont}(f) \wedge \mathsf{adm}(P) \Rightarrow \mathsf{adm}(\lambda x.\ P\ (f\ x)) \\ \mathsf{adm}(\lambda x :: \alpha_{\mathsf{chfin}}.\ P(x)) \end{array}$$

By combining these two theorems we get the following rule

$$\mathsf{cont}(f) \Rightarrow \mathsf{adm}(\lambda x.\ P\ ((f\ x) :: \alpha_{\mathsf{chfin}}))$$

which has been implemented as a proof procedure that enumerates all such chain-finite subterms $f(x)$ and checks if they are continuous in x. The proof procedure is automatically used whenever the standard admissibility check fails. The example $f\,`x \neq \mathsf{TT}$, where f is a continuous operation and TT the known value from the flat domain of truth values (which will be introduced formally in §5.2.3), may serve to illustrate this test, as it cannot be discharged by the structural rules described above. See §10 for several further applications.

Recursive Domains. HOLCF provides a package for the convenient definition of recursive domains [Ohe95]. It can handle mutually recursive definitions of free datatypes commonly used in functional programming, even infinite ones with non-strict constructors. It determines and proves the characteristic properties of each domain defined, including freeness, as well as induction and coinduction principles.

The HOLCF package, invoked with the keyword **domain**, works similar to the HOL datatype package, invoked with the keyword **datatype**. Internally it is based on a generalization of the type definition mechanism used in the HOL package tailored for Scott's domain theory. Specifically, for each datatype two axioms are introduced which characterize this type definition mechanism. A third axiom requires the limit of every chain approximating a given element x to be x already. This ensures that the resulting datatype represents indeed the initial (i.e. least) solution of the defining domain equation. The package will be explained in detail with the example of possibly infinite sequences in §6.

5.1.3 Presentation of Isabelle Proofs

Isabelle does not distinguish between theorems and proof rules, both are represented as formulae. If formulae are of the form $A_1 \wedge \ldots \wedge A_n \Rightarrow P$, we often employ a rule-style syntax and also allow $A_1, \ldots, A_n$ and P to be labeled. For example, the induction rule on natural numbers would look like

$$\frac{\mathcal{A}_1\colon\ P(0) \quad \mathcal{A}_2\colon\ \forall x.\ P(x) \Rightarrow P(\mathsf{Suc}(x))}{\mathcal{C}_1\colon\ \forall n.\ P(n)}\ (\mathit{nat\text{-}ind})$$

which in this thesis is regarded to be the same as[1]

$$P(0) \wedge (\forall x.\ P(x) \Rightarrow P(\mathsf{Suc}(x))) \Rightarrow \forall n.\ P(n)$$

We call $A_1, \ldots, A_n$ the *assumptions* and P the *conclusion* of the theorem/rule. If space is scarce, we write $A_1, \ldots, A_n$ in a vertical instead of horizontal list.

Proofs in Isabelle are performed by backward or forward reasoning in a natural deduction style. Normally the overall proof structure is a backward proof, where we start with a goal and refine it to progressively simpler subgoals until all have been solved. Forward reasoning is used to derive further facts from the assumptions.

As an example for backward reasoning let us verify the simple equation

$$\mathcal{C}_1\colon\quad \forall n.\ n + 0 = n$$

about natural numbers. We apply $(\mathit{nat\text{-}ind})$ in a backward fashion and therefore reduce the initial goal $\mathcal{C}_1$ to the two subgoals

$$\begin{array}{ll} \mathcal{C}_2\colon & 0 + 0 = 0 \\ \mathcal{C}_3\colon & \forall m.\ m + 0 = m \Rightarrow \mathsf{Suc}(m) + 0 = \mathsf{Suc}(m) \end{array}$$

which are both automatically solved by Isabelle via rewriting. As an example of forward reasoning let us prove

$$\frac{\mathcal{A}_1\colon\ P \quad \mathcal{A}_2\colon\ P \Rightarrow Q \quad \mathcal{A}_3\colon\ Q \Rightarrow R}{\mathcal{C}_1\colon\ R}$$

using the modus ponens rule $P \wedge (P \Rightarrow Q) \Rightarrow Q$. First, we apply modus ponens on $\mathcal{A}_1$ and $\mathcal{A}_2$ and get the new assumption $\mathcal{A}_4$: Q. Then, the same rule is applied once more on $\mathcal{A}_4$ and $\mathcal{A}_3$ which yields $\mathcal{C}_1$.

Note how the handling of assumptions is presented in this example. Whereas Isabelle displays every assumption after every proof step, we have simply recorded the change of the assumption list by introducing new assumptions via a new label. This shortens the

[1] Actually, there is a syntactical difference in Isabelle between connectives of the meta-logic and the object-logics and thus between rules and formulas. However, this difference can be neglected in our context.

proof description considerably and focuses the attention of the reader to the actual changes of the proof step.

Apart from these basic proof rules, which are internally based on resolution and higher-order unification, Isabelle provides two tools that work at a higher-level aiming for semi-automatic theorem proving. The *classical reasoner* performs different styles of proof searches using a tableau calculus for predicate logic. The *simplifier* performs conditional and unconditional rewriting and exploits contextual information. Both tools are sound as they work by building proofs. The user may himself define new proof strategies, so called *tactics*, using operators called *tacticals*.

In this thesis the presentation of proofs, which have been performed in Isabelle, follows rather closely the actual proof scripts, using the labeling mechanism described above. However, steps which are trivial for the reader, are omitted frequently. On the other side, the classical reasoner or the simplifier sometimes perform steps that are not obvious to the reader. In such cases, additional substeps are introduced in the presentation. In any case, it should be straightforward to replay the proof scripts simultaneously with the proof descriptions.

5.2 Methodology for HOLCF

In this section we address methodological questions concerning the combined use of different logics. In particular, for the combination of HOL and LCF, we explain the importance of a well structured approach offering different layers. In consequence, we propose a well defined interface between HOL and LCF.

5.2.1 Methodological Treatment of HOLCF

Although HOLCF is built on top of HOL, both logics were initially thought of as two separate worlds [Reg94] — HOL's set theoretic one of total functions and LCF's domain theoretic one of continuous functions. Methodologically this means — at least for users — that HOLCF domains should not be mixed up with HOL types, and partial functions not with total ones.

Two reasons could support this philosophy: First, one of HOLCF's historical roots is the LCF prover [Pau87], a system which was completely restricted to Scott's logic of computable functions. A key idea of HOLCF was to build a copy of LCF with the main difference that domain theory should not be axiomatized/hardwired, but represent a definitional extension of HOL. Thus, from the tool builder's point of view HOL was not much more than the vehicle allowing a theoretical foundation of LCF. Second, technical problems were to expect when merging HOL and LCF types and terms. In particular, the continuity check supported merely the LCF sublanguage.

However, from a methodological point of view it is not at all convenient to stay either in LCF or in HOL. LCF provides much more sophisticated constructions, like domains and arbitrary recursion. In return, reasoning in LCF is much more complicated than in HOL. Furthermore, for many specification and verification tasks these sophisticated constructions are not needed, or even disturbing. Not every type should be modeled as a domain, not every function has to be continuous.

What is desirable is rather the possibility to choose the adequate logic for each part of a given problem. Furthermore, theories and proofs of HOL should be reusable in HOLCF as much as possible, as HOL provides large theory libraries and a well-established proof infrastructure.

Therefore we propose the following methodology in order to get a fruitful synthesis of HOL and HOLCF:

- The specification should be structured into different logical layers, which are separated by clear and definite interfaces. Specifically, the order of layers should be HOL–LCF–HOL. This means that every term in HOLCF should obey the type structure given in the following figure, where asterisks represent type schemes and vertical lines denote logical interfaces.

HOL	**LCF**	**HOL**
$* \to * \to * \cdots$	$\cdots * \to_c * \to_c * \cdots$	$\cdots * \to * \to *$

- The guideline for such a structure should be: express as much as possible in the HOL basis (first layer), express as much as necessary in the LCF extension (second layer), use HOL again only to express properties *about* your specification (third layer).

Technically this methodology demands that HOLCF, being composed of HOL and LCF, has to be more than just the sum of its parts. The interfaces have to guarantee that reasoning in a layer does not interfere with concepts of the layer above it. In particular, continuity in the second layer should not depend on the underlying HOL.

In the following section §5.2.2 we present such an interface which is designed to ease the transition from HOL to its LCF extension and provides completely automated continuity support. In §5.2.3 we will illustrate this method by introducing the domain of truth values on top of the HOL type of boolean values.

For the transition from LCF back to HOL additional concepts are not needed, but experience shows that one should be very careful not to involve any LCF terms in the third layer thereby returning to the second layer again.

5.2.2 A Lifting Interface

The simple key idea of the HOL/LCF interface is to lift HOL types to flat domains, to extend functions on them in a strict way, and — most importantly — to prove that such constructions yield only continuous functions.

Lifting of Types. We introduce a type constructor $(\alpha)\,\mathsf{lift}$ that lifts HOL types to flat domains by defining

$$\textbf{datatype}\quad (\alpha_{\mathsf{term}})\,\mathsf{lift} = \mathsf{Undef} \mid \mathsf{Def}\ \alpha_{\mathsf{term}}$$

and declaring the least element and the approximation ordering as follows:

$$\begin{array}{lcl} \perp & \equiv & \mathsf{Undef} \\ x \sqsubseteq y & \equiv & (x = \mathsf{Undef} \lor x = y) \end{array}$$

Note that $\perp$ and $\sqsubseteq$ are overloaded and this definition only fixes their meaning at type $(\alpha_{\mathsf{term}})\,\mathsf{lift}$. Both definitions together yield immediately the property which justifies the desired arity for $(\alpha)\,\mathsf{lift}$:

$$\textbf{instance}\ \ \mathsf{lift}\ ::\ (\mathsf{term})\mathsf{flat} \qquad (x \sqsubseteq y \Rightarrow x = \perp \lor x = y)$$

Furthermore, we define an unpacking function $\mathsf{the} :: (\alpha)\,\mathsf{lift} \to \alpha$ such that $\mathsf{the}\,(\mathsf{Def}\ x) = x$ and $\mathsf{the}(\mathsf{Undef}) = \mathit{arbitrary}$ where $\mathit{arbitrary}$ is an unknown value.

As Undef plays the rôle of the least element, from now on $\perp$ is used in favour of Undef. Moreover, every theorem generated for $(\alpha)\mathsf{lift}$ by the datatype package of HOL is reformulated with $\perp$ instead of Undef, so that Undef is completely hidden from the user.

Lifting of Functions. In order to lift also functions on HOL types to continuous operations on domains, we introduce the functionals $\mathsf{lift\text{-}fun}_1$ and $\mathsf{lift\text{-}fun}_2$.

$$\begin{array}{lcl} \mathsf{lift\text{-}fun}_1 & :: & (\alpha_{\mathsf{term}} \to \beta_{\mathsf{pcpo}}) \to (\alpha_{\mathsf{term}}\,\mathsf{lift} \to_c \beta_{\mathsf{pcpo}}) \\ \mathsf{lift\text{-}fun}_2 & :: & (\alpha_{\mathsf{term}} \to \beta_{\mathsf{term}}) \to (\alpha_{\mathsf{term}}\,\mathsf{lift} \to_c \beta_{\mathsf{term}}\,\mathsf{lift}) \end{array}$$

The former lifts only the argument type of a HOL function, the latter both argument and range type. Basically, they extend HOL functions in a strict way. Technically they are defined via two auxiliary functions $\mathsf{lift\text{-}fun}'_1$ and $\mathsf{lift\text{-}fun}'_2$

$$\begin{array}{lcl} \mathsf{lift\text{-}fun}_1\,f & \equiv & \Lambda x.\ \mathsf{lift\text{-}fun}'_1\,f\ x \\ \mathsf{lift\text{-}fun}_2\,f & \equiv & \Lambda x.\ \mathsf{lift\text{-}fun}'_2\,f\ x \end{array}$$

which have almost the same type as $\mathsf{lift\text{-}fun}_1$ and $\mathsf{lift\text{-}fun}_2$, merely differing in the type constructor of the produced function:

$$\begin{array}{lcl} \mathsf{lift\text{-}fun}'_1 & :: & (\alpha_{\mathsf{term}} \to \beta_{\mathsf{pcpo}}) \to (\alpha_{\mathsf{term}}\,\mathsf{lift} \to \beta_{\mathsf{pcpo}}) \\ \mathsf{lift\text{-}fun}'_2 & :: & (\alpha_{\mathsf{term}} \to \beta_{\mathsf{term}}) \to (\alpha_{\mathsf{term}}\,\mathsf{lift} \to \beta_{\mathsf{term}}\,\mathsf{lift}) \end{array}$$

These auxiliary functions define the actual lifting behaviour:

$$\begin{array}{lcl} \mathsf{lift\text{-}fun}'_1\, f & \equiv & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\ & & \qquad\qquad \bot \quad \Rightarrow\ \bot \\ & & \qquad\quad\ \mid \mathsf{Def}\ a\ \Rightarrow\ (f\ a) \\ \mathsf{lift\text{-}fun}'_2\, f & \equiv & \lambda x.\ \textbf{case}\ x\ \textbf{of} \\ & & \qquad\qquad \bot \quad \Rightarrow\ \bot \\ & & \qquad\quad\ \mid \mathsf{Def}\ a\ \Rightarrow\ \mathsf{Def}\ (f\ a) \end{array}$$

The functions $\mathsf{lift\text{-}fun}'_1$ and $\mathsf{lift\text{-}fun}'_2$ are needed in an intermediate step in order to allow $\mathsf{lift\text{-}fun}_1$ and $\mathsf{lift\text{-}fun}_2$ to code the continuity of the lifted function into its type. This is not possible before proving this continuity as theorems about $\mathsf{lift\text{-}fun}'_1$ and $\mathsf{lift\text{-}fun}'_2$, which will be done in the next paragraph. Once the continuity is established, we can derive the desired behavior for $\mathsf{lift\text{-}fun}_1$ and $\mathsf{lift\text{-}fun}_2$, expressed by the following equations:

$$\begin{array}{lcl} \mathsf{lift\text{-}fun}_1\, f\ `\bot & = & \bot \\ \mathsf{lift\text{-}fun}_1\, f\ `(\mathsf{Def}\ x) & = & (f\ x) \\ \mathsf{lift\text{-}fun}_2\, f\ `\bot & = & \bot \\ \mathsf{lift\text{-}fun}_2\, f\ `(\mathsf{Def}\ x) & = & \mathsf{Def}\ (f\ x) \end{array}$$

Continuity Support. We present now some continuity theorems, which will turn out to form a sufficient HOL/LCF interface.

Lemma 5.2.1 (Continuity of Lifting)
The following continuity theorems hold about $\mathsf{lift\text{-}fun}'_1$ and $\mathsf{lift\text{-}fun}'_2$:

$$\frac{\mathcal{A}_1\!:\ \forall y.\ \mathsf{cont}\ (\lambda x.\ (f\ x)\ y) \quad \mathcal{A}_2\!:\ \mathsf{cont}(g)}{\mathcal{C}_1\!:\ \mathsf{cont}(\lambda x.\ \mathsf{lift\text{-}fun}'_1\ (f\ x)\ (g\ x))}\ (cont_1) \qquad \frac{\mathsf{cont}(g)}{\mathsf{cont}(\lambda x.\ \mathsf{lift\text{-}fun}'_2\, f\ (g\ x))}\ (cont_2)$$

Proof.
In order to prove $cont_1$ we apply the rule

$$\frac{\mathsf{cont}(f) \quad \forall y.\ \mathsf{cont}\ (f\ y) \quad \mathsf{cont}(g)}{\mathsf{cont}(\lambda x.\ (f\ x)\ (g\ x))}$$

which reduces the initial goal $\mathcal{C}_1$ to the subgoals

$$\begin{array}{ll} \mathcal{C}_2\!: & \mathsf{cont}(\lambda x.\ \mathsf{lift\text{-}fun}'_1\ (f\ x)) \\ \mathcal{C}_3\!: & \mathsf{cont}(\lambda x.\ \mathsf{lift\text{-}fun}'_1\ (f\ y)\ x) \\ \mathcal{C}_4\!: & \mathsf{cont}(g) \end{array}$$

$\mathcal{C}_4$ follows immediately by assumption $\mathcal{A}_2$. $\mathcal{C}_3$ holds as every strict function from a flat domain is continuous: $\mathsf{lift\text{-}fun}'_1$ is strict in its second argument $x :: \alpha_{\mathsf{flat}}$. $\mathcal{C}_2$ is reduced using the rule $\forall y.\, \mathsf{cont}(\lambda x.\, f\ x\ y) \Rightarrow \mathsf{cont}(f)$ to

$$\mathcal{C}_5\colon\quad \forall y.\, \mathsf{cont}(\lambda x.\, \mathsf{lift\text{-}fun}'_1\ (f\ x)\ y)$$

Let us eliminate the universal quantifier by assuming a fixed, but unknown c instead of y. Then, we distinguish the two cases $c = \bot$ and $c = \mathsf{Def}(a)$ for some a. Unfolding the definition of $\mathsf{lift\text{-}fun}'_1$ the continuity obligation holds trivially in the first case, whereas in the second case it boils down to $\mathcal{A}_1$.

The second rule *cont*$_2$ follows by a similar argument. □

Note that the continuity of functions which are the result of the lifting functionals $\mathsf{lift\text{-}fun}_1$ or $\mathsf{lift\text{-}fun}_2$ is given as a special case of *cont*$_1$ and *cont*$_2$, where $g := \lambda x.\, x$ and x does not occur in f.

Completeness of Continuity Support. In the sequel we explain why the theorems *cont*$_1$ and *cont*$_2$ are sufficient to discharge every continuity obligation about terms that satisfy the following interface prescription: every term is lifted from HOL to LCF merely by the three functions Def, $\mathsf{lift\text{-}fun}_1$, or $\mathsf{lift\text{-}fun}_2$. The proof is by an inductive argument over the structure of terms, but will not be given formally here. We merely sketch the intuitive idea.

First, consider the case of a term which refers merely to the second and third logical layer. In long $\beta\eta$-normal form such a term t has the structure

$$t \quad = \quad \lambda x_1 \ldots x_n.\, h\, f_1\, f_2\ \ldots\ \text{`}cf_1\ \text{`}cf_2\ \text{`} \ldots$$

where the f_i are terms of the same structure as t and the cf_i are LCF terms. For every continuity obligation $\mathsf{cont}(\lambda x.\, t(x))$ the type of variable x has to be at least of class po, as otherwise continuity is not expressible. Thus, x either occurs in the f_i or cf_i or nowhere. In the last case, the proposition holds trivially. In the first case the argument applies inductively as the f_i have the same structure as t. In the second case the rule $\mathsf{cont}(f) \Rightarrow \mathsf{cont}(\lambda x.\ c\ (f\ x))$ — which is part of the standard continuity check — is used to reduce the obligation $\mathsf{cont}(\lambda x.\ t(x))$ to a continuity obligation about a pure LCF term, which is continuous because of its syntactical structure.

Now assume that t covers the first logical layer as well. As $\mathsf{lift\text{-}fun}_1$, $\mathsf{lift\text{-}fun}_2$, and Def represent the interface between HOL and LCF, this means that the cf_i must have the form $\mathsf{lift\text{-}fun}_1 f\ g$, $\mathsf{lift\text{-}fun}_2 f\ g$, or $\mathsf{Def} f$. Consider $\mathsf{lift\text{-}fun}_1 f\ g$ first. Lemma *cont*$_1$ together with the definition of $\mathsf{lift\text{-}fun}_1$ reduces any continuity obligation about $\mathsf{lift\text{-}fun}_1 f\ g$ to the continuity of f and g, which have in turn the same structure as t. Thus, the entire argument applies inductively.

Note that this reasoning allows even an arbitrary switching between HOL and LCF layers. However, the interface prescription has to be respected, which means in particular that the continuity of variables x in an LCF layer beneath another LCF layer is restricted to those x which have been lifted to a flat domain. This is in general not the intended way to deal with pcpo variables. Thus, arbitrary switching is prohibited from a methodological point of view. This is the reason why there are not any continuity lemmas for $\mathsf{cont}(\lambda x.\, \mathsf{Def}\ (f\ x))$ or $\mathsf{cont}(\lambda x.\ \mathsf{lift\text{-}fun}'_2\ (f\ x) g)$. This in turn explains why lemma $cont_2$ differs slightly from lemma $cont_1$. Besides, the argumentation for $\mathsf{lift\text{-}fun}_2$ is exactly the same as for $\mathsf{lift\text{-}fun}_1$. This finishes the completeness proof for $\mathsf{lift\text{-}fun}_2$ as well.

At first sight one may argue that a third lifting functional would be interesting which lifts only the range type of a function and could be defined as

$$\begin{array}{lcl} \mathsf{lift\text{-}fun}_3 & :: & (\alpha_{\mathsf{pcpo}} \to \beta_{\mathsf{term}}) \to (\alpha_{\mathsf{pcpo}} \to_c \beta_{\mathsf{term}}\, \mathsf{lift}) \\ \mathsf{lift\text{-}fun}_3\, f & \equiv & \Lambda x.\, \mathsf{Def}\ (f\ x) \end{array}$$

However, this is not reasonable, as $\mathsf{lift\text{-}fun}_3$ is not even monotone in its second argument, which is due to the fact that the first argument $f :: (\alpha_{\mathsf{pcpo}} \to \beta_{\mathsf{term}})$ allows a transition from LCF to HOL, which cannot be supported continuously in general.

Result. Therefore, by adding the theorems $cont_1$ and $cont_2$ to the continuity tactic already available for LCF terms, Isabelle is now able to discharge automatically every continuity obligation of mixed HOL and LCF terms, provided that all HOL terms are "lifted" into LCF using only Def, $\mathsf{lift\text{-}fun}_1$ or $\mathsf{lift\text{-}fun}_2$. The result is a well-defined interface between HOL and its LCF extension that allows to integrate HOL terms without the drawback of manual continuity proofs.

5.2.3 An Application: Truth Values

A major application of the $(\alpha)\mathsf{lift}$ theory are argument types of recursive domain constructions. See §6 for the example of possibly infinite sequences. Another nice example for the use of type lifting are the operations on truth values, which will be introduced next.

The type tr of truth values is simply defined by lifting the type bool

$$\mathsf{tr} \;=\; (\mathsf{bool})\,\mathsf{lift}$$

Then, the truth values $\mathsf{TT}, \mathsf{FF}, \bot$ are introduced by associating them to the corresponding boolean values True and False:

$$\begin{array}{lcl} \mathsf{TT} & \equiv & \mathsf{Def}(\mathsf{True}) \\ \mathsf{FF} & \equiv & \mathsf{Def}(\mathsf{False}) \end{array}$$

The continuous conditional expression — written as **If** tr **then** e_1 **else** e_2 — is obtained by lifting HOL's **if** b **then** e_1 **else** e_2 in the first argument.

$$
\begin{array}{lcl}
\mathsf{If\text{-}then\text{-}else} & :: & \mathsf{tr} \to_c \alpha \to_c \alpha \to_c \alpha \\
\mathsf{If\text{-}then\text{-}else} & \equiv & (\Lambda tr\ e_1\ e_2.\, \mathsf{lift\text{-}fun}_1\ (\lambda b.\, \textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ `tr)
\end{array}
$$

We write **If** tr **then** e_1 **else** e_2 instead of $\mathsf{If\text{-}then\text{-}else}\,`tr\,`e_1\,`e_2$. Then, the logical connectives can be defined easily. The negation neg is obtained by lifting $\neg$ in both argument and result type

$$
\begin{array}{lcl}
\mathsf{neg} & :: & \mathsf{tr} \to_c \mathsf{tr} \\
\mathsf{neg} & \equiv & \mathsf{lift\text{-}fun}_2(\neg)
\end{array}
$$

whereas conjunction andalso and disjunction orelse are definable via the **If** tr **then** e_1 **else** e_2 construct:

$$
\begin{array}{lcl}
\mathsf{andalso}, \mathsf{orelse} & :: & \mathsf{tr} \to_c \mathsf{tr} \to_c \mathsf{tr} \\
\mathsf{andalso} & \equiv & (\Lambda x\ y.\, \textbf{If}\ x\ \textbf{then}\ y\ \textbf{else}\ \mathsf{FF}) \\
\mathsf{orelse} & \equiv & (\Lambda x\ y.\, \textbf{If}\ x\ \textbf{then}\ \mathsf{TT}\ \textbf{else}\ y)
\end{array}
$$

Theorems about these connectives can now be derived from theorems about the corresponding boolean connectives.

Chapter 6

Sequences in HOLCF

We develop a theory of possibly infinite sequences within Isabelle/HOLCF which will be used to model runs of I/O automata in subsequent chapters. By distinguishing total and partial sequences it is possible to define powerful functions like infinite concatenation using simple recursive equations. Using the interface methodology of the previous chapter sequence elements are formalized in HOL. This enables reuse and considerably increases the degree of automation. The package offers inductive as well as coinductive proof rules. For the latter we provide a tailored proof infrastructure and methodological guidance. Using these proof principles about 200 basic theorems have been derived.

6.1 Introduction

Executions and traces of I/O automata are modeled by possibly infinite sequences. In the literature and our introduction to I/O automata in §2 these sequences are treated informally. When formalizing the meta-theory of I/O automata rigorously, it is however necessary to provide a formal sequence model as well. There are several possibilities for such a model. We decided to take lazy lists, formally based on Scott's domain theory. In the following chapter we will provide a comparison with other sequence formalizations together with a justification for our design decision.

Lazy lists are defined by a simple recursive domain equation using HOLCF's domain package. In this setting arbitrary recursion is possible and induction as well as coinduction (take lemma, bisimulation) is provided. A further advantage is that sequences may profit from the powerful constructions in HOLCF, whereas sequence elements can be treated in the simpler HOL. This is possible because of the interface methodology of the previous chapter. Furthermore, we will provide proof infrastructure and a tailored methodology for coinductive proofs, which otherwise do not match recursive definitions. The entire sequence package comprises a remarkable amount of theorems, which mostly have been derived very easily. A selection of the theorems is displayed in Appendix B.1[1].

The chapter is organized as follows. In §6.2 sequences are defined using HOLCF's domain package. Furthermore, the definition of recursive functions is explained. In §6.3 the sequence model is generalized to deal with HOL elements. Then, the proof principles are introduced and explained by means of a simple example in §6.4. This reveals two situations where coinduction is more useful than induction: if the admissibility check fails or if the functions involved are more naturally defined corecursively. Both cases are dealt with from a logical as well as methodological point of view in §6.5 and §6.6, respectively.

6.2 Sequences as Recursive Domains

In the following we define the datatype of possibly infinite sequences (also known as *lazy lists*) over elements of any pcpo type α. Note that throughout the entire section the default type class is assumed to be pcpo.

Definition 6.2.1 (Type of Sequences)
Using the HOLCF domain package sequences are defined by the simple recursive domain equation

$$\textbf{domain} \quad (\alpha)\,\mathsf{seq} = \mathsf{nil} \;|\; (\mathsf{HD} : \alpha) \,\#\, (\textbf{lazy}\,(\mathsf{TL} : (\alpha)\,\mathsf{seq}))$$

where nil and the right-associative "cons"-operator[2] # are the constructors and HD and TL the selectors of the datatype. □

The domain package assumes domain constructors to be strict by default, therefore # is strict in its first argument and lazy in the second. This means, that elements of the type $(\alpha)\,\mathsf{seq}$ come in three flavors:

Finite total sequences:	$a_1 \# \ldots \# a_n \# \mathsf{nil}$
Finite partial sequences:	$a_1 \# \ldots \# a_n \# \perp$
Infinite sequences:	$a_1 \# a_2 \# a_3 \ldots$

A number of theorems are automatically proved by the package, including case distinctions, strictness and definedness properties, the application cases of the selectors, and the distinctness and injectivity of the constructors.

In particular, a continuous case functional is defined, which uses the same syntax as the HOL **case** construct, and a take functional of type

$$\mathsf{take} \;::\; \mathsf{nat} \to (\alpha)\,\mathsf{seq} \to_c (\alpha)\,\mathsf{seq}$$

[1]The sequence package is part of the Isabelle distribution and can be found on WWW under `http://www4.informatik.tu-muenchen.de/~isabelle/library/HOLCF/IOA/index.html`.

[2]Note that due to technical reasons the symbol does not reflect the actual Isabelle definition.

is introduced that is characterized by the following equations:

$$\begin{array}{lcl} \mathsf{take}\; n \text{ `} \bot & = & \bot \\ \mathsf{take}\; 0 \text{ `} s & = & \bot \\ \mathsf{take}\; (\mathsf{Suc}\; n) \text{ `} \mathsf{nil} & = & \mathsf{nil} \\ \mathsf{take}\; (\mathsf{Suc}\; n) \text{ `} (a \,\#\, s) & = & a \,\#\, \mathsf{take}\; n \text{ `} s \end{array}$$

Furthermore, several proof principles are derived, which will be discussed in §6.4. For more details about the domain package, especially concerning its logical foundations, the reader is referred to [MNOS98, Ohe95].

We are now capable to capture the three flavors of sequences formally.

Definition 6.2.2 (Finite, Partial, Infinite)
The set fin of finite total sequences is defined inductively as the least set of sequences satisfying the following two rules:

$$\textbf{inductive} \qquad \frac{}{\mathsf{nil} \in \mathsf{fin}}\,(\mathit{Finite}\text{-}0) \qquad \frac{s \in \mathsf{fin} \quad a \neq \bot}{a \,\#\, s \in \mathsf{fin}}\,(\mathit{Finite}\text{-}n)$$

Isabelle's syntax translation mechanism is used to write $\mathsf{Finite}(x)$ for $x \in \mathsf{fin}$.

Infinite and partial sequences are characterized by the following predicates:

$$\begin{array}{lcl} \mathsf{Infinite}(s) & \equiv & (\forall n.\, \mathsf{take}\; n \text{ `} s \neq s) \\ \mathsf{Partial}(s) & \equiv & \neg \mathsf{Infinite}(s) \wedge \neg \mathsf{Finite}(s) \end{array}$$

□

Isabelle's machinery for inductive definitions generates an induction principle for the predicate $\mathsf{Finite}(x)$ which can in particular be used to derive the following characterization of finite sequences:

$$\begin{array}{rcl} & \mathsf{Finite}(\mathsf{nil}) & \\ \neg & \mathsf{Finite}(\bot) & \\ a \neq \bot \Rightarrow & \mathsf{Finite}(a \,\#\, s) & = \quad \mathsf{Finite}(s) \end{array}$$

Alternatively, finite total sequences could be defined directly by these rules using the corresponding fixpoint definition. Then, however, the induction principle for finite total sequences has to be derived from the induction principle for non-infinite sequences as provided by the domain package (see §6.4). But this in turn is not possible — at least not by domain theoretic reasoning — as the predicate $\neg\mathsf{Infinite}(s)$ is not admissible.

Let us now show by the example of the mapping functional how recursive functions on sequences can be defined.

Definition 6.2.3 (Mapping Function)
The mapping functional smap[3] has type

$$\mathsf{smap} \;::\; (\alpha \to_c \beta) \to_c (\alpha)\,\mathsf{seq} \to_c (\beta)\,\mathsf{seq}$$

and is defined as the following fixpoint:

$$\begin{array}{lcl} \mathsf{smap} & \equiv & \mathsf{fix}\,`(\Lambda h.\,(\Lambda f.\,\Lambda s.\,\textbf{case}\;s\;\textbf{of} \\ & & \qquad\qquad\qquad\quad \mathsf{nil} \;\Rightarrow\; \mathsf{nil} \\ & & \qquad\qquad\quad\;\; \mid\; x \mathbin{\#} xs \;\Rightarrow\; (f\,`x) \mathbin{\#} (h\,`f\,`xs)) \end{array}$$

□

The recursive equations characterizing this function are obtained in a generic way using a tactic which essentially requires the fixpoint theorem and the continuity of the body of the fixpoint. In the case of smap the latter is trivially fulfilled because the function f and the case distinction are continuous by their definition using the type constructor $\to_c$.

Corollary 6.2.4
The following recursive equations characterize smap:

$$\begin{array}{rcl} \mathsf{smap}\,`f\,`\bot & = & \bot \\ \mathsf{smap}\,`f\,`\mathsf{nil} & = & \mathsf{nil} \\ a \neq \bot \Rightarrow \;\; \mathsf{smap}\,`f\,`(a \mathbin{\#} s) & = & (f\,`a) \mathbin{\#} \mathsf{smap}\,`f\,`s \end{array}$$

As this characterization is derived automatically, we will not give the fixpoint definition for future function definitions, but introduce them simply by the resulting recursive equations.

Definition 6.2.5 (Further Recursive Functions)
Filtering and finite/infinite concatenation are defined as follows:

$$\begin{array}{lcl} \mathsf{sfilter} & :: & (\alpha \to_c \mathsf{tr}) \to_c (\alpha)\,\mathsf{seq} \to_c (\alpha)\,\mathsf{seq} \\ \mathsf{sconc} & :: & (\alpha)\,\mathsf{seq} \to_c (\alpha)\,\mathsf{seq} \to_c (\alpha)\,\mathsf{seq} \\ \mathsf{sflatten} & :: & ((\alpha)\,\mathsf{seq})\,\mathsf{seq} \to_c (\alpha)\,\mathsf{seq} \end{array}$$

We write $x \oplus y$ for sconc $`x\,`y$. The characterizing recursive equations are derived immediately.

$$\begin{array}{rcl} \mathsf{sfilter}\,`P\,`\bot & = & \bot \\ \mathsf{sfilter}\,`P\,`\mathsf{nil} & = & \mathsf{nil} \\ a \neq \bot \Rightarrow \;\; \mathsf{sfilter}\,`P\,`(a \mathbin{\#} s) & = & \textbf{If}\;\; P\,`a\;\;\textbf{then}\;a \mathbin{\#} \mathsf{sfilter}\,`P\,`s \\ & & \qquad\qquad\;\textbf{else}\;\mathsf{sfilter}\,`P\,`s \end{array}$$

[3]The names of recursive functions on sequences are preceded by a 's' in order to distinguish them from the corresponding functions on lists.

$$\begin{array}{lcl} \bot \oplus s & = & \bot \\ \mathsf{nil} \oplus s & = & x \\ (a \# s) \oplus t & = & a \# (s \oplus t) \end{array}$$

$$\begin{array}{lcl} \mathsf{sflatten}\,`\bot & = & \bot \\ \mathsf{sflatten}\,`\mathsf{nil} & = & \mathsf{nil} \\ \mathsf{sflatten}\,`(s \# ss) & = & s \oplus \mathsf{sflatten}\,`ss \end{array}$$

□

Further recursive functions defined include stakewhile, sdropwhile and slast. They are defined in Appendix A in the way one would expect from functional programming with lazy lists. Here, only the following important predicate is presented.

Definition 6.2.6 (Forall Predicate)
The sforall predicate tests whether all elements of a sequence fulfill a given predicate P.

$$\begin{array}{lcl} \mathsf{sforall} & :: & (\alpha \to_c \mathsf{tr}) \to (\alpha)\,\mathsf{seq} \to \mathsf{bool} \\ \mathsf{sforall}\; P\; s & \equiv & \mathsf{sforall_c}\,`P\,`s \neq \mathsf{FF} \end{array}$$

It is realized by a computable function, the continuous predicate

$$\mathsf{sforall_c} \;::\; (\alpha \to_c \mathsf{tr}) \to_c (\alpha)\,\mathsf{seq} \to_c \mathsf{tr}$$

which "runs down" the sequence checking $P\,`x$ for every element x. The function $\mathsf{sforall_c}$ is characterized by the following recursive equations:

$$\begin{array}{rlcl} & \mathsf{sforall_c}\,`P\,`\bot & = & \bot \\ & \mathsf{sforall_c}\,`P\,`\mathsf{nil} & = & \mathsf{TT} \\ a \neq \bot \Rightarrow & \mathsf{sforall_c}\,`P\,`(a \# s) & = & P\,`a \;\mathsf{andalso}\; \mathsf{sforall_c}\,`P\,`s \end{array}$$

Therefore, the predicate sforall is true if $\mathsf{sforall_c}$ terminates and returns TT (for finite xs) or if it does not terminate, therefore returning $\bot$ (for infinite xs). □

Apart from the domain theoretic prefix $\sqsubseteq$ we define a further prefix notion.

Definition 6.2.7 (Total Prefix)
The notion of a *total prefix* is defined as follows:

$$\begin{array}{lcl} \ll & :: & (\alpha)\,\mathsf{seq} \to (\alpha)\,\mathsf{seq} \to \mathsf{bool} \\ s \ll t & \equiv & (\exists u.\; t = s \oplus u) \end{array}$$

□

The two prefix notions are in a sense complementary, the following rules highlight their different characteristics:

$$\frac{\mathsf{Finite}(s) \quad s \sqsubseteq t}{s = t} \qquad \frac{\neg\mathsf{Finite}(s) \quad s \ll t}{s = t}$$

This means that for finite total arguments $\sqsubseteq$ degenerates, whereas $\ll$ makes sense only for them. Note that in I/O automata theory the prefix notion $\ll$ is employed. Finite prefixes will as well be relevant for the corecursive characterization of recursive functions (see §6.6).

Finally, we justify the design decision to model sequences in three flavors (finite, partial, infinite) instead of only two (partial, infinite), as done in [SS95]. Actually, there are a number of advantages:

1. We are in in accordance with standard functional programming of lazy lists [BW88].

2. It is possible to define the concatenation operator $\oplus$ in a continuous way. This means, that for every partial sequence s holds $s \oplus t = s$. If we restricted us to two flavors, however, $\oplus$ would not be even monotone. For example we would get $\bot \oplus a \,\#\, \mathsf{nil} = a \,\#\, \mathsf{nil}$ and $b \,\#\, \bot \oplus a \,\#\, \mathsf{nil} = b \,\#\, a \,\#\, \mathsf{nil}$ with $a \,\#\, \mathsf{nil} \not\sqsubseteq a \,\#\, b \,\#\, \mathsf{nil}$ although $\bot \sqsubseteq b \,\#\, \bot$. A continuous concatenation allows us to to reason about $\oplus$ via structural induction, which is the natural proof principle for sequences. Second, we can even define the infinite concatenation sflatten using $\oplus$ and reason about it via structural induction. Infinite concatenation will be crucial for the correctness proof of refinements (see §9). Furthermore, a continuous $\oplus$ operation reflects the intuition of sequences that are generated by any kind of computable device (see §7.6 for a deeper discussion).

3. The distinction between partial and finite sequences will allow us to distinguish between automata that terminate and those that do not terminate but go on producing internal transitions.

6.3 Sequences of Lifted Elements

The definition of $(\alpha)\,\mathsf{seq}$ requires the argument type α to be of type class pcpo, as the complicated domain constructions provided by the domain package do not work for more general type classes in the class hierarchy. However, it will turn out to be crucial that elements of sequences can be handled in a total fashion, as types of class term. Therefore we introduce a new type of sequences that allows elements to be of any HOL type using the type constructor lift.

Definition 6.3.1 (Type of Sequences with Lifted Elements)
Sequences with elements lifted to a flat domain are defined as

$$(\alpha_{\mathsf{term}})\,\mathsf{sequence} = ((\alpha_{\mathsf{term}})\,\mathsf{lift})\,\mathsf{seq}$$

Furthermore, a new "cons"-operator for elements of type class term is introduced:

$$\begin{array}{lcl} \mathsf{Cons} & :: & \alpha_{\mathsf{term}} \to (\alpha_{\mathsf{term}})\,\mathsf{sequence} \to_c (\alpha_{\mathsf{term}})\,\mathsf{sequence} \\ \mathsf{Cons} & \equiv & \lambda a.\, \Lambda s.\, (\mathsf{Def}\ a) \,\#\, s \end{array}$$

We write $x\,\hat{}\,xs$ instead of $\mathsf{Cons}\ x\ \text{`}xs$. □

From now on the default type class is assumed to be term. The following syntactic sugar is provided for the different flavors of sequences:

	syntax by definition	abbreviation
finite total sequences	$a_1\hat{}\ldots\hat{}a_n\hat{}\mathsf{nil}$	$[a_1, \ldots, a_n!]$
finite partial sequences	$a_1\hat{}\ldots\hat{}a_n\hat{}\bot$	$[a_1, \ldots, a_n?]$

Recursive functions on $(\alpha)\,\mathsf{sequence}$ gain from this integration of HOL and LCF types, as follows.

Definition 6.3.2 (Recursive Functions)
The standard operations are now defined with total functions or predicates handling elements of type class term.

$$\begin{array}{lcl} \mathsf{Map} & :: & (\alpha \to \beta) \to (\alpha)\,\mathsf{sequence} \to_c (\beta)\,\mathsf{sequence} \\ \mathsf{Filter} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to_c (\alpha)\,\mathsf{sequence} \\ \mathsf{Forall} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to \mathsf{bool} \\ \mathsf{Takewhile}, \mathsf{Dropwhile} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to_c (\alpha)\,\mathsf{sequence} \end{array}$$

This is achieved by lifting them to the corresponding partial functions or predicates.

$$\begin{array}{lcl} \mathsf{Map}\, f & \equiv & \mathsf{smap}\ \text{`}(\mathsf{lift\text{-}fun}_2\, f) \\ \mathsf{Filter}\ P & \equiv & \mathsf{sfilter}\ \text{`}(\mathsf{lift\text{-}fun}_2\ P) \\ \mathsf{Forall}\ P & \equiv & \mathsf{sforall}\ (\mathsf{lift\text{-}fun}_2\ P) \\ \mathsf{Takewhile}\ P & \equiv & \mathsf{stakewhile}\ \text{`}(\mathsf{lift\text{-}fun}_2\ P) \\ \mathsf{Dropwhile}\ P & \equiv & \mathsf{sdropwhile}\ \text{`}(\mathsf{lift\text{-}fun}_2\ P) \end{array}$$

□

Note that for the functions $\oplus$, sflatten, and slast no arguments have to be lifted. Therefore, Flatten and Last equal their previous definition, just the parameter types have been specialized from α_{pcpo} to $(\alpha_{\mathsf{term}})\,\mathsf{lift}$ (see Appendix A).

Lemma 6.3.3 (Recursive Characterizations)
The recursive equations simplify as follows:

$$
\begin{array}{lcl}
\mathsf{Map}\, f \text{`}\bot & = & \bot \\
\mathsf{Map}\, f \text{`}\mathsf{nil} & = & \mathsf{nil} \\
\mathsf{Map}\, f \text{`}(a\hat{\ }s) & = & f(a)\hat{\ }\mathsf{Map}\, f \text{`}s \\
\mathsf{Filter}\ P \text{`}\bot & = & \bot \\
\mathsf{Filter}\ P \text{`}\mathsf{nil} & = & \mathsf{nil} \\
\mathsf{Filter}\ P \text{`}(a\hat{\ }s) & = & \textbf{if}\ P(a)\ \textbf{then}\ a\hat{\ }\mathsf{Filter}\ P \text{`}s \\
 & & \qquad\qquad \textbf{else}\ \mathsf{Filter}\ P \text{`}s \\
\mathsf{Forall}\ P \text{`}\bot & & \\
\mathsf{Forall}\ P \text{`}\mathsf{nil} & & \\
\mathsf{Forall}\ P \text{`}(a\hat{\ }s) & = & P(a)\ \wedge\ \mathsf{Forall}\ P \text{`}s
\end{array}
$$

Proof.
The proofs follow a common scheme and are based on the following properties about Def, lift-fun$_2$, and the truth values connectives:

$$
\begin{array}{rlcll}
 & \mathsf{Def}(a) \neq \bot & & & (\mathit{Lemma}\ 1) \\
 & \mathsf{lift\text{-}fun_2}\, f \text{`}(\mathsf{Def}\ a) & = & \mathsf{Def}\ (f\ a) & (\mathit{Lemma}\ 2) \\
a \neq \bot \Rightarrow & (a\ \mathsf{andalso}\ y \neq \mathsf{FF}) & = & (a \neq \mathsf{FF} \wedge y \neq FF) & (\mathit{Lemma}\ 3) \\
 & (\mathsf{Def}(a) \neq FF) & = & a & (\mathit{Lemma}\ 4) \\
 & \textbf{If}\ (\mathsf{Def}\ b)\ \textbf{then}\ P\ \textbf{else}\ Q & = & \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q & (\mathit{Lemma}\ 5)
\end{array}
$$

We only give the proof for the last Forall equation; the other rules follow analogously.

$$
\begin{array}{ll}
 & \mathsf{Forall}\ P \text{`}(a\hat{\ }s) \\
= & \quad \{\mathit{Def.\ 6.3.2\ (}\mathsf{Forall}\mathit{),\ Def.\ 6.2.6\ (}\mathsf{sforall}\mathit{),\ Def.\ 6.3.1\ (}\mathsf{Cons}\mathit{)}\} \\
 & \mathsf{sforall_c} \text{`}(\mathsf{lift\text{-}fun_2}\ P) \text{`}(\mathsf{Def}(a) \# s) \neq \mathsf{FF} \\
= & \quad \{\mathit{Def.\ 6.2.6\ (}\mathsf{sforall_c}\ \mathit{equations),\ Lemmas\ 1\ and\ 2}\} \\
 & ((\mathsf{Def}\ (P\ a))\ \ \mathsf{andalso}\ \ \mathsf{sforall_c} \text{`}(\mathsf{lift\text{-}fun_2}\ P) \text{`}s) \neq \mathsf{FF} \\
= & \quad \{\mathit{Lemmas\ 3\ and\ 1}\} \\
 & (\mathsf{Def}\ (P\ a)) \neq \mathsf{FF}\ \wedge\ \mathsf{sforall_c} \text{`}(\mathsf{lift\text{-}fun_2}\ P) \text{`}s \neq \mathsf{FF} \\
= & \quad \{\mathit{Lemma\ 4,\ Def.\ 6.3.2\ (}\mathsf{Forall}\mathit{),\ Def.\ 6.2.6\ (}\mathsf{sforall}\mathit{)}\} \\
 & P(a)\ \wedge\ \mathsf{Forall}\ P \text{`}s
\end{array}
$$

□

These examples demonstrate the general advantages of sequences whose argument domains are lifted HOL types. Note that these advantages are due to the lifting of types and not to the flatness of the argument domain.

- Elements of sequences that do not need support for infinity or undefinedness can be handled in the simpler logic HOL — see e.g. the total filtering predicate P — and lifted to domains as late as possible. Therefore, large theories and proof libraries about HOL types, as e.g. lists or natural numbers, can be reused.

- Not only *elements* of sequences but also the *operations* on the sequences themselves profit from pushing as much as possible into HOL. For example, the last equation for Forall uses the boolean operator $\wedge$ instead of its three-valued counterpart andalso. Similarly, Filter uses the handy **if** b **then** e_1 **else** e_2 instead of the partial **If** tr **then** e_1 **else** e_2. Therefore, proof tools tailored for two-valued predicate logic, like the built-in tableaux calculus or the simplifier, can be used efficiently. In our experience this represents the most important advantage, as it significantly increases the degree of proof automation.
  On the contrary, an analogous tableaux calculus for a three-valued logic would need a completely new and different design (see for example [AF97]).

- Because of the specific definition of the "cons"-operator as $x\hat{\ }xs = (\mathsf{Def}\ x) \# xs$, which involves the Def tag enjoying the property $(\mathsf{Def}\ x) \neq \bot$, it is ensured that all sequence elements are defined. Therefore the nasty assumption $x \neq \bot$ previously needed for every recursive equation involving $x \# xs$ can be omitted for the corresponding equations involving $x\hat{\ }xs$. This saves reasoning about a lot of unnecessary $\bot$ cases.

Of course, it is as well possible to define recursive functions directly on (α)sequence instead of lifting a corresponding function from (α) seq. As an example see the Zip function in Appendix A.

Frequently, we may want to perform recursion over the pcpo type of sequences with additional arguments. In our setting, these arguments should possibly be of class term. Indeed, this is possible, but the order of the recursive arguments is essential: the pcpo argument has preferably to come first[4]. However, this contradicts our interface philosophy between HOL and LCF, which does not permit terms of the shape f ‘$x\ y$. Therefore, continuity is not proved by the interface tactic presented in §5.2. Fortunately, for this special purpose the continuity lemma

$$\frac{\mathsf{cont}(f) \quad \mathsf{cont}(g)}{\mathsf{cont}(\lambda x.\ (f\ x)\ `(g\ x)\ c)}\ (cont_3)$$

which is easily proved, is sufficient and — more importantly — offers enough syntactic structure to guide the automatic continuity tactic efficiently. Therefore, continuity can further on be handled automatically.

[4]This is due to a sophisticated reason which deals with the automated proof of continuity.

We demonstrate recursion with an additional argument of some term type by a frequently used example:

Definition 6.3.4
The $\mathsf{Forall\text{-}Trans}$ predicate checks if all consecutive elements of a sequence are in a given relation R, provided that the first element is related to a given initial value x.

$$\begin{array}{lcl}\mathsf{Forall\text{-}Trans} & :: & (\alpha \times \alpha)\,\mathsf{set} \rightarrow \alpha \rightarrow (\alpha)\,\mathsf{sequence} \rightarrow \mathsf{bool} \\ \mathsf{Forall\text{-}Trans}\ R\ a\ s & \equiv & \mathsf{Forall\text{-}Trans_c}\ R\ `s\ a \neq \mathsf{FF}\end{array}$$

It is realized by a computable function, the continuous predicate

$$\mathsf{Forall\text{-}Trans_c}\ \ ::\ \ (\alpha \times \alpha)\,\mathsf{set} \rightarrow (\alpha)\,\mathsf{sequence} \rightarrow_c \alpha \rightarrow \mathsf{tr}$$

which is defined as a fixpoint. The following characterizing equations follow immediately when using the extended continuity tactic.

$$\begin{array}{lcl}\mathsf{Forall\text{-}Trans_c}\ R\ `\bot\ a & = & \bot \\ \mathsf{Forall\text{-}Trans_c}\ R\ `\mathsf{nil}\ a & = & \mathsf{TT} \\ \mathsf{Forall\text{-}Trans_c}\ R\ `(b\hat{\ }s)\ a & = & (\mathsf{Def}\ (a,b) \in R)\ \ \mathsf{andalso}\ \ \mathsf{Forall\text{-}Trans_c}\ R\ `s\ b\end{array}$$

□

Corollary 6.3.5
The recursive equations for $\mathsf{Forall\text{-}Trans}$ follow from those for $\mathsf{Forall\text{-}Trans_c}$:

$$\begin{array}{lcl}\mathsf{Forall\text{-}Trans}\ R\ a\ \bot & & \\ \mathsf{Forall\text{-}Trans}\ R\ a\ \mathsf{nil} & & \\ \mathsf{Forall\text{-}Trans}\ R\ a\ (b\hat{\ }s) & = & (a,b) \in R\ \wedge\ \mathsf{Forall\text{-}Trans}\ R\ s\ b\end{array}$$

The following remark concerning the definition of sequence predicates may be useful. Predicates like $\mathsf{Forall\text{-}Trans}$ cannot be defined directly as a fixpoint, as then the body of the fixpoint would not be continuous. Rather an auxiliary predicate like $\mathsf{Forall\text{-}Trans_c}$ has always to be employed. This continuous predicate yields truth values $\bot, \mathsf{TT}$, and FF, which can then be mapped in the desired manner to the boolean values True and False.

Therefore, all sequence predicates are defined via the common scheme $f\ `s \neq FF$ or $f\ `s \neq TT$. Note that this specific form relaxes admissibility problems considerably: as $s \neq FF$ and $s \neq TT$ are chain-finite predicates, the extended admissibility check in §5.1.2 discharges every predicate of this form automatically, if $f\ `s$ is continuous. Therefore, sequence predicates may occur either in positive or negated form without causing any admissibility problems.

6.4 Proof Principles and Examples

In this section we present the proof rules that are available for sequences of lifted elements. We illustrate them by a simple example and raise the question of selecting the most appropriate proof rule in a given context. This question is then dealt with in detail in the two subsequent sections.

Apart from the finite induction rule (*fin-induct*), which has been derived by the inductive definition of Finite, all proof principles are essentially proved by the HOLCF domain package. They merely had to be adapted to deal with sequences of lifted elements, i.e. of type $(\alpha)\,\mathsf{sequence}$ instead $(\alpha)\,\mathsf{seq}$.

Theorem 6.4.1 (Proof Principles)
The standard proof principle is *structural induction*:

$$\frac{\mathsf{adm}(P) \quad P(\bot) \quad P(\mathsf{nil}) \quad \forall a\ s.\ P(s) \Rightarrow P(a\hat{\ }s)}{P(t)}\ (\mathit{induct})$$

Additionally, there is the analogous rule for the finite case

$$\frac{P(\mathsf{nil}) \quad \forall a\ s.\ P(s) \wedge \mathsf{Finite}(s) \Rightarrow P(a\hat{\ }s)}{\mathsf{Finite}(t) \Rightarrow P(t)}\ (\mathit{fin\text{-}induct})$$

and the non-infinite case:

$$\frac{P(\mathsf{nil}) \quad P(\bot) \quad \forall a\ s.\ P(s) \wedge \neg\mathsf{Infinite}(s) \Rightarrow P(a\hat{\ }s)}{\neg\mathsf{Infinite}(t) \Rightarrow P(t)}\ (\mathit{noninf\text{-}induct})$$

Furthermore, the *take lemma*

$$\frac{\forall n.\ \mathsf{take}\ n\ `s = \mathsf{take}\ n\ `t}{s = t}\ (\mathit{take\text{-}lemma})$$

and the *bisimulation* rule, which follows easily from the take lemma, are available.

$$\frac{\mathsf{bisim}(R) \quad (s,t) \in R}{s = t}\ (\mathit{bisimulation})$$

Here, the predicate bisim expressing bisimularity is defined as follows:

$$\begin{array}{lcl}
\mathsf{bisim} & :: & ((\alpha)\,\mathsf{sequence} \times (\alpha)\,\mathsf{sequence})\,\mathsf{set} \to \mathsf{bool} \\
\mathsf{bisim}(R) & \equiv & \forall s\ t.\ (s,t) \in R \Rightarrow \\
& & \quad (s = \bot \Rightarrow t = \bot) \wedge \\
& & \quad (s = \mathsf{nil} \Rightarrow t = \mathsf{nil}) \wedge \\
& & \quad (\exists s\ s'.\ s = a\hat{\ }s' \Rightarrow \exists b\ t'.\ t = b\hat{\ }t' \wedge (s',t') \in R \wedge a = b)
\end{array}$$

Take lemma and bisimulation hold as well when replacing everywhere $=$ by $\sqsubseteq$. □

We demonstrate by a simple example the use of the two most important proof principles, structural induction and the take lemma.

Example 6.4.2 (Example for Structural Induction)
Let us prove the simple property

$$\mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `s) \quad = \quad \mathsf{Map}\ (f \circ g)\ `s$$

by structural induction. We get the following four proof obligations:

$$\begin{aligned}
\mathcal{C}_1: &\quad \mathsf{adm}(\lambda s.\, \mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `s) = \mathsf{Map}\ (f \circ g)\ `s) \\
\mathcal{C}_2: &\quad \mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `\bot) = \mathsf{Map}\ (f \circ g)\ `\bot \\
\mathcal{C}_3: &\quad \mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `\mathsf{nil}) = \mathsf{Map}\ (f \circ g)\ `\mathsf{nil} \\
\mathcal{C}_4: &\quad \mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `s) = \mathsf{Map}\ (f \circ g)\ `s \\
&\quad \Rightarrow \mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `(a\hat{\ }s) = \mathsf{Map}\ (f \circ g)\ `(a\hat{\ }s)
\end{aligned}$$

$\mathcal{C}_1$ is discharged automatically, as all occurring functions are continuous. $\mathcal{C}_2$ and $\mathcal{C}_3$ are trivially solved by rewriting with the equations for Map. Finally, $\mathcal{C}_4$ is true because of the following calculations.

$$\begin{aligned}
& \mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `(a\hat{\ }s) \\
= &\quad \{\textit{Equations for}\ \mathsf{Map}\} \\
& f(g(a))\hat{\ }\mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `s) \\
= &\quad \{\textit{Induction hypothesis}\} \\
& f(g(a))\hat{\ }\mathsf{Map}\ (f \circ g)\ `s \\
= &\quad \{\textit{Equations for}\ \mathsf{Map}\} \\
& \mathsf{Map}\ (f \circ g)\ `(a\hat{\ }s)
\end{aligned}$$

In Isabelle the entire proof is performed automatically by a tailored induction tactic that generates the four proof obligations and discharges them via rewriting. □

Example 6.4.3 (Example for the Take Lemma)
Let us prove the same property as above using the take lemma[5]. After applying it we strengthen the result by adding an explicit universal quantification and get:

$$\mathcal{C}_1: \quad \forall s.\, \mathsf{take}\ n\ `(\mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `s)) = \mathsf{take}\ n\ `(\mathsf{Map}\ (f \circ g)\ `s)$$

Now we use induction on n, where the base case is trivial, as $\mathsf{take}\ 0\ `s = \bot$ for all s. Assuming $\mathcal{C}_1$ as induction hypothesis, it remains to show:

$$\mathcal{C}_2: \quad \forall s.\, \mathsf{take}\ (\mathsf{Suc}\ n)\ `(\mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `s)) = \mathsf{take}\ (\mathsf{Suc}\ n)\ `(\mathsf{Map}\ (f \circ g)\ `s)$$

[5]The proof using the bisimulation rule is similar, see Thm. 6.5.1

Now we make a case distinction on s, where the cases $s = \bot$ and $s = \mathsf{nil}$ are trivial, as $\mathsf{take}\ n\ \bot = \bot$ and $\mathsf{take}\ (\mathsf{Suc}\ n)\ \mathsf{nil} = \mathsf{nil}$. Therefore, assume $s = a\hat{\ }s_1$. Then, the desired result is obtained by the following simple calculations:

$$
\begin{array}{ll}
& \mathsf{take}\ (\mathsf{Suc}\ n)\ `(\mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `(a\hat{\ }s_1)) \\
= & \quad \{\textit{Equations for}\ \mathsf{Map}\} \\
& \mathsf{take}\ (\mathsf{Suc}\ n)\ `(f(g(a))\hat{\ }\mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `s_1)) \\
= & \quad \{\textit{Equations for}\ \mathsf{take}\} \\
& f(g(a))\hat{\ }\mathsf{take}\ n\ `(\mathsf{Map}\, f\ `(\mathsf{Map}\ g\ `s_1)) \\
= & \quad \{\textit{Induction Hypothesis with}\ s := s_1\} \\
& f(g(a))\hat{\ }\mathsf{take}\ n\ `(\mathsf{Map}\ (f \circ g)\ `s_1) \\
= & \quad \{\textit{Equations for}\ \mathsf{take}\} \\
& \mathsf{take}\ (\mathsf{Suc}\ n)\ `(f(g(a))\hat{\ }\mathsf{Map}\ (f \circ g)\ `s_1) \\
= & \quad \{\textit{Equations for}\ \mathsf{Map}\} \\
& \mathsf{take}\ n\ `(\mathsf{Map}\ (f \circ g)\ `(a\hat{\ }s))
\end{array}
$$

Note that the universal quantification in $\mathcal{C}_1$ is important, as the induction hypothesis is not used for the n-th prefix of $a\hat{\ }s$, i.e. the subsequence with indexes 1 to n, but for the subsequence s, with indexes 2 to $n+1$. $\Box$

The two proofs are surprisingly similar. However, there are technical differences. In the first case induction is done directly on sequences, whereas in the second case it is done on the length of the sequence. In addition, in the first case induction deals with the input sequence, whereas in the second case it deals with the output sequence. This is the reason, why in the take lemma proof an additional rewriting step with the "cons"-rule for take is needed. Intuitively speaking, it pushes the added input element a to the resulting output sequence such that the induction hypothesis concerning the output length can be applied. Therefore, the take lemma proof is a bit more circumstantial. On the other hand, it does not require admissibility.

Therefore, we face the following methodological question: which of the proof principles should be preferred? Inductive proof principles (finite, non-infinite, general structural induction) or coinductive proof principles (take lemma, bisimulation)?

Actually, the great majority of the basic sequence lemmas are very conveniently proved by structural induction. We proved about 170 theorems, mostly by a single invocation of the induction tactic. Therefore, at first sight it seems to be superfluous to use coinductive proof principles in our setting. In fact, up to our knowledge there is no convincing argument for their use in the literature dealing with coinduction in a PCF setting [Pit94, BW88, Gor95], i.e. a context where functions are defined recursively. The only motivation given in [BW88] is the example of the *iterate* function which does not contain an explicit sequence argument to induct over. This seems to be a rather artificial case. However, we have identified two significant reasons for employing coinduction instead of induction:

1. Sometimes coinduction is preferable if induction breaks down because the admissibility check fails. The application of coinductive proof principles (in such cases) is, however, not always as simple as in the example given above. The reason is that recursive definitions match inductive proof principles, whereas corecursive definitions match coinductive proof principles. Therefore, we have to provide further infrastructure, which allows to apply coinductive proof methods for recursive definitions as well.

2. In addition, functions occurring in the proof goal may be more naturally defined corecursively. Therefore, coinductive proof principles are more appropriate. However, as domain theory does not support a corecursive definition style, we have to derive a corecursive characterization of the relevant recursively defined functions.

We deal with these two subjects in the next two sections §6.5 and §6.6, respectively.

6.5 Infrastructure and Methodology for Coinduction

As mentioned already, structural induction is a very convenient proof principle in our setting, as it is well amenable for automation. However, sometimes the admissibility check may fail, as it does not handle negation and existential quantification in general. As an example see Lemma 6.5.2(1). In our experience it is in almost all such cases possible to reformulate the goal to satisfy the admissibility criterion or to get by with the finite induction rule. In the remaining cases, however, it seems not to be advisable to prove admissibility via its definition, as this then often becomes the hardest part of the entire proof. Instead, one should switch to coinductive proof principles, i.e. the take lemma or the bisimulation rule.

In such cases, it is not possible to apply the take lemma or the bisimulation rule directly, as they allow to treat merely sequence *equalities*. Equalities, however, are always automatically proved to be admissible, as we deal only with continuous functions. Therefore, we have to improve the coinductive proof rules to cover negation or existential quantification as well, if they ought to be of any use for tackling the problems mentioned above. We will accomplish this, at least partially, later on.

First, we restrict our attention to sequence equalities and recast Examples 6.4.2 – 6.4.3 in an abstract way. Specifically, we show $f(s) = g(s)$ for sequences s using either structural induction, take lemma, or bisimulation. To make the points clearer we hereby restrict us to functions f and g which obey a specific recursion scheme, namely one that corresponds to primitive recursion for finite lists. We pursue three goals:

- Demonstrate the use of the three proof principles in a schematic way.
- Compare the power of the respective proof principles.
- Elaborate the problems occurring w.r.t. coinduction in a setting featuring recursion.

The following theorem deals with these goals for all three proof principles collectively.

Theorem 6.5.1 (Comparison of Proof Principles I)
Let f and g be continuous functions of type $(\alpha)\,\mathsf{sequence} \to_c (\alpha)\,\mathsf{sequence}$. Furthermore, assume that for every $X \in \{f, g\}$ there is an h_X such that:

$$X(a\hat{\ }s) = h_X(a) \oplus X(s) \wedge \mathsf{Finite}(h_X(a))$$

Then, $f(s) = g(s)$ is provable using either structural induction, take lemma, or bisimulation, provided the following respective assumptions are fulfilled:

1) $f(\bot) = g(\bot) \wedge f(\mathsf{nil}) = g(\mathsf{nil})$

2) $a)$ Structural Induction:
$h_f = h_g$

$b)$ Take Lemma:
$h_f = h_g \wedge$
$\forall a.\ h_f(a) \neq \mathsf{nil} \wedge h_g(a) \neq \mathsf{nil}$

$c)$ Bisimulation:
$h_f = h_g \wedge$
$\forall a.\ \exists e.\ h_f(a) = [e!] \wedge h_g(a) = [e!]$

Proof.
We apply structural induction, the take lemma and the bisimulation rule subsequently.

- **Structural Induction:** The necessary admissibility requirement follows from the continuity of f and g, the base cases follow from 1). Using $2)a)$ and the induction hypothesis $f(s) = g(s)$ the inductive case is proved as follows:

$$f(a\hat{\ }s) = h_f(a) \oplus f(s) = h_g(a) \oplus g(s) = g(a\hat{\ }s)$$

- **Take Lemma:** After applying the take lemma and strengthening the result with an explicit quantification we get:

$$\mathcal{C}_1\colon\quad \forall s.\ \mathsf{take}\ n\ \text{`}(f\ s) = \mathsf{take}\ n\ \text{`}(g\ s)$$

In contrast to Example 6.4.3 we apply the more general induction rule

$$\forall m.\ (m < n \Rightarrow P(m) \Rightarrow P(n)) \Rightarrow P(n)$$

on natural numbers, and therefore have to prove $\mathcal{C}_1$ under the assumption $\mathcal{A}_1$:

$$\mathcal{A}_1:\quad \forall m.\, m < n \Rightarrow \forall s.\, \mathsf{take}\ m\ \text{‘}(f\ s) = \mathsf{take}\ m\ \text{‘}(g\ s)$$

After eliminating the universal quantor of $\mathcal{C}_1$, we make a case distinction on s. The base cases $s = \bot$ and $s = \mathsf{nil}$ follow immediately from 1). For the "cons"-case $s = a\hat{\ }s_1$ it remains to show:

$$\mathcal{C}_2:\quad \forall s.\, \mathsf{take}\ n\ \text{‘}(f\ (a\hat{\ }s_1)) = \mathsf{take}\ n\ \text{‘}(g\ (a\hat{\ }s_1)) \quad (*)$$

First, we derive the following lemma:

$$\frac{s_1 \neq \bot \wedge s_1 \neq \mathsf{nil} \quad s_1 = s_2 \quad \forall m.\, m < n \Rightarrow \mathsf{take}\ m\ \text{‘}t_1 = \mathsf{take}\ m\ \text{‘}t_2}{\mathsf{take}\ n\ \text{‘}(s_1 \oplus t_1) = \mathsf{take}\ n\ \text{‘}(s_2 \oplus t_2)}\ (\textit{take-reduce})$$

Then, we get the desired result as follows:

$$\begin{aligned}
\mathsf{take}\ n\ \text{‘}(f\ (a\hat{\ }s_1)) &= \mathsf{take}\ n\ \text{‘}(h_f(a) \oplus f(s_1)) \\
&= \mathsf{take}\ n\ \text{‘}(h_g(a) \oplus g(s_1)) \\
&= \mathsf{take}\ n\ \text{‘}(g\ (a\hat{\ }s_1))
\end{aligned}$$

Here, we applied lemma (*take-reduce*) in the second rewriting step, which requires 2)b), the finiteness of the h_X, and the induction hypothesis $\mathcal{A}_1$. Note that the non-emptiness of the h_X in 2)b) is required in lemma (*take-reduce*) as otherwise the reduction to a truly smaller m would not be guaranteed.

- **Bisimulation:** For the bisimulation relation we take the "canonical" choice:

$$R \quad \equiv \quad \{(x, y) \mid \exists s.\, x = f(s) \wedge y = g(s)\}$$

Therefore, applying the bisimulation rule, the emerging subgoal $(f(s), g(s)) \in R$ holds trivially. It remains to show that R is a bisimulation, i.e. to show:

$$\begin{aligned}
&\mathcal{C}_1:\ f(s) = \bot \Rightarrow g(s) = \bot \\
&\mathcal{C}_2:\ f(s) = \mathsf{nil} \Rightarrow g(s) = \mathsf{nil} \\
&\mathcal{C}_3:\ f(s) = b\hat{\ }t_1 \Rightarrow \exists c\ t_2.\, g(s) = c\hat{\ }t_2 \wedge b = c \wedge \exists s'.\, t_1 = f(s') \wedge t_2 = g(s')
\end{aligned}$$

We proceed by case distinction on s. For the cases $s = \bot$ and $s = \mathsf{nil}$ $\mathcal{C}_1$ and $\mathcal{C}_2$ are trivially true because of 1). Similarly, $\mathcal{C}_3$ is true because of 1) and $b\hat{\ }t_1 \neq \mathsf{nil} \wedge b\hat{\ }t_1 \neq \bot$. Therefore, assume $s = a\hat{\ }s_1$. Now, $\mathcal{C}_1$ is true, as $h_f(a)$ is finite and therefore $f(a\hat{\ }s_1) = h_f(a) \oplus f(s_1) \neq \bot$, which yields a contradiction in the assumption of $\mathcal{C}_1$. A similar contradiction is generated for $\mathcal{C}_2$ by $f(a\hat{\ }s_1) = h_f(a) \oplus f(s_1) \neq \mathsf{nil}$, which holds because of $h_f(a) \neq \mathsf{nil}$. It remains to show $\mathcal{C}_3$ for $s = a\hat{\ }s_1$. From the assumption $f(a\hat{\ }s_1) = b\hat{\ }t_1$ of $\mathcal{C}_3$ we get with $f(a\hat{\ }s_1) = h_f(a) \oplus f(s_1)$ and 2)c) the existence of an

e such that $e\hat{\ }f(s_1) = b\hat{\ }t_1$. Therefore, $b = e$ and $t_1 = f(s_1)$. Therefore, it remains to show:

$$\mathcal{C}_4\colon\quad \exists c\ t_2.\ g(a\hat{\ }s_1) = c\hat{\ }t_2 \wedge e = c \wedge \exists s'.\ f(s_1) = f(s') \wedge t_2 = g(s')$$

Instantiating c with e, t_2 with $g(s_1)$, and s' with s_1 we get the result immediately using the recursion scheme for g. □

Therefore, comparing the proof principles it is obvious that proofs by structural induction are the simplest in this context. A further interesting result is that the use of the bisimulation rule is stronger restricted than that of the take lemma. Whereas bisimulation requires that every input element produces exactly one output element ($\exists e.\ h_X(a) = [e]$), the take lemma admits also more than one output element ($h_X(a) \neq \mathsf{nil}$).

Both restrictions, however, are rather severe and are not fulfilled for many functions f and g. Take for example the Filter function w.r.t. a predicate P. If an input element a does not satisfy P, it will not produce any output. Rather we have $\neg P(a) \Rightarrow \mathsf{Filter}\ P\ `(a\hat{\ }s) = \mathsf{Filter}\ P\ `s$. Therefore, it is not possible to apply the take lemma or bisimulation in the fashion proposed in Thm. 6.5.1 and Examples 6.4.2 – 6.4.3. The problem is that coinduction fits well with corecursive definitions, but not with recursive ones. This is motivated as follows. Coinduction essentially means to reason about the *output* elements produced subsequently by f and g. Corecursion matches this proof style conveniently as it determines for every *output* element the corresponding input elements, if one exists. In our context, however, f and g are given recursively. This means, that for every *input* element the corresponding output elements are determined, if one exists. Therefore, when applying coinduction, a neat correspondence between input and output is of advantage. Such a neat correspondence is given if every input produces exactly one output (as assumed in Examples 6.4.2 – 6.4.3) or at least one output (as assumed in Thm. 6.5.1). If this is not the case, as for Filter, we have to refine the proof method proposed in Thm. 6.5.1, i.e. to provide a more powerful proof infrastructure.

In the sequel we will accomplish this by requiring the user to supply a predicate Q, defined on the elements of the input sequence s, that allows to treat the cases of empty outputs separately. Concretely, Q should be chosen in such a way that it characterizes the first element in s that produces a nonempty output by both f and g. Basically, the predicate Q incorporates the intuition which would have been necessary if f and g were defined corecursively. In fact, we will see in §6.6 that Q can be found automatically if f and g are characterized corecursively.

Before presenting the refined proof method we first need the following lemma.

Lemma 6.5.2 (Subdivision of Sequences by Predicates)
If a is the first element produced by filtering a sequence s w.r.t. a predicate P, then s can be subdivided into a finite prefix not satisfying P, followed by a (which satisfies P) and

a suffix. Moreover, if not all elements in s satisfy P, then there is an element a with the aforementioned property in negated form.

$$
\begin{aligned}
&(a\hat{\ }xs) \sqsubseteq \mathsf{Filter}\ P\ \text{`}s \Rightarrow && (1)\\
&\quad s = ((\mathsf{Takewhile}\ (\lambda x.\ \neg P(x))\ \text{`}s)\ \oplus\ a\hat{\ }\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\ (\lambda x.\ \neg P(x))\ \text{`}s)) \wedge\\
&\quad \mathsf{Finite}\ (\mathsf{Takewhile}\ (\lambda x.\ \neg P(x))\ \text{`}s \wedge P(a)\\
&\neg\mathsf{Forall}\ P\ s \Rightarrow \exists a\ bs\ rs.\ s = (bs\ \oplus\ a\hat{\ }rs) \wedge \mathsf{Finite}(bs) \wedge \mathsf{Forall}\ P\ bs \wedge \neg P(a) && (2)
\end{aligned}
$$

Proof.

(1) It is not reasonable to apply structural induction immediately, as the admissibility check would fail: $\mathsf{adm}(\lambda s.\ (a\hat{\ }xs) \not\sqsubseteq \mathsf{Filter}\ P\ \text{`}s)$ cannot be discharged automatically. However, we get by with a reformulation of the goal. Using the implication

$$(a\hat{\ }xs) \sqsubseteq \mathsf{Filter}\ P\ \text{`}s \Rightarrow \mathsf{HD}\ \text{`}(\mathsf{Filter}\ P\ \text{`}s) = \mathsf{Def}(a)$$

which holds because of the monotonicity of HD, we strengthen (1) to the following proposition, which can easily be proved using structural induction.

$$
\begin{aligned}
&\mathsf{HD}\ \text{`}(\mathsf{Filter}\ P\ \text{`}s) = \mathsf{Def}(a) \Rightarrow && (1')\\
&\quad s = ((\mathsf{Takewhile}\ (\lambda x.\ \neg P(x))\ \text{`}s)\ \oplus\ a\hat{\ }\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\ (\lambda x.\ \neg P(x))\ \text{`}s)) \wedge\\
&\quad \mathsf{Finite}\ (\mathsf{Takewhile}\ (\lambda x.\ \neg P(x))\ \text{`}s \wedge P(a)
\end{aligned}
$$

Note that the admissibility obligation can be discharged automatically (but only by the extended admissibility check), as $\mathsf{Def}(a)$ is of chain-finite type.

(2) Similar to the previous proof, the goal can be reduced to (1′) using the implication

$$\neg\mathsf{Forall}\ P\ s \Rightarrow (\exists a.\ \mathsf{HD}\ \text{`}(\mathsf{Filter}\ (\lambda a.\ \neg P(a))\ \text{`}s) = \mathsf{Def}(a))$$

which is provable by structural induction. The admissibility of this implication is again automatically proved by the extended check, as $\mathsf{Def}(a)$ is chain-finite. □

Now we are able to refine the proof of $f(s) = g(s)$ using either bisimulation or the take lemma by requiring a specific predicate Q. The following theorem shows that therefore the restriction $\forall a.\ h_X(a) \neq \mathsf{nil}$ can be dropped for the take lemma proof. The bisimulation proof, however, still needs the former restriction $\forall a. \exists e.\ h_X(a) = [e]$, and therefore appears to be less powerful.

Theorem 6.5.3 (Comparison of Proof Principles II)
Let f and g be continuous functions of type $(\alpha)\,\mathsf{sequence} \rightarrow_c (\alpha)\,\mathsf{sequence}$. Furthermore, assume a predicate $Q :: \alpha \rightarrow \mathsf{bool}$ such that for every $X \in \{f, g\}$ there is a h_X which

satisfies the following equations:

$$\begin{array}{rcll} \neg Q(a) & \Rightarrow & X(a\hat{\ }s) = h_X(a) \oplus X(s) \wedge \mathsf{Finite}(h_X(a)) \wedge h_X(a) \neq \mathsf{nil} & (rec_1) \\ Q(a) & \Rightarrow & X(a\hat{\ }s) = f(s) & (rec_2) \end{array}$$

Then, $f(s) = g(s)$ is provable using either the take lemma or the bisimulation rule, provided the following respective assumptions are fulfilled:

1) $\mathsf{Forall}\ Q\ s \wedge \neg\mathsf{Finite}(s) \Rightarrow f(s) = \bot \wedge g(s) = \bot$

2) $a)$ Take Lemma: $h_f = h_g$ | $b)$ Bisimulation: $h_f = h_g \wedge$ $\forall a.\, \exists e.\ h_f(a) = [e!] \wedge h_g(a) = [e!]$

Proof.
We apply the take lemma and the bisimulation rule subsequently.

- **Take Lemma:** First, we proceed analogously to the proof of Thm. 6.5.1. Therefore, we apply the take lemma, strengthen the result with an explicit quantification, and apply induction from $m < n$ to n. Therefore, we have to prove $\mathcal{C}_1$ under assumption $\mathcal{A}_1$ given as follows:

$$\begin{array}{ll} \mathcal{C}_1: & \forall s.\, \mathsf{take}\ n\ `(f\ s) = \mathsf{take}\ n\ `(g\ s) \\ \mathcal{A}_1: & \forall m.\, m < n \Rightarrow \forall s.\, \mathsf{take}\ m\ `(f\ s) = \mathsf{take}\ m\ `(g\ s) \end{array}$$

  At this point, we do not make a case distinction on s as in the proof of Thm. 6.5.1, as the first input element in s may not produce the first output element. Rather we make a case distinction on the predicate Q which characterizes according to (rec_1) and (rec_2) the first element of s that produces a nonempty output:

  - **Case** $\mathsf{Forall}\ Q\ s$**:**
    - **Case** $\mathsf{Finite}(s)$**:** From $\mathsf{Forall}\ Q\ s$, $\mathsf{Finite}(s)$, and (rec_2) follows easily by finite structural induction on s that $f(s) = \mathsf{nil}$ and $g(s) = \mathsf{nil}$. Therefore we get $f(s) = g(s)$, which implies $\mathcal{C}_1$.
    - **Case** $\neg\mathsf{Finite}(s)$**:** From 1) we get $f(s) = \bot$ and $g(s) = \bot$, which implies $\mathcal{C}_1$ similar to the previous case. Note that 1) cannot be derived in general from $\mathsf{Forall}\ Q\ s$ and $\neg\mathsf{Finite}(s)$ by structural induction, as $\mathsf{Finite}(s)$ is definitely not admissible. Therefore, 1) has to be proven by other means depending on the actual Q.
  - **Case** $\neg\mathsf{Forall}\ Q\ s$**:**
    The assumption of this case allows us to apply Lemma 6.5.2(2) which ensures the existence of values a, bs, and rs, such that

$$\mathcal{A}_2: \quad s = (bs \oplus a\hat{\ }rs) \wedge \mathsf{Finite}(bs) \wedge \mathsf{Forall}\ Q\ bs \wedge \neg Q(a)$$

Therefore, rewriting $\mathcal{C}_1$ with this equation for s it remains to show:

$$\mathcal{C}_2\text{:}\quad \mathsf{take}\ n\ \text{‘}(f\ (bs\ \oplus\ a\hat{\ }rs)) = \mathsf{take}\ n\ \text{‘}(g\ (bs\ \oplus\ a\hat{\ }rs))$$

Using the facts $\mathsf{Finite}(bs)$, $\mathsf{Forall}\ Q\ bs$, and (rec_2) it is easily shown by finite structural induction that for $X \in \{f, g\}$ holds: $X(bs\ \oplus\ x) = X(x)$. Therefore, $\mathcal{C}_2$ reduces to

$$\mathcal{C}_3\text{:}\quad \mathsf{take}\ n\ \text{‘}(f\ (a\hat{\ }rs)) = \mathsf{take}\ n\ \text{‘}(g\ (a\hat{\ }rs))$$

Now we have reached a proposition similar to $(*)$ in the proof of Thm. 6.5.1, and the argumentation is from now on analogous to there. This means that we apply lemma (*take-reduce*) in order to be able to apply the induction hypothesis, which finishes the proof.

- **Bisimulation:** First, we proceed analogously to the proof of Thm. 6.5.1. Therefore, we take the canonical bisimulation relation, apply the bisimulation rule, and get:

$$\begin{aligned}
&\mathcal{C}_1\text{:}\ \ f(s) = \bot \Rightarrow g(s) = \bot\\
&\mathcal{C}_2\text{:}\ \ f(s) = \mathsf{nil} \Rightarrow g(s) = \mathsf{nil}\\
&\mathcal{C}_3\text{:}\ \ f(s) = b\hat{\ }t_1 \Rightarrow \exists c\ t_2.\ g(s) = c\hat{\ }t_2 \wedge b = c \wedge \exists s'.\ t_1 = f(s') \wedge t_2 = g(s')
\end{aligned}$$

  Now we leave the proof of Thm. 6.5.1 and proceed similar to the take-lemma proof above.

  - **Case** $\mathsf{Forall}\ Q\ s$**:**
    - **Case** $\mathsf{Finite}(s)$**:** From $\mathsf{Forall}\ Q\ s$, $\mathsf{Finite}(s)$, and (rec_2) follows easily by finite structural induction on s that $f(s) = \mathsf{nil}$ and $g(s) = \mathsf{nil}$. Therefore, $\mathcal{C}_1$ – $\mathcal{C}_3$ are directly fulfilled.
    - **Case** $\neg\mathsf{Finite}(s)$**:** From 1) we get $f(s) = \bot$ and $g(s) = \bot$, which implies $\mathcal{C}_1$ – $\mathcal{C}_3$ similar to the previous case.
  - **Case** $\neg\mathsf{Forall}\,Q\,s$**:** The assumption of this case allows us to apply Lemma 6.5.2(2) which ensures the existence of values a, bs, and rs, such that

$$\mathcal{A}_2\text{:}\quad s = (bs\ \oplus\ a\hat{\ }rs) \wedge \mathsf{Finite}(bs) \wedge \mathsf{Forall}\ Q\ bs \wedge \neg Q(a)$$

    Therefore, rewriting $\mathcal{C}_1$ – $\mathcal{C}_3$ with this equation for s, it remains to show:

$$\begin{aligned}
&\mathcal{C}_4\text{:}\ \ f(bs\ \oplus\ a\hat{\ }rs) = \bot \Rightarrow g(bs\ \oplus\ a\hat{\ }rs) = \bot\\
&\mathcal{C}_5\text{:}\ \ f(bs\ \oplus\ a\hat{\ }rs) = \mathsf{nil} \Rightarrow g(bs\ \oplus\ a\hat{\ }rs) = \mathsf{nil}\\
&\mathcal{C}_6\text{:}\ \ f(bs\ \oplus\ a\hat{\ }rs) = b\hat{\ }t_1\\
&\qquad \Rightarrow \exists c\ t_2.\ g(bs\ \oplus\ a\hat{\ }rs) = c\hat{\ }t_2 \wedge b = c \wedge \exists s'.\ t_1 = f(s') \wedge t_2 = g(s')
\end{aligned}$$

    Using the facts $\mathsf{Finite}(bs)$, $\mathsf{Forall}\ Q\ bs$, and (rec_2) it is easily shown by finite structural induction that for $X \in \{f, g\}$ holds: $X(bs\ \oplus\ x) = X(x)$. Therefore,

$\mathcal{C}_1 - \mathcal{C}_3$ reduce to

$$\begin{array}{ll} \mathcal{C}_4\text{:} & f(a\hat{\ }rs) = \bot \Rightarrow g(a\hat{\ }rs) = \bot \\ \mathcal{C}_5\text{:} & f(a\hat{\ }rs) = \mathsf{nil} \Rightarrow g(a\hat{\ }rs) = \mathsf{nil} \\ \mathcal{C}_6\text{:} & f(a\hat{\ }rs) = b\hat{\ }t_1 \\ & \quad \Rightarrow \exists c\ t_2.\ g(a\hat{\ }rs) = c\hat{\ }t_2 \wedge b = c \wedge \exists s'.\ t_1 = f(s') \wedge t_2 = g(s') \end{array}$$

Now we return to the proof of Thm. 6.5.1 and proceed analogously to the case $s = a\hat{\ }s_1$ described there. This finishes the proof. □

Therefore, we have compared the power of coinductive proof principles and proposed a proof methodology for them. However, the method is still restricted to sequence equalities. In order to be useful for problems for which the admissibility check fails we have to develop extensions that cover negation or existential quantification as well. Concerning existential quantification we will see below that coinduction is as useless as induction. Concerning negation, however, we will now present proof rules that allow to treat sequence equalities under an arbitrary assumption, i.e. we prove $A(s) \Rightarrow f(s) = g(s)$. This seems to be a very common and useful property whose admissibility, however, is discharged only in rare cases, as A appears in negated form. We will restrict the presentation to the take lemma, as the corresponding rule for bisimulation is analogous, but weaker.

The two subsequent rules incorporate the main proof ideas of Thm. 6.5.1 and Thm. 6.5.3, respectively. However, they are more general (even apart from the additional assumption A), as they permit arbitrary recursion. This is the reason why the application of the induction hypothesis cannot be included in the proof of the rules. Instead, we give the induction hypothesis as explicit premise in the rules and provide a lemma that allows to reduce the n-takes of the respective sequences. This lemma has to be used after applying the actual proof rule. Note that the theorems below have actually been proved in Isabelle, whereas the weaker Thm. 6.5.1 and Thm. 6.5.3 were merely described to motivate them.

Theorem 6.5.4 (Take Lemma Proof Rule I)
Sequence equalities $f(s) = g(s)$ for sequences s can be shown under an assumption $A(s)$ by the following proof rule, if f and g produce exactly one output element for every input element.

$$\frac{\begin{array}{c} A(\bot) \\ \vdots \\ f(\bot) = g(\bot) \end{array} \qquad \begin{array}{c} A(\mathsf{nil}) \\ \vdots \\ f(\mathsf{nil}) = g(\mathsf{nil}) \end{array} \qquad \begin{array}{c} \forall t.\ A(t) \Rightarrow \mathsf{take}\ n\ `(f\ t) = \mathsf{take}\ n\ `(g\ t) \\ A(a\hat{\ }s2) \\ \vdots \\ \mathsf{take}\ (\mathsf{Suc}\ n)\ `(f(a\hat{\ }s_2)) = \mathsf{take}\ (\mathsf{Suc}\ n)\ `(g(a\hat{\ }s_2)) \end{array}}{\forall s.\ A(s) \Rightarrow f(s) = g(s)}$$

Proof.
Simple generalization of the proof of Thm. 6.5.1. □

Theorem 6.5.5 (Take Lemma Proof Rule II)
Sequence equalities $f(s) = g(s)$ for sequences s can be shown in general under an assumption $A(s)$ by the following proof rule.

$$\dfrac{\begin{array}{c} \mathsf{Forall}\ Q\ s \\ \neg\mathsf{Finite}(s) \\ A(s) \\ \vdots \\ f(s) = \bot \wedge \\ g(s) = \bot \end{array} \qquad \begin{array}{c} \mathsf{Forall}\ Q\ s \\ \mathsf{Finite}(s) \\ A(s) \\ \vdots \\ f(s) = \mathsf{nil} \wedge \\ g(s) = \mathsf{nil} \end{array} \qquad \begin{array}{c} \forall t\ m.\ m < n \wedge A(t) \Rightarrow \mathsf{take}\ m\ `(f\ t) = \mathsf{take}\ m\ `(g\ t) \\ \mathsf{Forall}\ Q\ s_1 \wedge \mathsf{Finite}(s_1) \wedge \neg Q(a) \\ A(s_1 \oplus a \hat{}\ s2) \\ \vdots \\ \mathsf{take}\ n\ `(f(s_1 \oplus a \hat{}\ s_2)) = \mathsf{take}\ n\ `(g(s_1 \oplus a \hat{}\ s_2)) \end{array}}{\forall s.\ A(s) \Rightarrow f(s) = g(s)}$$

Proof.
Simple generalization of the proof of Thm. 6.5.3. □

In order to apply the induction hypothesis in these proof rules, i.e. the first assumption of the third assumption, we provide the following lemma.

Lemma 6.5.6 (Take Reduction)
Equalities of sequence prefixes of length n can be reduced to analogous equalities of length less than n if a nonempty coinciding prefix exists.

$$\dfrac{a = b \quad s_1 = s_2 \quad \forall m.\ m < n \Rightarrow \mathsf{take}\ m\ `t_1 = \mathsf{take}\ m\ `t_2}{\mathsf{take}\ n\ `(s_1 \oplus a \hat{}\ t_1) = \mathsf{take}\ n\ `(s_2 \oplus b \hat{}\ t_2)}$$

□

For an example of how these rules are used see Lemma 6.6.3. Furthermore, they will be of vital importance for the compositionality proof of I/O automata in §10.

Finally, we show why these rules cannot be extended to existential quantification. Assume we want to prove the sequence equality $\exists x.\ f(x) = g(x)$ with x being of arbitrary type. Applying the take lemma leaves us with $\exists x.\ \forall n.\ \mathsf{take}\ n\ `(f\ x) = \mathsf{take}\ n\ `(g\ x)$. In order to apply induction on n now, we first have to change the order of the quantifiers $\exists$ and $\forall$. However, the needed $\forall x.\ \exists y.\ f\ x\ y \Rightarrow \exists y.\ \forall x.\ f\ x\ y$ does not hold in general. Therefore, the take lemma cannot be applied in this context.

6.6 Corecursive Characterizations

In the previous section we presented coinductive proof support for properties of the form $A(s) \Rightarrow f(s) = g(s)$. The motivation was the failing admissibility check for induction.

In this section we deal with a further motivation for employing coinduction instead of induction: it could be the case that f and g are definable via recursion, though only in an awkward way, whereas a corecursive definition would be much more natural. Then we may derive rewrite rules from the recursive definition, which *characterize* the operators corecursively. In such a case coinduction obviously is more appropriate than induction.

However, up to our experience, it is not frequently the case that a corecursive definition is more intuitive than a recursive one. The reason may be that functions are more naturally specified by describing how they react on inputs than by determining the associated inputs to observed outputs. In any case, corecursive definitions usually — compare the experiences by Paulson [Pau97] — involve more case distinctions than recursive definitions. In particular, there seems to be no explicit example for a mandatory corecursive function in the literature [JR97, HJ97b, Pit94, BW88, Gor95]. Nevertheless, we encountered an example during the compositionality proof for I/O automata, which we will use in the following to demonstrate corecursive characterizations.

Definition 6.6.1 (Chop)
The function Chop cuts a sequence s after the last element satisfying a predicate P. It is given via a recursive definition by first filtering s w.r.t. P and then remerging every element in s not satisfying P into the resulting sequence, up to its last element.

$$\begin{array}{lcl} \mathsf{Chop} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\ \mathsf{Chop}\ P\ s & \equiv & \mathsf{Paste}\ P\ \text{`}s\ \text{`}(\mathsf{Filter}\ P\ \text{`}s) \end{array}$$

The merging process is defined by the recursive function Paste.

$$\begin{array}{lcll} \mathsf{Paste} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to_c (\alpha)\,\mathsf{sequence} \to_c (\alpha)\,\mathsf{sequence} & \\ \mathsf{Paste}\ P\ \text{`}s\ \text{`}\bot & = & \bot & (Paste_1) \\ \mathsf{Paste}\ P\ \text{`}s\ \text{`}\mathsf{nil} & = & \mathsf{nil} & (Paste_2) \\ \mathsf{Paste}\ P\ \text{`}bs\ \text{`}(a\hat{}rs) & = & \mathsf{Takewhile}\ (\lambda x.\ \neg P(x))\ \text{`}bs) & (Paste_3) \\ & & \oplus\ (a\hat{}\mathsf{Paste}\ P\ \text{`}(\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\ (\lambda x.\ \neg P(x))\ \text{`}bs)\ \text{`}rs)) & \end{array}$$

□

At first sight it seems that Chop cannot be defined as a computable function at all, as running down the sequence one never knows when the last element satisfying P will appear. However, as the definition above shows, it is possible by running down the sequence twice: first the length of the resulting sequence is determined and then every element that has been lost during this check is re-added. Nevertheless, a corecursive characterization of Chop is much more natural:

Corollary 6.6.2 (Corecursive Characterization of Chop)
The following rewrite rules characterize Chop according to the produced outputs.

$$\frac{\mathcal{A}_1\colon\ \mathsf{Forall}\ (\lambda x.\ \neg P(x))\ s \quad \mathcal{A}_2\colon\ \neg\mathsf{Finite}(s)}{\mathsf{Chop}\ P\ s = \bot}\ (Chop_1)$$

$$\frac{\mathcal{A}_1\colon \mathsf{Forall}\,(\lambda x.\,\neg P(x))\; s \quad \mathcal{A}_2\colon \mathsf{Finite}(s)}{\mathsf{Chop}\; P\; s = \mathsf{nil}}\;(Chop_2)$$

$$\frac{\mathcal{A}_1\colon P(a) \qquad \mathcal{A}_2\colon \mathsf{Forall}\,(\lambda x.\,\neg P(x))\; bs \qquad \mathcal{A}_3\colon \mathsf{Finite}(bs)}{\mathcal{C}_1\colon \mathsf{Chop}\; P\; (bs \,\oplus\, a\,\hat{}\, rs) = bs \,\oplus\, a\,\hat{}\,\mathsf{Chop}\; P\; rs}\;(Chop_3)$$

Proof.

- *Chop*$_1$: From $\mathcal{A}_1$ and $\mathcal{A}_2$ we derive $\mathsf{Filter}\; P\,`s = \bot$ using the sequence lemma

$$\frac{\mathsf{Forall}\,(\lambda x.\,\neg P(x))\; s \quad \neg\mathsf{Finite}(s)}{\mathsf{Filter}\; P\,`s = \bot}\;(*)$$

  which is proved by structural induction. Then the result follows as given below:

$$\mathsf{Chop}\; P\; s = \mathsf{Paste}\; P\,`s\,`(\mathsf{Filter}\; P\,`s) = \mathsf{Paste}\; P\,`s\,`\bot = \bot$$

- *Chop*$_2$: Analogous to the previous case. Instead of $(*)$ the following lemma is used:

$$\frac{\mathsf{Forall}\,(\lambda x.\,\neg P(x))\; s \quad \mathsf{Finite}(s)}{\mathsf{Filter}\; P\,`s = \mathsf{nil}}\;(**)$$

- *Chop*$_3$: First, we derive the following sequence lemmas by structural induction.

$$\mathsf{Filter}\; P\,`(s \,\oplus\, t) = \mathsf{Filter}\; P\,`s \,\oplus\, \mathsf{Filter}\; P\,`t \qquad (1)$$
$$\mathsf{Forall}\; P\; s \Rightarrow \mathsf{Takewhile}\; P\,`(s \,\oplus\, t) = s \,\oplus\, (\mathsf{Takewhile}\; P\,`t) \qquad (2)$$
$$\mathsf{Finite}(s) \wedge \mathsf{Forall}\; P\; s \Rightarrow \mathsf{Dropwhile}\; P\,`(s \,\oplus\, t) = \mathsf{Dropwhile}\; P\,`t \qquad (3)$$

  Then, the desired result follows by the following calculations:

$$\begin{array}{ll}
 & \mathsf{Chop}\; P\; (bs \,\oplus\, a\,\hat{}\, rs) \\
= & \quad \{\textit{Def. 6.6.1}\ (\mathsf{Chop})\} \\
 & \mathsf{Paste}\; P\,`(bs \,\oplus\, a\,\hat{}\, rs)\,`(\mathsf{Filter}\; P\,`(bs \,\oplus\, a\,\hat{}\, rs)) \\
= & \quad \{(1),\ (**),\ \mathcal{A}_1\text{–}\mathcal{A}_3,\ \textit{Equations for}\ \mathsf{Filter}\} \\
 & \mathsf{Paste}\; P\,`(bs \,\oplus\, a\,\hat{}\, rs)\,`a\,\hat{}(\mathsf{Filter}\; P\,`rs) \\
= & \quad \{\textit{Def. 6.6.1}\ (\mathsf{Paste})\} \\
 & \mathsf{Takewhile}\,(\lambda x.\,\neg P(x))\,`(bs \,\oplus\, a\,\hat{}\, rs)) \\
 & \oplus\,(a\,\hat{}\,\mathsf{Paste}\; P\,`(\mathsf{TL}\,`(\mathsf{Dropwhile}\,(\lambda x.\,\neg P(x))\,`(bs \,\oplus\, a\,\hat{}\, rs))\,`rs)) \\
= & \quad \{(2),\ \mathcal{A}_1\ \text{–}\mathcal{A}_3,\ \textit{Equations for}\ \mathsf{Takewhile},\ (3),\ \textit{Equations for}\ \mathsf{Dropwhile}\} \\
 & bs \,\oplus\, (a\,\hat{}\,\mathsf{Paste}\; P\,`(\mathsf{TL}\,`(\mathsf{Dropwhile}\,(\lambda x.\,\neg P(x))\,`rs)\,`rs)) \\
= & \quad \{\textit{Def. 6.6.1}\ (\mathsf{Chop})\} \\
 & bs \,\oplus\, a\,\hat{}\,\mathsf{Chop}\; P\; rs
\end{array}$$

$\square$

Using this new characterization of Chop we can now conveniently prove properties about it using the take lemma. The following lemma displays such a proof exemplarily. Note that the three equations Chop_i match directly the three cases that have to be considered according to the proof rule of Thm. 6.5.5. In particular, the predicate Q can therefore be found automatically by resolution.

More generally this means that the predicate Q can either be determined beforehand by characterizing every occurring function corecursively or case by case when a coinductive proof principle is employed. We argue that an a priori characterization should only be aspired to if it is the more natural one. Otherwise, the predicate Q should be provided on demand, as coinductive proofs will then be the exception.

Lemma 6.6.3 (Basic Properties about Chop)
The Chop operation has no relevance if a corresponding Filter operation is performed afterwards. Furthermore, it is idempotent and commutes with Map.

$$\mathsf{Filter}\ P\ `(\mathsf{Chop}\ P\ x) = \mathsf{Filter}\ P\ `x \qquad (1)$$
$$\mathsf{Chop}\ P\ (\mathsf{Chop}\ P\ x) = \mathsf{Chop}\ P\ x \qquad (2)$$
$$\mathsf{Map}\, f\ `(\mathsf{Chop}\ (P \circ f)\ x) = \mathsf{Chop}\ P\ (\mathsf{Map}\, f\ `x) \qquad (3)$$

Proof.
We prove only (2) using the take lemma support provided by Thm. 6.5.5. The remaining properties are proved analogously: they are slightly more complicated, as they involve not corecursively characterized operations as well (Filter, Map), but follow along the same line. Applying Thm. 6.5.5 to (2) with $Q := \lambda x.\ \neg P(x)$ and $A := \lambda s.\ \mathsf{True}$ leaves us with two base cases and the main case.

Base Cases: For both base cases we may assume $\mathcal{A}_1$: $\mathsf{Forall}\ (\lambda a.\ \neg P(a))\ s$.

- **Case** $\neg\mathsf{Finite}(s)$**:** We have to show $\mathcal{C}_1$: $\mathsf{Chop}\, P\ (\mathsf{Chop}\, P\ s) = \bot$ and $\mathcal{C}_2$: $\mathsf{Chop}\, P\ s = \bot$. $\mathcal{C}_2$ follows immediately by $(Chop_1)$ using $\mathcal{A}_1$ and the case information. Therefore, $\mathcal{C}_1$ reduces to $\mathsf{Chop}\, P\ \bot = \bot$ which again follows by $(Chop_1)$, where the assumptions are trivially fulfilled.

- **Case** $\mathsf{Finite}(s)$**:** Analogous to the previous case, using $(Chop_2)$ instead of $(Chop_1)$.

Main Case: For the main case the proof assumptions and obligations are given as follows:

$$\mathcal{A}_1\text{:}\quad \forall t\ m.\ m < n \Rightarrow \mathsf{take}\ m\ `(\mathsf{Chop}\ P\ (\mathsf{Chop}\ P\ t)) = \mathsf{take}\ m\ `(\mathsf{Chop}\ P\ t)$$
$$\mathcal{A}_2\text{:}\quad \mathsf{Forall}\ (\lambda x.\ \neg P(x))\ s_1 \wedge \mathsf{Finite}(s_1) \wedge P(a)$$
$$\mathcal{C}_1\text{:}\quad \mathsf{take}\ n\ `(\mathsf{Chop}\ P\ (\mathsf{Chop}\ P\ (s_1\ \oplus\ a\hat{}\,s_2))) = \mathsf{take}\ n\ `(\mathsf{Chop}\ P\ (s_1\ \oplus\ a\hat{}\,s_2))$$

Here it pays off that Chop is characterized corecursively, as $(Chop_3)$ matches $\mathcal{C}_1$ immediately. Concretely, the following calculations yield the result.

$$\begin{array}{ll} & \mathsf{take}\ n\ `(\mathsf{Chop}\ P\ (\mathsf{Chop}\ P\ (s_1\ \oplus\ a\hat{\ } s_2))) \\ = & \quad \{\textit{Cor. 6.6.2 } (Chop_3),\ \mathcal{A}_2\} \\ & \mathsf{take}\ n\ `(s_1\ \oplus\ a\hat{\ }\mathsf{Chop}\ P\ (\mathsf{Chop}\ P\ s_2)) \\ = & \quad \{\textit{Lemma 6.5.6 (Take-reduction)},\ \mathcal{A}_1\} \\ & \mathsf{take}\ n\ `(s_1\ \oplus\ a\hat{\ }\mathsf{Chop}\ P\ s_2) \\ = & \quad \{\textit{Cor. 6.6.2 } (Chop_3),\ \mathcal{A}_2\} \\ & \mathsf{take}\ n\ `(\mathsf{Chop}\ P\ (s_1\ \oplus\ a\hat{\ } s_2)) \end{array}$$

Given the $(Chop_i)$ the proof is completely automated in Isabelle. □

Lemma 6.6.4 (Chop is Prefix Closed)
The Chop operator is prefix closed w.r.t. the two prefix notions $\sqsubseteq$ and $\ll$.

$$\frac{\mathsf{Finite}(s)}{\mathsf{Chop}\ P\ s \ll s}\ (1) \qquad\qquad \frac{\neg\mathsf{Finite}(s)}{\mathsf{Chop}\ P\ s \sqsubseteq s}\ (2)$$

Proof.
The first proposition cannot be proved using the take lemma, as it involves an existential quantifier ($s \ll t = \exists r.\ t = r \oplus s$). Rather we employ finite structural induction, which is not hindered by an existential quantifier, because it does not require admissibility. The proof is nasty as Chop is easier dealt with coinductively than inductively.

The second proposition is proved using the take lemma version with $\sqsubseteq$ instead of $=$, by means of a proof rule analogous to Thm. 6.5.5. □

6.7 Conclusion and Related Work

We described a formalization of possibly infinite sequences as recursive domains in HOLCF. In the presentation emphasis was laid on logical design decisions and the methodological infrastructure provided for the underlying proof principles.

The following objections may be raised against the framework of domain theory: all types must denote domains; all functions must be continuous; admissibility obligations encumber reasoning. The contributions of this chapter, however, turn domain theory into a powerful as well as practical and convenient vehicle for reasoning about sequences. In detail, the mentioned objections are invalidated as follows:

First, the lifting datatype allows to employ domains only there where they are actually needed. This contributes to more adequate formalizations, but increases as well the degree of automation, saves reasoning about undefinedness, and offers the possibility of reusing theory libraries from HOL.

Second, continuity indeed requires functions to be computable. This means that non-computable functions cannot be defined without leaving the LCF sublanguage. However, we did not encounter any non-computable function apart from the fair merge function during our extensive use of the sequence package in verifying I/O automata theory. This is, at least partly, due to our important design decision to model sequences as consisting of three flavors instead of only two. This enables us, for example, to define powerful functions like infinite concatenation continuously.

Third, the notion of admissibility actually provides advantages rather than disadvantages. It allows to extend the familiar structural induction rule to infinite objects modulo a merely syntactical criterion. This makes induction a convenient and easily automated proof tool. Our experience with the syntactical criterion is quite encouraging: in almost all cases it was possible (sometimes by reformulating the goal) to satisfy this criterion or to get by with the finite induction rule. In the remaining cases, it turned out to be more reasonable to switch to coinduction than to prove admissibility by its definition. Therefore, we extended coinduction to deal also with properties for which the admissibility check fails. Furthermore, we provided a tailored proof infrastructure for the take lemma and bisimulation. This closes a gap which still existed between recursive definitions and coinductive proofs. Previously it has merely been shown in [Pit94] that coinduction is in principle derivable in domain theory, which has then been implemented in the domain package of HOLCF [Ohe95] without providing the necessary infrastructure.

Related Work. The entire following chapter is dedicated to discuss different formalizations of sequences. Here we merely mention some approaches that are, similar to our work, based on recursion.

The domain package of Isabelle/HOLCF is also used by Schaetz and Spies [SS95], who provide a formalization of streams for the development method FOCUS [BDD$^+$93, Bro93b]. However, their model consists of merely two sequence flavors: partial and infinite ones. This forces them to define concatenation non-continuously using Hilbert's ε-operator. It is by no means clear how to reason about this definition in a feasible way, or how to extend this definition to infinite concatenation. Nevertheless, most of the contributions of this chapter are easily (or have already been) transfered to their setting.

Recently, Feferman [Fef96] developed a recursion theory and applied it to the formalization of sequences. Similar to our sequence model, his solution incorporates finite, partial and infinite sequences. However, it does not require continuity. His approach has not been mechanized in a theorem prover yet.

Chapter 7

Comparison of Sequence Formalizations

In this chapter we compare the domain-theoretic sequence model of the previous chapter with four further formalizations of possibly infinite sequences in theorem provers based on higher-order logic. The formalizations have been carried out in Isabelle, Gordon's HOL, and PVS, mostly in the context of system verification. In the comparison particular emphasis is laid upon the proof principles made available for each approach.

7.1 Introduction

Sequences occur frequently in all areas of computer science and mathematics. In particular, formal models of reactive, distributed systems often employ (possibly infinite) sequences to describe system behavior over time. This applies to I/O automata, but as well to several further formalisms, e.g. TLA [Lam94], model checking frameworks [Hol91, CGL94, BBC⁺96], or FOCUS [BDD⁺93].

Recently, there is a growing interest in using theorem provers to support reasoning in such formalisms. This is the reason why the sequence model of the previous chapter represents not the only effort to formalize possibly infinite sequences. In this chapter we compare a number of such formalizations, which were carried out in theorem provers based on higher-order logic. The aim is to draw conclusions w.r.t. appropriateness and applicability.

Specifically, the following five approaches are evaluated and compared:

- HOL-Fun: Sequences are defined as functions by

$$(\alpha)\,\mathsf{sequence} \quad = \quad \mathsf{nat} \rightarrow (\alpha)\,\mathsf{option}$$

where the $(\alpha)\mathsf{option}$ datatype is used to incorporate finite sequences into the model: "non-existing" elements are marked explicitly by None . This approach has been taken by Nipkow, Slind and Müller [NS95, MN97], where it has been used to formalize parts of I/O automata meta-theory. It has been carried out in Isabelle/HOL [Pau94].

- PVS-FUN: Sequences are defined as functions from a downward closed subset of nat, where the cardinality of the subset corresponds to the length of the sequence. This is achieved by the dependent sum

$$(\alpha)\,\mathsf{sequence} \quad = \quad \sum S \in \mathsf{index\text{-}sets}\,.\; S \to \alpha$$

where $\mathsf{index\text{-}sets} \subseteq (\mathsf{nat})\,\mathsf{set}$ denotes the set of all downward closed subsets of nat. This approach has been taken by Devillers and Griffioen [DG97], who also formalized I/O automata meta-theory. It has been carried out in PVS [ORSH95].

- HOL-SUM: Sequences are defined as the disjoint sum of finite and infinite sequences:

$$(\alpha)\,\mathsf{sequence} \quad = \quad \mathsf{FinSeq}((\alpha)\,\mathsf{list}) \;\mid\; \mathsf{InfSeq}(\mathsf{nat} \to \alpha)$$

This approach has been taken by Chou and Peled [CP96] in the verification of a partial-order reduction technique for model checking and by Agerholm [Age94b] as an example of his formalization of domain theory. Both versions have been carried out independently from each other in Gordon's HOL [GM93].

- PVS-CO: Sequences are characterized by an unspecified type (α) $\mathsf{sequence}$ with destruction operator

$$\mathsf{next} \quad :: \quad (\alpha)\,\mathsf{sequence} \to (\alpha \times (\alpha)\,\mathsf{sequence})\,\mathsf{option}$$

incorporating the intuition of HD and TL. A single axiom postulates that every function f on sequences is uniquely determined by the outputs of f successively observed by next. This coalgebraic approach has been taken by Hensel and Jacobs [HJ97a], based on the principles presented in [HJ97b]. It has been carried out in PVS [ORSH95]. Further coalgebraic sequence formalizations have been performed by Paulson [Pau97] in Isabelle/HOL and by Leclerc and Paulin-Mohring [LPM93] in Coq [DFH^{+}93].

- HOL-LCF: In domain theory, sequences can be defined by the recursive domain equation

$$(\alpha)\,\mathsf{sequence} \quad = \quad \mathsf{nil} \;\mid\; (\alpha)\,\#\,(\alpha)\,\mathsf{sequence}$$

where the "cons"-operator $\#$ is strict in the first and lazy in its second argument. This approach is the one that has been presented in §6 and will therefore not be explained here. It has been carried out in Isabelle/HOLCF [MNOS98, Reg95].

In the comparison we consider the following five operations on possibly infinite sequences: *tl, map, length, concat*, and *filter*. Furthermore, we will develop a classification, which allows to evaluate all other standard functions as well. This guarantees a representative result for the general datatype of possibly infinite sequences.

The aim of any formalization is a sufficiently rich collection of theorems, such that independence on the specific model is reached. In our experience, this will not completely be possible, for instance because one always wants to be able to define new operations for the datatype. Therefore we focus our comparison on the proof principles that are offered by the respective approaches. Their usability, applicability and degree of automation are especially essential for the user of the sequence package and influence proof length considerably. In addition, specific characteristics of the respective logics are taken into account.

The chapter is organized as follows. In §7.2 – §7.5 the four newly introduced approaches are described in brief. In §7.6 each approach is discussed for itself. Finally, §7.7 provides a comparison and the resulting conclusions.

7.2 HOL-Fun: Functions in Isabelle/HOL

In the sequel we summarize the approach taken by Slind, Nipkow, and Müller [NS95, MN97] using Isabelle/HOL.

Definition 7.2.1 (Type of Sequences)
Sequences are defined by the type

$$(\alpha)\,\mathsf{sequence} \quad = \quad \mathsf{nat} \to (\alpha)\,\mathsf{option}$$

Furthermore a predicate is introduced, which has to hold or every well-formed sequence:

$$\mathsf{is\text{-}sequence}(s) \quad \equiv \quad (\forall i.\ s(i) = \mathsf{None} \Rightarrow s(i+1) = \mathsf{None})$$

The None element of the option datatype represent "nonexisting" elements and is used to model finite sequences. The is-sequence predicate avoids the case in which Nones appear within a sequence not as an infinite suffix of the sequence only.

Sequences therefore can be regarded as a quotient structure, where is-sequence characterizes the normal form of each equivalence class. Of course, every operation has to yield a value in normal form. This is the main disadvantage of this approach, as it is not straightforward to construct the normal form for e.g. the Filter function, which will be discussed below.

In the following we use several times natural numbers extended by an infinity element.

$$\textbf{datatype} \quad \mathsf{nat}_\infty \quad = \quad \mathsf{Fin}(\mathsf{nat}) \ \mid \ \mathsf{Inf}$$

In particular, we assume that the arithmetic operations and relations on them (as e.g. $+$, $-$, $<$) have been extended accordingly. Furthermore, the cardinality $\#$ on sets is supposed to be expanded to cover infinite sets as well.

Functions on sequences are defined pointwise. This is especially simple if the output length is equal to the input length (e.g. Map) or if it can easily be computed from it (e.g. $\oplus$).

Definition 7.2.2 (Basic Functions)

$$
\begin{array}{lcl}
\mathsf{TL} & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{TL}(s) & \equiv & \lambda i. s(i+1) \\[1ex]
\mathsf{Map} & :: & (\alpha \to \beta) \to (\alpha)\,\mathsf{sequence} \to (\beta)\,\mathsf{sequence} \\
\mathsf{Map}\, f\ s & \equiv & f \circ s \\[1ex]
\mathsf{Length} & :: & (\alpha)\,\mathsf{sequence} \to \mathsf{nat}_\infty \\
\mathsf{Length}(s) & \equiv & \#\{i \,.\, s(i) \neq \mathsf{None}\} \\[1ex]
\oplus & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
s \oplus t & \equiv & \lambda i.\ \mathbf{if}\ i < \mathsf{Length}(s)\ \mathbf{then}\ s(i)\ \mathbf{else}\ t(i - \mathsf{Length}(s))
\end{array}
$$

Filtering is divided into two steps (see Fig. 7.1): first, the projection function proj replaces every element not satisfying P by None, then the resulting sequence is brought into normal form. Normalization is achieved by an index transformation it, that has to meet three requirements: first, normalization has to maintain the ordering of the elements, second, every $\mathsf{Some}(a)$ has to appear in the normal form, and third, if there is a None in the normal form, then there will be no Some afterwards. These requirements can directly serve as the definition for it using Hilbert's description operator ε.

Definition 7.2.3 (Filter)

$$
\begin{array}{lcl}
\mathsf{proj} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{proj}\, P\ s & \equiv & \lambda i.\ \ \mathbf{case}\ s(i)\ \mathbf{of}\ \mathsf{None}\ \ \Rightarrow\ \ \mathsf{None} \\
 & & \qquad\qquad\qquad \mid \mathsf{Some}(a)\ \ \Rightarrow\ \ \mathbf{if}\ P(a)\ \mathbf{then}\ \mathsf{Some}(a)\ \mathbf{else}\ \mathsf{None} \\
\mathsf{it} & :: & (\alpha)\,\mathsf{sequence} \to (\mathsf{nat} \to \mathsf{nat}) \\
\mathsf{it}(s) & \equiv & \varepsilon\, it\,.\ \ \mathsf{monofun}(it) \wedge \\
 & & \qquad \forall i.\ s(i) \neq \mathsf{None} \Rightarrow i \in \mathsf{range}(it) \wedge \\
 & & \qquad \mathsf{is\text{-}sequence}(s \circ it) \\
\mathsf{NF} & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{NF}(s) & \equiv & s \circ \mathsf{it}(s) \\[1ex]
\mathsf{Filter} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{Filter}\ P\ s & \equiv & \mathsf{NF} \circ (\mathsf{proj}\ P\ s)
\end{array}
$$

The definition for it is a nice requirement specification, but it is not simple to work with it, as for every $\varepsilon\, x.\, Q(x)$ the existence of an x satisfying Q has to be shown. Theoretically, this

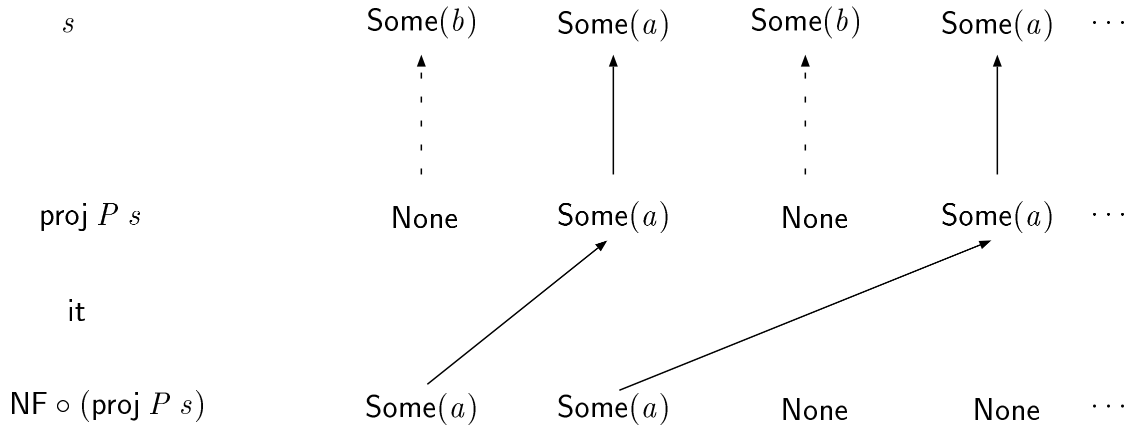

Figure 7.1: Filter function with $P = (\lambda x.\, x = a)$.

can be done using proof by contradiction, as we are in a classical logic, but it is not obvious how to do this in this case. In practice, an explicit construction seems to be unavoidable. In the subsequent approach PVS-FUN this index transformation will reappear in a similar fashion, and an explicit construction will be provided.

Theorem 7.2.4 (Proof Principles)
The standard proof principle in this setting is extensionality, i.e. the pointwise equality of functions, where sequences have to be in normal form.

$$\frac{\textsf{is-sequence}(s) \qquad \textsf{is-sequence}(t) \qquad \forall i.\; s(i) = t(i)}{s = t}$$

7.3 PVS-FUN: Functions in PVS

The sequence formalization in [DG97] makes use of PVS concepts, which will briefly be introduced in the sequel. The notation we use for the PVS logic will stay as near as possible to our previous presentations, although sets and types are identified in PVS.

The PVS Logic. Similar to HOL, the PVS logic [ORSH95] is based on higher-order logic, but type expressions are more expressive, featuring set-theoretic semantics. Whereas HOL allows only simple types, PVS offers mechanisms for *subtyping* and *dependent types*.

Subtyping is expressed with the usual set notation, e.g., $\{n \in \textsf{nat} \,.\, \mathit{even}(n)\}$ is the set of all even natural numbers. The dependent sum $\sum x \in A \,.\, B_x$ — in which the second component B_x depends on a member x of the first set A — denotes the set of all pairs (a, b) where $a \in A$ and $b \in B_a$. For example, if S^i denotes a string of length i then some members of $(i :: \textsf{nat} \times \{a, b, c\}^i)$ would be $(2, ab)$ and $(3, bac)$. A dependent product $\prod x \in A \,.\, B_x$ denotes all functions f where if $a \in A$ then $f(a) \in B_a$. For example, if f is a member of the dependent product $(i :: \mathsf{nat} \to \{a, b, c\}^i)$, then $f(2) = ab$ and $f(3) = acb$ are type-correct. Furthermore, we use π_0 and π_1 for the first and second projection in a pair, e.g., $\pi_0((a, b)) = a$. Note that type checking is not decidable in this framework.

To define sequences we first need the set of all downward closed subsets of the natural numbers, called *index sets*.

Definition 7.3.1 (Index Sets)

$$\begin{array}{lcl} \mathsf{below} & :: & \mathsf{nat} \to (\mathsf{nat})\,\mathsf{set} \\ \mathsf{below}(n) & \equiv & \{m \in \mathsf{nat} \,.\, m < n\} \\ \mathsf{index\text{-}sets} & = & \{\mathsf{below}(n) \,.\, n \in \mathsf{nat}\} \cup \mathsf{nat} \end{array}$$

Note that index-sets is isomorphic to nat_∞. We use index-sets to represent the possible domains of sequences which are modeled as functions. Finite sequences are represented by functions whose domain is an initial segment of nat, whereas infinite sequences are functions from the entire natural numbers. This is achieved by a dependent sum of an index set and a mapping from that index set to the data set. The sets of finite and infinite sequences are defined with the use of predicate subtyping, where the predicate finite describes sets with finite cardinality.

Definition 7.3.2 (Type of Sequences)

$$\begin{array}{lcl} (\alpha)\,\mathsf{sequence} & = & \sum S \in \mathsf{index\text{-}sets} \,.\, S \to \alpha \\ (\alpha)\mathsf{fin\text{-}sequence} & = & \{s \in (\alpha)\,\mathsf{sequence} \,.\, \mathsf{finite}(\pi_0(s))\} \\ (\alpha)\mathsf{inf\text{-}sequence} & = & \{s \in (\alpha)\,\mathsf{sequence} \,.\, \neg\mathsf{finite}(\pi_0(s))\} \end{array}$$

Subsequently, we will write $\mathsf{dom}(s)$ for the domain of a sequence s, and $s(i)$ for the i-th element of a sequence. $\lfloor S \rfloor$ denotes the smallest element of the set S, and Finite describes sequences of type (α)fin-sequence.

Since index-sets contains finite sets as well as an infinite set, basic functions on sequences can sometimes be defined without the distinction between finite and infinite arguments (e.g. Map). However, in other cases the distinction is still necessary to determine the domain of the sequence function (e.g. $\oplus$).

Definition 7.3.3 (Basic Functions)

$$\begin{array}{lcl} \mathsf{TL} & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\ \mathsf{TL}(s) & \equiv & \textbf{let} \quad S = \mathsf{dom}(s) \backslash \lfloor \mathsf{dom}(s) \rfloor \\ & & \textbf{in} \quad (S, \lambda i :: S.\, f\ (i+1)) \\ \mathsf{Map} & :: & (\alpha \to \beta) \to (\alpha)\,\mathsf{sequence} \to (\beta)\,\mathsf{sequence} \\ \mathsf{Map}\, f\ s & \equiv & (\mathsf{dom}(s), \lambda i :: \mathsf{dom}(s).\, f\ (s\ i)) \end{array}$$

$$
\begin{array}{lcl}
\mathsf{Length} & :: & (\alpha)\mathsf{fin\text{-}sequence} \to \mathsf{nat} \\
\mathsf{Length}(s) & \equiv & \#\mathsf{dom}(s) \\[4pt]
\oplus & :: & (\alpha)\mathsf{fin\text{-}sequence} \to (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
s \oplus t & \equiv & \textbf{let} \quad l = \mathsf{Length}(s);\ S = \textbf{if}\ \mathsf{Finite}(t)\ \textbf{then}\ \mathsf{below}(l + \mathsf{Length}(t))\ \textbf{else}\ \mathsf{nat} \\
 & & \textbf{in} \quad (S, (\lambda i :: S.\ \textbf{if}\ i < l\ \textbf{then}\ s(i)\ \textbf{else}\ t(i - l)))
\end{array}
$$

Filtering a sequence s w.r.t. a predicate P is basically defined by an index transformation on ordered sets of natural numbers. Let $\mathsf{witness}\, P\, s$ be the set of all indexes i which satisfy $P(s(i))$. Furthermore, for all ordered subsets S of the natural numbers let $\mathsf{it}(S)$ be the index transformation function satisfying $\mathsf{it}\ S\ i = j$, whenever j is the i-th element of S. Note that $\mathsf{it}(S)$ is as well a sequence, namely of natural numbers. Then $s \circ \mathsf{it}\,(\mathsf{witness}\ P\ s)$ is the desired filtered sequence.

As an example consider Fig. 7.2. The goal is to filter all symbols a in the sequence $s = [b, a, a, c, b, a, \ldots]$. Therefore, $P = (\lambda x.\ x = a)$ and $\mathsf{witness}\ P\ s = \{1, 2, 5, \ldots\}$. Then, the index transformation function $\mathsf{it}\,(\mathsf{witness}\ P\ s)$ is represented by the sequence $[1, 2, 5, \ldots]$. Therefore, $s \circ \mathsf{it}\,(\mathsf{witness}\ P\ s) = [b, a, a, c, b, a, \ldots] \circ [1, 2, 5, \ldots] = [a, a, a, \ldots]$.

Definition 7.3.4 (Filter)

$$
\begin{array}{lcl}
\mathsf{witness} & :: & \prod (S, s) \in ((\alpha)\,\mathsf{set} \to (\alpha)\,\mathsf{sequence})\,.\,(\mathsf{dom}(s))\,\mathsf{set} \\
\mathsf{witness}\ P\ s & \equiv & \{i \in \mathsf{dom}(s) \mid P(s(i))\} \\[4pt]
\mathsf{iso\text{-}set}(S) & :: & (\mathsf{nat})\,\mathsf{set} \to \mathsf{index\text{-}sets} \\
\mathsf{iso\text{-}set}(S) & \equiv & \textbf{if}\ \mathsf{finite}(S)\ \textbf{then}\ \mathsf{below}(\#(S))\ \textbf{else}\ \mathsf{nat} \\[4pt]
\mathsf{greater} & :: & (\mathsf{nat})\,\mathsf{set} \to \mathsf{nat} \to (\mathsf{nat})\,\mathsf{set} \\
\mathsf{greater}\ S\ 0 & = & S \\
\mathsf{greater}\ S\ (n+1) & = & \textbf{if}\ \ (\mathsf{greater}\ S\ n) = \emptyset\ \ \textbf{then}\ \ \emptyset\ \ \textbf{else}\ \ (\mathsf{greater}\ S\ n) \backslash \lfloor \mathsf{greater}\ S\ n \rfloor \\[4pt]
\mathsf{it} & :: & \prod S \in (\mathsf{nat})\,\mathsf{set}\,.\,\mathsf{iso\text{-}set}(S) \to S \\
\mathsf{it}(S) & \equiv & \lambda i.\ \lfloor \mathsf{greater}\ S\ i \rfloor \\[4pt]
\mathsf{Filter} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{Filter}\ P\ s & \equiv & (\mathsf{iso\text{-}set}\,(\mathsf{witness}\ P\ s), s \circ (\mathsf{it}\,(\mathsf{witness}\ P\ s)))
\end{array}
$$

Note that the index transformation it in this context plays the same rôle as for HOL-FUN. However, here it is not merely defined descriptively, but given in a constructive way. Basically, $\mathsf{it}\ S\ i$ is defined recursively by removing the i smallest elements of S and taking then the smallest element of the remaining set.

However, there is a technical difference between the index transformations used in HOL-FUN and PVS-FUN. Whereas in PVS-FUN the index transformation is an integral part of Filter, in HOL-FUN it is separated into the additional function NF that generates the

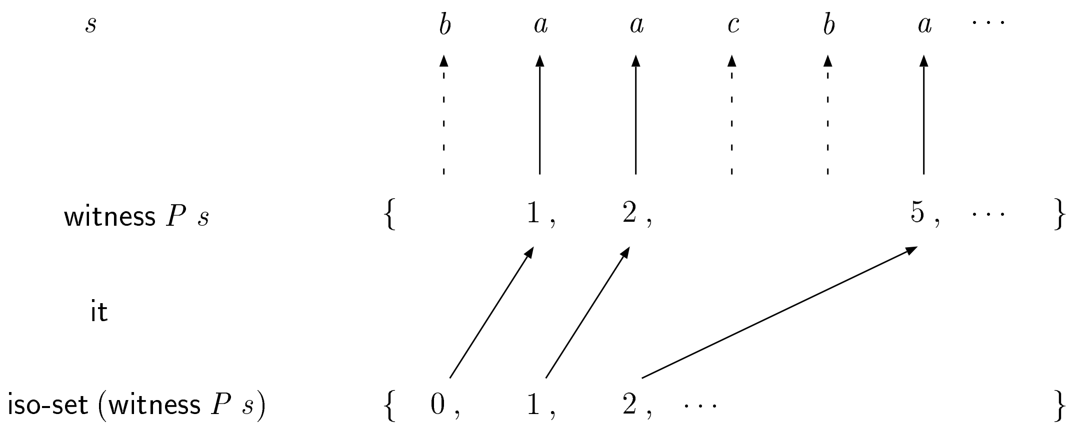

Figure 7.2: Filter function in PVS-Fun

normal form. This is necessary and not only a matter of methodology. Consider an infinite sequence s filtered to a finite one, say of length n. An index transformation in HOL-Fun would have to relate the Nones appearing for all indexes greater n to indexes of the original sequences s, which is not possible, as s is infinite. In PVS-Fun this is possible, as finite sequences have indeed finite domains, due to the construction by dependent types.

Nevertheless, the index transformations are in both cases the reason why proofs about Filter tend to be tremendously complicated, although proofs about the previous sequence operators are mostly simple and straightforward.

Theorem 7.3.5 (Proof Principles)
The standard proof principle in this setting is extensionality, i.e. the pointwise equality of functions, where the domains of the functions have to coincide.

$$\frac{\mathsf{dom}(s) = \mathsf{dom}(t) \qquad \forall i \in \mathsf{dom}(s).\ s(i) = t(i)}{s = t}$$

For finite sequences, rules for structural induction and induction on the length of the sequence have been derived as well.

7.4 HOL-Sum: Lists and Functions in Gordon's HOL

In the following we present the approach by Chou and Peled [CP96] carried out in Gordons's HOL [GM93]. Agerholm [Age94b] formalized a similar model that relies on domain theory and is more elaborate. We will discuss the relevant extensions in §7.6.

Definition 7.4.1 (Type of Sequences)
Sequences are defined as the disjoint union of lists (representing finite sequences) and functions from the natural numbers (representing infinite sequences).

$$(\alpha)\,\mathsf{sequence} \;=\; \mathsf{FinSeq}((\alpha)\,\mathsf{list}) \;\mid\; \mathsf{InfSeq}(\mathsf{nat} \to \alpha)$$

The twofold character of $(\alpha)\mathsf{sequence}$ is reflected in the definition of all operators on it. For most of the basic functions it is sufficient to implement them separately for finite lists and for functions. As examples, see the following definitions of TL, Map, and Length, where tl, map, and length denote the respective functions on finite lists.

Definition 7.4.2 (Basic Functions)

$$\begin{array}{lcl}
\mathsf{TL} & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{TL}(\mathsf{FinSeq}\; l) & = & \mathsf{tl}(l) \\
\mathsf{TL}(\mathsf{InfSeq}\, f) & = & \mathsf{InfSeq}(\lambda i.\, f(i+1)) \\[1ex]
\mathsf{Map} & :: & (\alpha \to \beta) \to (\alpha)\,\mathsf{sequence} \to (\beta)\,\mathsf{sequence} \\
\mathsf{Map}\; g\; (\mathsf{FinSeq}\; l) & = & \mathsf{map}\; g\; l \\
\mathsf{Map}\; g\; (\mathsf{InfSeq}\, f) & = & \mathsf{InfSeq}(g \circ f) \\[1ex]
\mathsf{Length} & :: & (\alpha)\,\mathsf{sequence} \to \mathsf{nat}_\infty \\
\mathsf{Length}(\mathsf{FinSeq}\; l) & = & \mathsf{Fin}(\mathsf{length}\; l) \\
\mathsf{Length}(\mathsf{InfSeq}\, f) & = & \mathsf{Inf}
\end{array}$$

Whenever it is not straightforward to define operators separately for both representations, the idea is to stay in one representation as long as possible and to convert the result afterwards. For the functional representation, for instance, there is a conversion function seq, which takes a number $n :: \mathsf{nat}_\infty$ and a function f as arguments, and constructs the corresponding sequence to f of length n. In the definition below, $\mathsf{genlist}\, f\; n$ is the finite list of the first n values $f(1), \ldots, f(n)$.

Definition 7.4.3 (Function Conversion)

$$\begin{array}{lcl}
\mathsf{seq} & :: & \mathsf{nat}_\infty \to (\mathsf{nat} \to \alpha) \to (\alpha)\,\mathsf{sequence} \\
\mathsf{seq}\; (\mathsf{Fin}\; n)\, f & = & \mathsf{FinSeq}(\mathsf{genlist}\, f\; n) \\
\mathsf{seq}\; \mathsf{Inf}\, f & = & \mathsf{InfSeq}(f)
\end{array}$$

The concatenation function $\oplus$, for example, is defined by means of this function. If $\oplus$ were defined using standard case distinctions on the arguments, one would need four cases instead of two. In the following definition, $\mathsf{nth}\; s\; i$ denotes the i-th element of s.

Definition 7.4.4 (Concatenation)

$$\begin{array}{lcl}
\oplus & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
s \oplus t & \equiv & \mathsf{seq}\; (\mathsf{Length}(s) + \mathsf{Length}(t)) \\
 & & \quad (\lambda i.\; \mathbf{if}\; i < \mathsf{Length}(s)\; \mathbf{then}\; \mathsf{nth}\; s\; i\; \mathbf{else}\; \mathsf{nth}\; t\; (i - \mathsf{Length}(s)))
\end{array}$$

Once more, Filter is the most complicated function to define. This time, because it may produce both finite and infinite sequences from an infinite sequence. Therefore, a connection between the finite and infinite representation has to be constructed. It is achieved by defining Filter as the limit of all finite filtered prefixes, which are in turn defined via recursion on lists. Therefore, the operation is essentially performed on the finite representation and converted to general sequences in the end, just the opposite approach as taken for $\oplus$.

In detail, $\sqsubseteq$ and chains are defined according to the prefix ordering. The limit of a chain c is the sequence $\mathsf{seq}\, n\, f$ where the length n is the least upper bound lub of all lengths in the chain c, and $f(i)$ is the i-th element of the first sequence in the chain that contains at least i elements. The function $\mathsf{Filter}\ P\ s$ is then the limit of the chain of all filtered prefixes of s, which are produced by $\mathsf{FilterChain}$. In the following definition, $\mathsf{mk\text{-}list}$ transforms finite sequences into finite lists, and $\mathsf{take}\ s\ i$ denotes the prefix of s up to the i-th element.

Definition 7.4.5 (Filter)

$$
\begin{array}{lcl}
\sqsubseteq & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \to \mathsf{bool} \\
s \sqsubseteq t & \equiv & \exists u.\ t = s \oplus z \\[4pt]
\mathsf{chain} & :: & (\mathsf{nat} \to (\alpha)\,\mathsf{list}) \to \mathsf{bool} \\
\mathsf{chain}(c) & \equiv & \forall j.\ c(j) \sqsubseteq c(j+1) \\[4pt]
\mathsf{limit} & :: & (\mathsf{nat} \to (\alpha)\,\mathsf{list}) \to (\alpha)\,\mathsf{sequence} \\
\mathsf{limit}(c) & \equiv & \mathsf{seq}\ (\mathsf{lub}\ (\lambda n.\ \exists j.\ n = \mathsf{Length}(c\ j))) \\
 & & \qquad (\lambda i.\ \mathsf{nth}\ (c\ (\mathsf{least}(\lambda j.\ i < \mathsf{Length}(c\ j))))\ i) \\[4pt]
\mathsf{FilterChain} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to \mathsf{nat} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{FilterChain}\ P\ s & \equiv & \lambda i.\ \mathsf{FinSeq}(\mathsf{filter}\ P\ (\mathsf{mk\text{-}list}(\mathsf{take}\ s\ i))) \\[4pt]
\mathsf{Filter} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{Filter}\ P\ s & \equiv & \mathsf{limit}\ (\mathsf{FilterChain}\ P\ s)
\end{array}
$$

For instance, when filtering all even numbers out of the sequence of squares $[1, 4, 9, 16, 25, \ldots]$ the resulting chain will be $\mathsf{nil} \sqsubseteq [4] \sqsubseteq [4] \sqsubseteq [4, 16] \sqsubseteq \ldots$. The limit of this chain is, of course, the infinite sequence of squares of even numbers.

Theorem 7.4.6 (Proof Principles)
The basic proof principles are structural induction on finite lists and extensionality for infinite sequences. Using seq, proofs have to be split up as follows:

$$\frac{\forall n\, f.\ P(\mathsf{seq}\ (\mathsf{Fin}\ n)\ f) \qquad \forall g.\ P(\mathsf{seq}\ \mathsf{Inf}\ g)}{\forall t.\ P(t)}$$

The following extensionality principle is also available:

$$\frac{\mathsf{Length}(s) = \mathsf{Length}(t) \qquad \forall i < \mathsf{Length}(s).\ \mathsf{nth}\ s\ i = \mathsf{nth}\ t\ i}{s = t}$$

For particular functions as *filter* and $\oplus$, the notions of chains, limits and sometimes continuity are used to prove equality of sequences only by proving their equality for all finite sequences.

7.5 PVS-Co: Coalgebraic Lists in PVS

In the sequel we present the PVS framework for coalgebraicly defined sequences developed in [HJ97a]. Although the main ideas are reproduced directly, the actual encoding and presentation has been considerably modified and improved. Differences to the alternative coalgebraic frameworks by Paulson [Pau97] in Isabelle and by Leclerc and Paulin-Mohring [LPM93] in Coq [DFH$^+$93] are discussed mainly in §7.6.

Definition 7.5.1 (Type of Sequences)
Sequences are characterized by an unspecified, parametrized type $(\alpha)\,\mathsf{sequence}$ with destruction operator

$$\mathsf{next} \quad :: \quad (\alpha)\,\mathsf{sequence} \to (\alpha \times (\alpha)\,\mathsf{sequence})\,\mathsf{option}$$

The approach is in some sense dual to HOL-LCF. Whereas domain-theoretic sequences are characterized by the *constructors* nil and $\hat{}$, the next operation here incorporates the intuition of the *destructors* HD and TL, together with the additional output None, if the sequence is empty[1].

First, we describe the single axiom, which is stated to derive both a definition and a proof principle for corecursively defined sequences. The key idea is that functions f of type $\beta \to (\alpha)\mathsf{sequence}$ are always defined via an auxiliary function h of type $\beta \to (\alpha \times \beta)\mathsf{option}$. The desired function f is then obtained by "unfolding" h. The function h is called a *structure map*.

Definition 7.5.2 (Structure Maps)

$$\begin{array}{lcl}
(\alpha,\beta)\mathsf{struct\text{-}type} & = & \beta \to (\alpha \times \beta)\,\mathsf{option} \\[4pt]
\mathsf{struct\text{-}map} & :: & (\alpha,\beta)\mathsf{struct\text{-}type} \to (\beta \to (\alpha)\,\mathsf{sequence}) \to \mathsf{bool} \\
\mathsf{struct\text{-}map}\; h\, f & \equiv & \forall s.\ \mathsf{next}(f(s)) = \textbf{case}\; h(s)\; \textbf{of} \\
 & & \qquad\qquad\qquad\quad \mathsf{None} \quad \Rightarrow \quad \mathsf{None} \\
 & & \qquad\qquad\qquad\ \mid\ \mathsf{Some}\,(a, s_1) \quad \Rightarrow \quad \mathsf{Some}(a, f(s_1))
\end{array}$$

The intuition behind this definition style is exactly dual to recursive definitions in HOL-LCF. For any element s the structure map h is used to determine the next *output* element of the desired function f, i.e. h either determines that the output is empty or produces the

[1] We will define HD and TL directly by means of next, as soon as the empty sequence has been defined.

next output element a together with the remaining rest s_1 of s, which has still to be treated by f. Recursive definitions, however, determine how the next *input* element is treated by f, i.e. it is declared how the empty sequence is treated and how a possible input a is treated, where the remaining rest s_1 is trivially determined as the tail of s.

The following axiom states now that for every structure map h there is a unique function f generated from h by unfolding.

$$\textbf{axiom} \qquad \forall h.\, \exists!\, f.\, \mathsf{struct\text{-}map}\ h\ f \qquad (ax)$$

Subtyping is used to denote the only function f satisfying $\mathsf{struct\text{-}map}\ h\ f$ as $\mathsf{corec}(h)$. The axiom is sound because of category theory: it simply states that we deal with a final coalgebra [JR97]. In the following the *existence* stated in (ax) is used as a *definition* principle for functions dealing with sequences. Later on, the *uniqueness* will be employed to derive the associated *proof* principle for sequences.

As a simple example we define the usual constructors nil and ˆ using the inverse of next as the structure map h.

Definition 7.5.3 (Constructors)

$$\begin{array}{lcl}
\mathsf{next\text{-}inv\text{-}h} & :: & (\alpha, (\alpha \times (\alpha)\,\mathsf{sequence})\,\mathsf{option})\mathsf{struct\text{-}type} \\
\mathsf{next\text{-}inv\text{-}h}\ opt & \equiv & \textbf{case}\ opt\ \textbf{of} \\
 & & \quad\ \ \mathsf{None} \quad\quad\ \Rightarrow\ \mathsf{None} \\
 & & \quad \mid \mathsf{Some}\,(a, s)\ \Rightarrow\ \mathsf{Some}(a, opt) \\
\mathsf{next\text{-}inv} & :: & (\alpha \times (\alpha)\,\mathsf{sequence})\,\mathsf{option} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{next\text{-}inv} & \equiv & \mathsf{corec}(\mathsf{next\text{-}inv\text{-}h}) \\
\mathsf{nil} & \equiv & \mathsf{next\text{-}inv}(\mathsf{None}) \\
a\,\hat{}\,s & \equiv & \mathsf{next\text{-}inv}(\mathsf{Some}(a, s))
\end{array}$$

As mentioned above, the usual destructors are now easily defined.

Definition 7.5.4 (Destructors)

$$\begin{array}{lcl}
\mathsf{HD} & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{option} \\
\mathsf{HD}(s) & \equiv & \textbf{case}\ \mathsf{next}(s)\ \textbf{of}\ \mathsf{None} \Rightarrow \mathsf{None} \mid \mathsf{Some}(a, s_1) \Rightarrow \mathsf{Some}(a) \\
\mathsf{TL} & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{TL}(s) & \equiv & \textbf{case}\ \mathsf{next}(s)\ \textbf{of}\ \mathsf{None} \Rightarrow \mathsf{nil} \mid \mathsf{Some}(a, s_1) \Rightarrow s_1
\end{array}$$

Further definitions of functions f on sequences always follow the same scheme: in order to construct the structure map h the input sequence s is inspected to find the element which is responsible for the next output produced by f. This is especially simple if the definition is pointwise, i.e. if the desired input element is directly available as the head of s. This means that a case distinction w.r.t. $\mathsf{next}(s)$ suffices to define h. Examples are Map and $\oplus$.

Definition 7.5.5 (Map and Concatenation)

$$
\begin{array}{lll}
\mathsf{Map\text{-}h} & :: & (\alpha \to \beta) \to (\beta, (\alpha)\,\mathsf{sequence})\mathsf{struct\text{-}type} \\
\mathsf{Map\text{-}h}\, f\ s & \equiv & \textbf{case}\ \mathsf{next}(s)\ \textbf{of} \\
 & & \quad \begin{array}{lll} \ \ \mathsf{None} & \Rightarrow & \mathsf{None} \\ \mid\ \mathsf{Some}\,(a, s_1) & \Rightarrow & \mathsf{Some}(f(a), s_1) \end{array} \\
\mathsf{Map}\, f & \equiv & \mathsf{corec}(\mathsf{Map\text{-}h}\, f)
\end{array}
$$

$$
\begin{array}{lll}
\oplus\mathsf{\text{-}h} & :: & (\alpha, (\alpha)\,\mathsf{sequence} \times (\alpha)\,\mathsf{sequence})\mathsf{struct\text{-}type} \\
s \oplus\mathsf{\text{-}h}\ t & \equiv & \textbf{case}\ \mathsf{next}(s)\ \textbf{of} \\
 & & \quad \begin{array}{lll} \ \ \mathsf{None} & \Rightarrow & \textbf{case}\ \mathsf{next}(t)\ \textbf{of} \\ & & \quad \begin{array}{lll} \ \ \mathsf{None} & \Rightarrow & \mathsf{None} \\ \mid\ \mathsf{Some}\,(b, t_1) & \Rightarrow & \mathsf{Some}(b, (\mathsf{nil}, t_1)) \end{array} \\ \mid\ \mathsf{Some}\,(a, s_1) & \Rightarrow & \mathsf{Some}(a, (s_1, t)) \end{array} \\
\oplus & \equiv & \mathsf{corec}(\oplus\mathsf{\text{-}h})
\end{array}
$$

The case is more complicated for functions, where the desired input element has to be searched in some way. The filter function is an instructive example. In particular, for Filter it is possible that for some s the desired input element does not exist at all, as the output is empty, although the input sequence s still contains some elements. Therefore, the search process is not computable and cannot be "programmed". There are several solutions out. In [HJ97a] the search is specified *declaratively*. The definition is given below, where the functions nth :: (α) sequence $\to$ nat $\to$ (α) option (denoting the n-th element of a sequence, if it exists) and Drop :: (α) sequence $\to$ nat $\to$ (α) sequence (denoting the n-th tail of a sequence) are defined by primitive recursion on the nat argument.

Definition 7.5.6 (Filter)

$$
\begin{array}{lll}
\mathsf{holds\text{-}first\text{-}at} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to (\mathsf{nat})\,\mathsf{option} \\
\mathsf{holds\text{-}first\text{-}at}\ P\ s & \equiv & \textbf{if}\ \forall n.\ \mathsf{nth}\ s\ n \neq \mathsf{None} \Rightarrow \neg P(\mathsf{the}(\mathsf{nth}\ s\ n)) \\
 & & \textbf{then}\ \mathsf{None} \\
 & & \textbf{else}\ \mathsf{Some}(\mathsf{least}(\{n \mid \mathsf{nth}\ s\ n \neq \mathsf{None} \wedge P(\mathsf{the}(\mathsf{nth}\ s\ n))\}))
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{Filter\text{-}h} & :: & (\alpha \to \mathsf{bool}) \to (\alpha, (\alpha)\,\mathsf{sequence})\mathsf{struct\text{-}type} \\
\mathsf{Filter\text{-}h}\ P\ s & \equiv & \textbf{case}\ \mathsf{holds\text{-}first\text{-}at}\ P\ s\ \textbf{of} \\
 & & \quad \begin{array}{lll} \ \ \mathsf{None} & \Rightarrow & \mathsf{None} \\ \mid\ \mathsf{Some}\ n & \Rightarrow & \mathsf{Some}(\mathsf{the}(\mathsf{nth}\ s\ n), \mathsf{Drop}\ s\ (1+n)) \end{array} \\
\mathsf{Filter}\ P & \equiv & \mathsf{corec}(\mathsf{Filter\text{-}h}\ P)
\end{array}
$$

Inspired by the comparison in this chapter, Paulson extended his sequence model in [Pau97] by a filter function where the necessary search is specified *inductively*. Concretely, the relation between a given sequence s and its possibly existing longest suffix t, whose head satisfies P, is formalized by an inductive definition. If it does not exist, the desired t is set to nil. This seems to be a convenient definition combining induction and coinduction in an appropriate way.

Definition 7.5.7 (Alternative Definition of Filter)

$$
\begin{array}{lcl}
\textsf{search-suffix} & :: & (\alpha \to \textsf{bool}) \to ((\alpha)\,\textsf{sequence} \times (\alpha)\,\textsf{sequence})\,\textsf{set} \\
\textbf{inductive} & & \dfrac{P(a)}{(a\hat{\ }s, a\hat{\ }s) \in \textsf{search-suffix}(P)} \qquad \dfrac{\neg P(a) \quad (s,t) \in \textsf{search-suffix}(P)}{(a\hat{\ }s, t) \in \textsf{search-suffix}(P)} \\
\textsf{get-suffix} & :: & (\alpha \to \textsf{bool}) \to (\alpha)\,\textsf{sequence} \to (\alpha)\,\textsf{sequence} \\
\textsf{get-suffix}\ P\ s & \equiv & \varepsilon\, t.\ (s,t) \in \textsf{search-suffix}(P) \lor (t = \textsf{nil} \land s \notin \mathbf{domain}(\textsf{search-suffix}(P))) \\
\textsf{Filter-h} & :: & (\alpha \to \textsf{bool}) \to (\alpha, (\alpha)\,\textsf{sequence})\textsf{struct-type} \\
\textsf{Filter-h}\ P\ s & \equiv & \textbf{case}\ \textsf{get-suffix}\ P\ s\ \textbf{of} \\
 & & \quad \textsf{nil} \;\Rightarrow\; \textsf{None} \\
 & & \mid\ a\hat{\ }t \;\Rightarrow\; \textsf{Some}(a,t) \\
\textsf{Filter}\ P & \equiv & \textsf{corec}(\textsf{Filter-h}\ P)
\end{array}
$$

Similar combinations of induction and coinduction are possible for infinite concatenation, but have not yet been formalized in Isabelle.

Theorem 7.5.8 (Proof Principles)
The natural proof principle in this setting is the bisimulation rule

$$\frac{\textsf{bisim}\ h_1\ h_2\ R \quad (x_1, x_2) \in R}{(\textsf{corec}\ h_1)\ x_1 = (\textsf{corec}\ h_2)\ x_2}$$

which is easily derived from the uniqueness stated in axiom (ax). Here, the bisimulation property is specified by the predicate bisim:

$$
\begin{array}{lcl}
\textsf{bisim} & :: & (\alpha_1, \beta_1)\textsf{struct-type} \to (\alpha_2, \beta_2)\textsf{struct-type} \to (\beta_1 \times \beta_2)\,\textsf{set} \to \textsf{bool} \\
\textsf{bisim}\ h_1\ h_2\ R & \equiv & \forall x_1\ x_2.\ (x_1, x_2) \in R \Rightarrow \\
 & & \qquad h_1(x_1) = \textsf{None} \Leftrightarrow h_2(x_2) = \textsf{None} \land \\
 & & \qquad \exists x_1', x_2', a_1, a_2.\ h_1(x_1) = \textsf{Some}\ (a_1, x_1') \Rightarrow \\
 & & \qquad\qquad h_2(x_2) = \textsf{Some}\ (a_2, x_2') \\
 & & \qquad\qquad a_1 = a_2 \land (x_1', x_2') \in R
\end{array}
$$

A take lemma can easily be proved via bisimulation.

$$\frac{\forall n.\ \textsf{nth}\ s\ n = \textsf{nth}\ t\ n}{s = t}$$

Note that the bisimulation rule is more general than the analogous rule for HOL-LCF: the bisimulation relation is not required directly for the sequences s and t, which are shown to be equal, but only for elements x_1 and x_2 of arbitrary type, which then generate s and t via two structure maps h_1 and h_2. On the other hand, the take lemma is weaker than the corresponding one in HOL-LCF, as it refers to nth instead of take. As a result, the take lemma follows from bisimularity and not vice versa, as in HOL-LCF.

In the setting of [HJ97a] rules for proving invariants and for structural induction are derived as well. However, the latter is only reasonably applicable for finite sequences, as the otherwise necessary admissibility obligation cannot be discharged automatically, as continuity cannot be guaranteed (e.g. Filter is not defined continuously in this setting).

7.6 Discussion of the Individual Approaches

In this section we discuss the different approaches separately, i.e. for each approach advantages and drawbacks are analyzed. Furthermore, differences between alternative versions of the same approach are mentioned. Comparisons between different approaches and resulting conclusions will be provided in §7.7.

Discussion of HOL-FUN**.** As already mentioned, the crucial point of this formalization is the dependency of filtering on normal forms: if no normal form is postulated for sequences, then filtering is easy to define as projection, otherwise filtering requires a sophisticated index transformation, which makes reasoning about it extremely awkward.

Therefore, why do not abstain from normal forms at all? The problem is that for most applications None-elements between defined sequence elements have no semantical counterpart, i.e. all sequences of an equivalence class are semantically equivalent. In a presentation without normal forms, however, this semantical equivalence cannot be expressed.

To illustrate this, we consider reactive systems, where a distinction is made between observable and non-observable actions. Take I/O automata as an example. Traces characterize the observable behaviors, whereas schedules contain internal as well as external actions. Implementation is defined via trace inclusion. The problem is that this notion of implementation cannot be captured by sequences without normal forms. This is due to None-steps which stand for nothing, but in fact are not really nothing. Rather they enforce the same time raster between implementation and specification, which does not allow the implementation or the specification to make internal steps while the other component waits. Adding arbitrary "stuttering" steps [AL88] merely helps in one direction: the implementation is allowed to take internal steps, but the specification is not. What is necessary is to allow "mumbling" [Bro93a], i.e. the removal of None-steps, which corresponds to the generation of normal forms.

As a concrete counterexample see Fig. 7.3, where two I/O automata with the same set of traces are depicted. If filtering is just represented by projection, we do not get $traces(C) \subseteq traces(A)$. Indeed, Fig. 7.4 shows traces of C which are not traces of A.

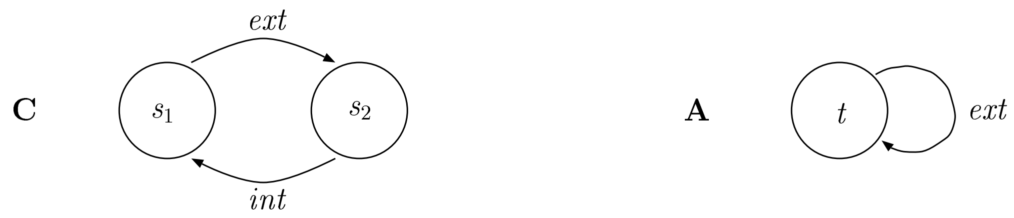

Figure 7.3: Automata with same observable behavior

	All possible schedules	All possible traces
C	$(\mathsf{Some}(ext)\ \mathsf{Some}(int))^*$	$(\mathsf{Some}(ext)\ \mathsf{None})^*$
A	$\mathsf{Some}(ext)^*$	$\mathsf{Some}(ext)^*$

Figure 7.4: Traces which should represent the same observable behavior

The starting point of this thesis was the I/O automaton model in [NS95] based on the HOL-FUN sequence formalization, but without normal forms. By adding arbitrary stuttering steps, the implementation could perform internal steps, but the specification could not. Therefore, merely a very restricted notion of implementation could be supported (cf. §8.3). Nevertheless, a nontrivial case study has been carried out in this model.

The intention of this thesis is to support the original implementation relation between I/O automata. However, we did not extend the existing sequence model by normal forms, but developed the new sequence model HOL-LCF. The reason was the awkward index transformations, which are discussed in the next paragraph.

Discussion of PVS-FUN**.** Similar to HOL-FUN, the crucial point of this formalization is the complexity of index transformations needed for the filter function. At first sight it seems to be a facilitation that this transformation is not given descriptively as in HOL-FUN, but by an explicit construction. However, the intrinsic difficulties are inherently connected with the relocation of elements in a functional model, i.e. a model where indexes and therefore fixed locations play a vital rôle. Therefore, proofs of even the most basic properties about Filter, as e.g. its idempotency (which in HOL-LCF is proved automatically), appear to be very awkward.

Discussion of HOL-SUM**.** The HOL-SUM approach is a pragmatic mixture of algebraic lists for finite sequences and functions for infinite sequences. Equality of sequences therefore can be proven with structural induction for the finite case, and function equality in the infinite case. Therefore proofs always show a twofold character. The Filter function, however, is still a problem, as there the two representations have to be related, since Filter may produce a finite sequence from an infinite one. For this reason, notions of chains, limits and continuity are introduced in [CP96] which, however, are only used for proofs about specific functions, whereas general proof principles known from domain theory are not developed.

Agerholm [Age94b], however, takes this step and carries over the entire world of domain theory to this setting. For that purpose he employs his embedding of domain theory in Gordon's HOL, called HOL-CPO [Age94b]. The partial order $\sqsubseteq$ on sequences is defined in a threefold fashion — both sequences finite, both infinite, and finite/infinite. The main disadvantage of this approach is that this results in up to six versions of every single fact, because proofs show a twofold character. Agerholm himself concludes that this development of sequences was long and tedious (50 pages of 70 lines each) and rather an "ad-hoc approach".

Discussion of PVS-Co**.** As already mentioned, the approach PVS-Co is almost perfectly dual to HOL-LCF. Therefore, it is of particular interest to compare these both approaches. As this will be done in §7.7 in detail, we sketch here only alternative versions of PVS-Co.

Closely related to PVS-Co is the coalgebraic formalization of sequences in Isabelle/HOL by Paulson [Pau97]. Paulson does not provide a body of lemmas as large as the formalizations discussed so far, because his theory of sequences represents only an example for a more general framework for (co)induction and (co)recursion. However, a remarkable advantage w.r.t. PVS-Co is that he does not state the existence and uniqueness of corec as an axiom, but derives them within Isabelle. This is possible because Paulson does not introduce an *unspecified* type sequence (as done in PVS-Co), but encodes this type in a sophisticated way as greatest fixpoint of monotone operators on a suitable domain.

Furthermore, Leclerc and Paulin-Mohring formalize streams coalgebraicly in Coq [LPM93] using existential types. Their impredicative encoding provides a corecursion combinator similar to corec, which is based on [Wra89]. However, existential types (without an additional axiomatization for uniqueness) yield only weakly final coalgebras and therefore only the existence part of corec. Therefore, they cannot develop coinductive proof principles like bisimulation or invariants, but rather stick to stream specific principles using indexes and positions.

Discussion of HOL-LCF**.** For a comprehensive discussion of HOL-LCF the reader is referred to the previous chapter, in particular to §6.7. Here, we will merely add the following additional point which is interesting in the context of this chapter.

Definitions in HOL-LCF have a computational semantics, whereas the other formalizations feature descriptive semantics. This is best explained by the equation $\mathsf{Filter} P`(x \oplus y) = \mathsf{Filter}\ P\ `x \oplus \mathsf{Filter}\ P\ `y$ which holds in HOL-LCF in general, but in the other formalizations only for finite x. Take, for example, $P = (\lambda x.\ x = a)$, $x = [a, b, b, b, \ldots]$ and $y = [a, a, a, \ldots]$. Then, in HOL-LCF both sides produce $[a?]$, the other formalizations, however, produce $[a!]$ and $[a, a, a, \ldots]$. The reason is that the filter function in the domain-theoretic formalization returns "divergence" which is preserved by concatenation.

We argue that this computational semantics is more appropriate for applications where sequences represent the behaviour of any kind of computing device, as for example modeled by I/O automata. However, it is possible to get a descriptive semantics in HOL-LCF as well by using a non-continuous $\oplus$ operator, see [SS95]. Just the other way around, it is possible to define an alternative next operator in PVS-Co, which features a computational semantics, see [HJ97a]. Therefore, the type of semantics seems not to be an especially important distinguishing feature.

7.7 Comparative Analysis

Based on the analysis of §7.6 we now compare the different approaches and draw conclusions. First, we explain why the three approaches HOL-Fun, PVS-Fun, and HOL-Sum are not really appropriate. Then, we discuss the remaining two formalizations: the algebraic and the coalgebraic framework.

7.7.1 Evaluating HOL-Fun, PVS-Fun, and HOL-Sum

As mentioned earlier, HOL-Fun and PVS-Fun are similar to the extent that they both use functions to define sequences. To achieve this common goal, two complementary ways are chosen. Whereas HOL-Fun *extends* the *codomain* of the function by the element None modeling partiality, PVS-Fun *restricts* the *domain* of the function using dependent types. As both approaches are based on functions, operations are defined via indexes, and the natural proof principle is extensionality. This is adequate for operations where the element positions of the original and resulting sequence can easily be computed from each other, which is the case for functions like TL, Map, $\oplus$, etc. (let us call them to be *of type* M). It is, however, infeasible for functions that change element positions in an inhomogenous way. Examples for such functions are Filter, Flatten, Takewhile, and Dropwhile (let us call them to be *of type* F). We have seen in §7.6 that the problems are due to the complexity of index transformations, which on the one hand cannot be circumvented, and on the other hand cannot be significantly facilitated by a constructive instead of declarative definition. It seems to be inappropriate in general to handle functions of type F by, intuitively speaking, modifying existing sequences via the relocation of elements. Instead it appears to be preferable to build a really new sequence, either by regarding the input of the function (recursion in HOL-LCF) or the output of the function (corecursion in PVS-Co).

As a result, the approaches HOL-Fun and PVS-Fun appear to be infeasible for applications where functions of type F are required. Therefore, there are only very restricted application areas, like refinement notions with the same time frame for specification and implementation, or temporal logics without a notion of stuttering (cf. §11).

HOL-SUM seems to be a promising mixture of algebraic lists and functions at first sight. However, it turned out that notions from domain theory are needed to relate the finite and infinite representations. Therefore it appears to be preferably to use HOL-LCF instead, where domain-theoretic notions are provided in a more general fashion. This has been realized by Agerholm [Age94b], but he goes only half the way. Domain theory deals with both recursive *functions* and recursive *domains*. Agerholm's inconsequence lies in the fact that he employs domain theory to define recursive functions on sequences, but represents the sequences themselves not by recursive domains. Actually, his motivation is obvious: he wants to circumvent the axiomatic, externally justified foundation underlying the domain package of Isabelle/HOLCF. As a strictly definitional embedding of recursive domains is not feasable in simply typed higher-order logic[2], he switched to a HOL-SUM related sequence model, which, however, turned out to be inappropriate in practice.

7.7.2 Comparing HOL-LCF and PVS-CO

Let us first mention an implementation-specific point. Both approaches HOL-LCF and PVS-CO are essentially based on a single axiom. For PVS-CO it determines the property of being a final coalgebra, whereas in HOL-LCF it ensures that initial algebras and final coalgebras coincide (see [AJ94, Fio96]). Interestingly, both approaches have alternatively been developed in a definitional way as well: PVS-CO by Paulson [Pau97] and HOL-LCF by Agerholm [Age94b]. But whereas Paulson succeeded in hiding the additional overhead efficiently, Agerholm had to turn to an unmanageable, HOL-SUM like approach, as the necessary limit construction cannot be formalized in simply typed higher-order logic.

In the following we compare algebraic and coalgebraic sequence models in general, independent of specific realizations in a theorem prover. We discuss first (co)recursion and then (co)induction.

Concerning definition principles, recursion in LCF is restricted to computable functions, whereas corecursion can describe non-computable functions as well. On the other hand, corecursion cannot handle divergent computations. Furthermore, the fixpoint constructions in LCF allow arbitrary recursion, whereas the definition scheme offered by corec permits only “primitive” corecursion. Furthermore, corecursive definitions seem to be less intuitive than recursive ones. The reason may be that functions are more naturally specified by describing how they react on inputs than by determining the associated inputs to observed outputs. In addition, corecursive definitions usually — compare the experiences by Paulson [Pau97] — involve more case distinctions than recursive ones.

Concerning proof principles, structural induction in LCF is more powerful than coinduction. Whereas bisimulation is essentially restricted to the proof of sequence equalities,

[2]However, remark Agerholm's definitional embedding of recursive domains [Age94a] by means of the inverse limit construction in HOL-ST [Gor94], a version of Gordon's HOL system that supports a ZF-like set theory together with higher-order logic.

induction can handle arbitrary predicates, subject to the syntactical admissibility criterion. As this criterion holds for every equation of sequences, admissibility offers more power than bisimulation. Invariants provide an alternative proof rule in the coalgebraic setting, but are restricted to deal with safety properties.

Furthermore, the experiences made with HOL-LCF and PVS-Co suggest that structural induction offers a greater potential for proof automation than bisimulation does. This is mainly due to the higher amount of case distinctions in corecursive definitions and the automated admissibility check in HOL-LCF.

Finally, we have shown in the previous chapter that coinduction can be used in HOL-LCF as well by providing an infrastructure that makes coinduction accessible for recursive definitions. On the other hand, the structural induction rule has been derived in PVS-Co, but it is of little use in this framework as there are no means to tackle the generated admissibility obligations by simple syntactic criteria. In particular they cannot be reduced to continuity, as computability is not a general prerequisite in this setting. Therefore, in HOL-LCF the complementary proof principle is of advantage, in PVS-Co it is not.

To conclude, HOL-LCF seems to be the reasonable choice. This applies in particular, if the problem in question incorporates a notion of computability. Only if this is not the case, PVS-Co might be considered an alternative.

Part III: I/O Automata in Isabelle

Chapter 8

Basic Theory of I/O Automata

In this chapter fair I/O automata are embedded into Isabelle, where system runs are formalized using the sequence model described in the previous two chapters. An important result from a methodological point of view is that the use of HOLCF is restricted to meta-theoretic arguments while actual refinement proofs are carried out in the simpler HOL. This is possible because of the interface between HOL and HOLCF described in §5.2. As a result we get a framework, which is both efficient for system verification and sufficiently expressive for meta-theoretic investigations.

8.1 Introduction

This chapter is the first in a series of chapters describing the formalization of I/O automata in Isabelle. First of all we explain the methodological background and give an overview of the entire developments within Isabelle.

The overall aim of the I/O automata embedding in Isabelle is twofold:

1. A tool environment for the analysis of I/O automata should be provided, which facilitates and (at least partially) automates the often tedious refinement and abstraction proofs on paper. Thus, the embedding should be efficient and easy to use in practice.

2. We want to verify the theory of I/O automata itself, i.e. prove meta-theoretic notions like the soundness of refinement concepts by setting them on a firm logical foundation.

Thus, the embedding should be powerful and allow sophisticated constructions in theory.

These requirements seem to be contradictory. We solve the problem by combining the two logics HOL and HOLCF in a tailored way using the interface methodology described in §5.2. Whereas the user of the tool environment employs the simpler logic HOL for reasoning about single steps of an automaton, meta-theoretic proofs concerning entire system runs may employ the more powerful logic HOLCF. The prerequisites for this methodology are treated in more detail in §8.5.

Below we give an overview of the Isabelle developments described in the next five chapters. Fig. 8.1 and Fig. 8.2 display the dependency graph of the Isabelle theories[1], which shows the interconnections between the associated sections of this text as well.

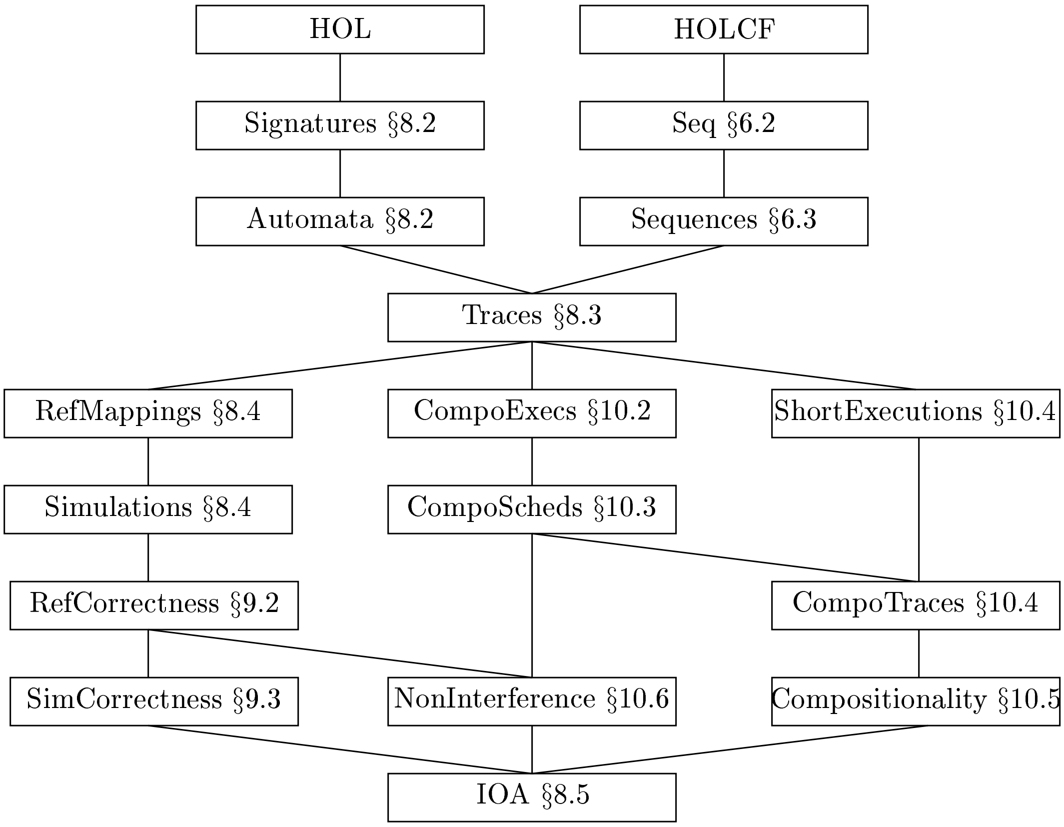

Figure 8.1: Isabelle Theory Structure for Lynch/Vaandrager I/O Automata

The present chapter deals with the basic notions of fair I/O automata. This includes the I/O automata definition itself (§8.2), runs of automata (§8.3), refinement notions (§8.4),

[1]The entire I/O automaton model is part of the Isabelle distribution and can be found under `http://www4.informatik.tu-muenchen.de/~isabelle/library/HOLCF/IOA/index.html`.

a description of the proof infrastructure and methodology (§8.5), and an evaluation of the resulting tool environment (§8.6). Thus, it offers a rather user-oriented view on the formalization.

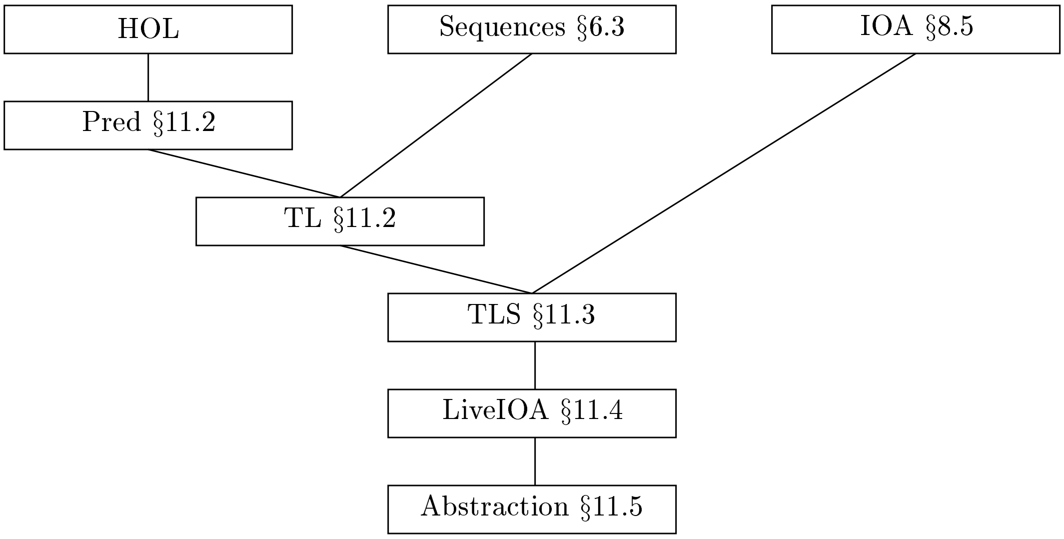

Figure 8.2: Isabelle Theory Structure for I/O Automata Extensions

The two subsequent chapters 9 and 10 deal with meta-theoretic notions by proving the soundness of (fair) refinement mappings (§9.2) and forward simulations (§9.3), compositionality for safe I/O automata (§10.2 – §10.5), and noninterference (§10.6).

Finally, chapter 11 deals with extensions to the theory of Lynch/Vaandrager [Lyn96, LV95] including temporal logics (§11.2 – §11.3), which are used to support live I/O automata (§11.4), and an abstraction theory (§11.5), which aims for an effective combination with model checkers.

8.2 Fair I/O Automata

This section presents the basic I/O automaton definition, operators to build more complex automata, and an invariant proof rule.

Definition 8.2.1 (Action Signatures)
An action signature is described by a triple of action sets of the following type:

$$(\alpha)\,\mathsf{signature} = (\alpha)\,\mathsf{set} \times (\alpha)\,\mathsf{set} \times (\alpha)\,\mathsf{set}$$

The first, second and third components of an action signature may be extracted by the functions inputs, $\mathsf{outputs}$, and $\mathsf{internals}$, respectively.

The following sets contain all actions, the externally visible actions, and the locally controllable actions, respectively.

$$\begin{array}{lcl} \mathsf{actions}(sig) & \equiv & \mathsf{inputs}(sig) \cup \mathsf{outputs}(sig) \cup \mathsf{internals}(sig) \\ \mathsf{externals}(sig) & \equiv & \mathsf{inputs}(sig) \cup \mathsf{outputs}(sig) \\ \mathsf{locals}(sig) & \equiv & \mathsf{outputs}(sig) \cup \mathsf{internals}(sig) \end{array}$$

Furthermore, for well-formed signatures the action sets have to be disjoint:

$$\begin{array}{lcl} \mathsf{is\text{-}sig}(sig) & \equiv & \mathsf{inputs}(sig) \cap \mathsf{outputs}(sig) \;\; = \{\} \wedge \\ & & \mathsf{inputs}(sig) \cap \mathsf{internals}(sig) \;\; = \{\} \wedge \\ & & \mathsf{outputs}(sig) \cap \mathsf{internals}(sig) = \{\} \end{array}$$

□

Lemmas proven about action signatures can be found in Appendix B.2.

Definition 8.2.2 (Type of Fair I/O Automata)
A fair I/O automaton with action type α and state type σ is defined as a tuple of type

$$(\alpha,\sigma)\,\mathsf{ioa} \;\;=\;\; (\alpha)\,\mathsf{signature} \times (\sigma)\,\mathsf{set} \times (\sigma \times \alpha \times \sigma)\,\mathsf{set} \times ((\alpha)\,\mathsf{set})\,\mathsf{set} \times ((\alpha)\,\mathsf{set})\,\mathsf{set}$$

where the members of the tuple are extracted by the projection functions sig-of, starts-of, trans-of, wfair-of, and sfair-of, repectively. □

Notation. Isabelle's syntax translation mechanism is used to write $s \stackrel{a}{\rightarrow}_A t$ for $(s,a,t) \in \mathsf{trans\text{-}of}(A)$. Furthermore the abbreviations $\mathsf{act}, \mathsf{ext}, \mathsf{int}, \mathsf{in}, \mathsf{out}$, and local are introduced for $\mathsf{actions} \circ \mathsf{sig\text{-}of}$, $\mathsf{externals} \circ \mathsf{sig\text{-}of}$, $\mathsf{internals} \circ \mathsf{sig\text{-}of}$, $\mathsf{inputs} \circ \mathsf{sig\text{-}of}$, $\mathsf{outputs} \circ \mathsf{sig\text{-}of}$, and $\mathsf{locals} \circ \mathsf{sig\text{-}of}$, respectively.

Definition 8.2.3 (Fair I/O Automata)
Requirements posed on safe I/O automata are expressed by the predicate is-safe-IOA. It demands that the first component be an action signature, the second be an non-empty set of start states and the third be an input-enabled state transition relation, whose actions stem from the action signature:

$$\begin{array}{rcl} \mathsf{is\text{-}sig\text{-}of}(A) & \equiv & \mathsf{is\text{-}sig}\ (\mathsf{sig\text{-}of}\ A) \\ \mathsf{is\text{-}starts\text{-}of}(A) & \equiv & \mathsf{starts\text{-}of}(A) \neq \{\} \\ \mathsf{is\text{-}trans\text{-}of}(A) & \equiv & \forall (s,a,t) \in \mathsf{trans\text{-}of}(A).\ a \in \mathsf{act}(A) \\ \mathsf{input\text{-}enabled}(A) & \equiv & \forall a \in \mathsf{in}(A).\ \forall s.\ \exists t.\ s \stackrel{a}{\rightarrow}_A t \\ \mathsf{is\text{-}safe\text{-}IOA}(A) & \equiv & \mathsf{is\text{-}sig\text{-}of}(A) \wedge \mathsf{is\text{-}starts\text{-}of}(A) \wedge \mathsf{is\text{-}trans\text{-}of}(A) \wedge \mathsf{input\text{-}enabled}(A) \end{array}$$

Additional requirements needed for fair I/O automata are expressed by the predicate is-fair-IOA. For its definition we have to introduce the notion of enabledness of an action set as in a state s first:

$$\begin{array}{lcl} \mathsf{enabled} & :: & (\alpha,\sigma)\,\mathsf{ioa} \rightarrow (\alpha)\,\mathsf{set} \rightarrow \sigma \rightarrow \mathsf{bool} \\ \mathsf{enabled}\ A\ as\ s & \equiv & \exists a \in as.\ \exists t.\ s \stackrel{a}{\rightarrow}_A t \end{array}$$

Then, $\mathsf{is\text{-}fair\text{-}IOA}$ demands that the fairness sets are input resistant subsets of the locally-controlled actions.

$$
\begin{array}{lcl}
\mathsf{input\text{-}resistant}, \mathsf{is\text{-}fair\text{-}IOA} & :: & (\alpha,\sigma)\,\mathsf{ioa} \to \mathsf{bool} \\
\mathsf{input\text{-}resistant}(A) & \equiv & \forall as \in \mathsf{sfair\text{-}of}(A).\ \forall s\ a\ t. \\
 & & \mathsf{reachable}\ A\ s \wedge \mathsf{reachable}\ A\ t \wedge \\
 & & a \in \mathsf{in}(A) \wedge \mathsf{enabled}\ A\ as\ s \wedge\ s \xrightarrow{a}_A t \\
 & & \Rightarrow \mathsf{enabled}\ A\ as\ t \\
\mathsf{is\text{-}fair\text{-}IOA}(A) & \equiv & (\forall as \in (\mathsf{wfair\text{-}of}(A) \cup \mathsf{sfair\text{-}of}(A)).\ as \subseteq \mathsf{local}(A)) \wedge \\
 & & \mathsf{input\text{-}resistant}(A)
\end{array}
$$

□

We have introduced constants for every subproperty of $\mathsf{is\text{-}safe\text{-}IOA}$ and $\mathsf{is\text{-}fair\text{-}IOA}$, as with this finer distinction it is possible to state precisely the assumptions needed for every meta-theoretic theorem to be proved in the sequel. For instance, compositionality does not need input-enabledness in the safe case, or the $\mathsf{is\text{-}starts\text{-}of}$ assumption is not needed anywhere in the meta-theoretic parts.

Interaction of I/O automata is specified by a parallel composition operator. Automata synchronize on their common actions and evolve independently on the others. Composition is defined only for *compatible* I/O automata. Compatibility requires that each action is under the control of at most one automaton.

Definition 8.2.4 (Parallel Composition)
Parallel Composition is defined as a binary operator $\|$ as follows:

$$
\begin{array}{lcl}
\| & :: & (\alpha,\sigma)\,\mathsf{ioa} \to (\alpha,\tau)\,\mathsf{ioa} \to (\alpha,\sigma\times\tau)\,\mathsf{ioa} \\
A \,\|\, B & \equiv & (\mathsf{sig\text{-}comp}\ (\mathsf{sig\text{-}of} A)\ (\mathsf{sig\text{-}of} B), \\
 & & \quad \{(u,v) \mid u \in \mathsf{starts\text{-}of}(A) \wedge v \in \mathsf{starts\text{-}of}(B)\}, \\
 & & \quad \{(s,a,t) \mid (a \in \mathsf{act}(A) \vee a \in \mathsf{act}(B)) \qquad \wedge \\
 & & \qquad\quad \textbf{if}\ a \in \mathsf{act}(A)\ \textbf{then}\ (\mathsf{fst}\ s) \xrightarrow{a}_A (\mathsf{fst}\ t) \\
 & & \qquad\qquad\qquad\qquad \textbf{else}\ (\mathsf{fst}\ s) = (\mathsf{fst}\ t) \quad \wedge \\
 & & \qquad\quad \textbf{if}\ a \in \mathsf{act}(B)\ \textbf{then}\ (\mathsf{snd}\ s) \xrightarrow{a}_B (\mathsf{snd}\ t) \\
 & & \qquad\qquad\qquad\qquad \textbf{else}\ (\mathsf{snd}\ s) = (\mathsf{snd}\ t)\}, \\
 & & \quad \mathsf{wfair\text{-}of}(A) \cup \mathsf{wfair\text{-}of}(B), \\
 & & \quad \mathsf{sfair\text{-}of}(A) \cup \mathsf{sfair\text{-}of}(B))
\end{array}
$$

where $\mathsf{sig\text{-}comp}$ defines the composition of signatures as follows:

$$
\begin{array}{lcl}
\mathsf{sig\text{-}comp} & :: & (\alpha)\,\mathsf{signature} \to (\alpha)\,\mathsf{signature} \to (\alpha)\,\mathsf{signature} \\
\mathsf{sig\text{-}comp}\ sig_A\ sig_B & \equiv & ((\mathsf{inputs}(sig_A) \cup \mathsf{inputs}(sig_B)) \setminus (\mathsf{outputs}(sig_A) \cup \mathsf{outputs}(sig_B)), \\
 & & \ \mathsf{outputs}(sig_A) \cup \mathsf{outputs}(sig_B), \\
 & & \ \mathsf{internals}(sig_A) \cup \mathsf{internals}(sig_B))
\end{array}
$$

Compatibility requires that each action is an output action of at most one I/O automaton and that internal action names are unique.

$$
\begin{array}{lcl}
\mathsf{compatible} & :: & (\alpha,\sigma)\,\mathsf{ioa} \to (\alpha,\tau)\,\mathsf{ioa} \to \mathsf{bool} \\
\mathsf{compatible}\ A\ B & \equiv & (\mathsf{out}(A) \cap \mathsf{out}(B) = \{\}) \wedge \\
 & & (\mathsf{act}(A) \cap \mathsf{int}(B) = \{\}) \wedge \\
 & & (\mathsf{act}(B) \cap \mathsf{int}(A) = \{\})
\end{array}
$$

□

There are several reasons for defining parallel composition as a binary operator and not by index sets as in §2. First, it is easier to reason about pairs than about indexed families. Second, a binary operator gives more flexibility in the type of states than an operator based on indexed families. Whereas binary composition has type

$$(\alpha,\sigma)\,\mathsf{ioa} \to (\alpha,\tau)\,\mathsf{ioa} \to (\alpha,\sigma\times\tau)\,\mathsf{ioa}$$

a version using β as an index set for automata families would have type

$$(\beta \to (\alpha,\sigma)\,\mathsf{ioa}) \to (\alpha,\beta\to\sigma)\,\mathsf{ioa}$$

Thus it requires that every member of the indexed set has the same type, i.e. all automata are defined over the same state and action space. This may be fine for replicating the same I/O automaton many times but awkward when composing a system from many different components. The binary operator, however, requires only a common action signature, but is flexible w.r.t. the type of the component states. Finally, restricting the parallel composition operator to finitely many automata preserves compatibility with the timed model, where composition of infinitely many live I/O automata is not possible.

Note that by the definition of sig-comp an action a modeling interprocess communication remains as an (external) output action of the composed automaton. Hiding of internal communication between subcomponents is provided by a separate operator:

Definition 8.2.5 (Hiding and Restriction)
Hiding a set of actions *acts* of an I/O automaton is accomplished by an operation on its action signature *sig*: All external actions in *acts* are changed to internal actions.

$$
\begin{array}{lcl}
\mathsf{hide\text{-}sig} & :: & (\alpha)\,\mathsf{signature} \to (\alpha)\,\mathsf{set} \to (\alpha)\,\mathsf{signature} \\
\mathsf{hide\text{-}sig}\ \mathit{sig}\ \mathit{acts} & \equiv & (\mathsf{inputs}(\mathit{sig}), \\
 & & \ \mathsf{outputs}(\mathit{sig}) \setminus \mathit{acts}, \\
 & & \ \mathsf{internals}(\mathit{sig}) \cup \mathit{acts})
\end{array}
$$

Hiding is canonically extended from signatures to automata:

$$
\begin{array}{lcl}
\mathsf{hide} & :: & (\alpha,\sigma)\,\mathsf{ioa} \to (\alpha)\,\mathsf{set} \to (\alpha,\sigma)\,\mathsf{ioa} \\
\mathsf{hide}\ A\ \mathit{acts} & \equiv & (\mathsf{restrict\text{-}sig}\ (\mathsf{sig\text{-}of} A)\ \mathit{acts}, \\
 & & \ \mathsf{starts\text{-}of}(A), \mathsf{trans\text{-}of}(A) \\
 & & \ \mathsf{wfair\text{-}of}(A), \mathsf{sfair\text{-}of}(A))
\end{array}
$$

Similarly, a dual operator allows to restrict the external interface of a signature *sig* to a set of actions *acts* by changing all external actions not in *acts* to internal actions:

$$\begin{array}{lcl}
\textsf{restrict-sig} & :: & (\alpha)\,\textsf{signature} \to (\alpha)\,\textsf{set} \to (\alpha)\,\textsf{signature} \\
\textsf{restrict-sig}\ sig\ acts & \equiv & (\textsf{inputs}(sig), \\
& & \ \textsf{outputs}(sig) \cap acts, \\
& & \ \textsf{internals}(sig) \cup (\textsf{externals}(sig) \setminus acts))
\end{array}$$

The extension from signatures to automata is the same as above:

$$\begin{array}{lcl}
\textsf{restrict} & :: & (\alpha,\sigma)\,\textsf{ioa} \to (\alpha)\,\textsf{set} \to (\alpha,\sigma)\,\textsf{ioa} \\
\textsf{restrict}\ A\ acts & \equiv & (\textsf{restrict-sig}\ (\textsf{sig-of}A)\ acts, \\
& & \ \textsf{starts-of}(A), \textsf{trans-of}(A) \\
& & \ \textsf{wfair-of}(A), \textsf{sfair-of}(A))
\end{array}$$

□

Recall that the parallel composition operator requires that each process to be composed must already be defined over all actions in the composition, even those not in its own action signature. This is feasible for a system built from a fixed number of processes but precludes the reuse of generic components. The latter can be achieved with the following operation for *renaming* actions.

Definition 8.2.6 (Renaming)
A mapping $f :: \beta \to (\alpha)\textsf{option}$ induces a renaming of every set S of type α in the following way:

$$\begin{array}{lcl}
\textsf{rename-set} & :: & (\alpha)\,\textsf{set} \to (\beta \to (\alpha)\,\textsf{option}) \to (\beta)\,\textsf{set} \\
\textsf{rename-set}\ S\ f & \equiv & \{b \mid \exists a.\, f(b) = \textsf{Some}(a) \wedge a \in S\}
\end{array}$$

Renaming of automata extends rename-set to automata as follows:

$$\begin{array}{lcl}
\textsf{rename} & :: & (\alpha,\sigma)\,\textsf{ioa} \to (\beta \to (\alpha)\,\textsf{option}) \to (\beta,\sigma)\,\textsf{ioa} \\
\textsf{rename}\ A\ f & \equiv & ((\textsf{rename-set}\ (\textsf{in}\ A), \textsf{rename-set}\ (\textsf{out}\ A), \textsf{rename-set}\ (\textsf{int}\ A)), \\
& & \ \textsf{starts-of}(A), \\
& & \ \{(s,b,t) \mid \exists a.\, f(b) = \textsf{Some}(a) \wedge\ s \stackrel{a}{\to}_A t\ \}, \\
& & \ \{\textsf{rename-set}\ acts\ f\ |_{acts}\ acts \in \textsf{wfair-of}(A)\}, \\
& & \ \{\textsf{rename-set}\ acts\ f\ |_{acts}\ acts \in \textsf{sfair-of}(A)\})
\end{array}$$

□

Action renaming can also be used to avoid name clashes that lead to incompatibilities of I/O automata to be composed. In contrast to the literature (e.g. [LT87], see §2 as well) the action renaming function f has been defined with type $\beta \to (\alpha)$ option instead of

$\alpha \to \beta$. Defining f in the inverse direction has the advantage that the otherwise necessary injectivity requirement can be omitted. For example, in our setting the theorem

$$\textsf{is-sig}(A) \Rightarrow \textsf{is-sig}(\textsf{rename}\ A\ f)$$

holds without further requirements on f. Therefore, every renamed I/O automata is again an I/O automata for an arbitrary renaming function f. This considerably facilitates reasoning about complex automata built by composition, hiding, and renaming. The price for that simplification is merely that the meta-theory does not exclude already that actions of the original automaton may be ignored in the renamed automaton. However, this does not affect any meta-theoretic results and, in addition, may even sometimes be intended by the specifier.

Definition 8.2.7 (Reachability and Invariants)
The set of reachable states of an I/O automaton A is defined inductively as the least set of states satisfying the following two rules:

$$\textbf{inductive}\quad \frac{s \in \textsf{starts-of}(A)}{s \in \textsf{reachable}(A)}\ (\textit{reachable}\text{-}0) \qquad \frac{s \in \textsf{reachable}(A) \quad s \stackrel{a}{\to}_A t}{t \in \textsf{reachable}(A)}\ (\textit{reachable}\text{-}n)$$

Isabelle's syntax translation mechanism is used to write $\textsf{reachable}\ A\ s$ for $s \in \textsf{reachable}\ A$.

A state predicate $P :: \sigma \to \textsf{bool}$ is called an invariant of an I/O automaton A if it holds for all reachable states:

$$\textsf{invariant}\ A\ P \quad \equiv \quad (\forall s.\ \textsf{reachable}\ A\ s \Rightarrow P(s))$$

□

The definition of $\textsf{reachable}$ differs from the standard one in the literature [GSSL93]: *A state s is reachable iff there is a finite execution that ends in s.* The reason is that our inductive definition is tailored for inductive proofs on sequences, namely by talking about steps rather than about entire sequences. Moreover, it allows to derive the following important induction theorem for invariants directly:

Theorem 8.2.8 (Invariant Rules)
A state predicate is an invariant iff it holds for every start state and is preserved by every reachable step:

$$\frac{(\forall s.\ s \in \textsf{starts-of}(A) \Rightarrow P(s)) \quad (\forall s\ a\ t.\ \textsf{reachable}\ A\ s \wedge P(s) \wedge\ s \stackrel{a}{\to}_A t\ \Rightarrow P(t))}{\textsf{invariant}\ A\ P}\ (\textit{invI})$$

An invariant holds for every reachable state:

$$\frac{\textsf{invariant}\ A\ P \quad \textsf{reachable}\ A\ s}{P(s)}\ (\textit{invE})$$

Proof.
Rule (*invI*) is easily proved by the induction rule for reachable which is generated by Isabelle for every inductive definition. The second rule (*invE*) follows immediately from the definition of an invariant. □

Lemmas about these basic I/O automata notions can be found in Appendix B.2.

Isabelle Syntax of I/O Automata. So far, we defined I/O automata as parameterized tuples over arbitrary state and action types. In the sequel we will describe how concrete state spaces and actions are represented in Isabelle.

Actions are defined easily using Isabelle's **datatype** construct. States are not represented by variables, but as tuples, where each component represents a variable. Selector functions are introduced, whose names are identical to the variable names in the informal description. This approach conforms with [Lyn96].

Let us illustrate this format with the simple example of a buffer. The buffer *Buf* is modeled by a variable *queue* of type (*bool*)*list*, which is initially empty. Actions and transitions are given in the usual precondition/effect style as:

input $S(m), m \in bool$	**output** $R(m), m \in bool$
post: $queue := queue@[m]$	**pre**: $queue = m : rst$
	post: $queue := rst$

In Isabelle the action type of *Buf* is defined as **datatype** $action = S(bool) \mid R(bool)$. The automaton *Buf* is then described as follows, where *queue* is the identity on the (trivial) tuple of type (*bool*)*list*, and transitions are represented in a set comprehension format.

$$
\begin{array}{lcl}
Buf & \equiv & (Buf\text{-}sig, \{[\,]\}, Buf\text{-}trans, \{\}, \{\}) \\
Buf\text{-}sig & \equiv & (\bigcup_{m \in bool}\{S(m)\}, \bigcup_{m \in bool}\{R(m)\}, \{\}) \\
Buf\text{-}trans & \equiv & \{(s, a, s') \mid \textbf{case } a \textbf{ of} \\
& & \qquad\qquad\quad S(m) \;\Rightarrow\; queue(s') = queue(s)@[m] \\
& & \qquad\qquad \mid\; R(m) \;\Rightarrow\; queue(s) = m : rst \wedge queue(s') = rst\}
\end{array}
$$

An automatic translation of the precondition/effect style into the set comprehension format is trivially possible, but not the focus of our research.

8.3 Fair Traces

This section contains definitions and theorems concerning the behaviors of I/O automata — executions, schedules, and traces — and, in particular, the definition of implementation relations between I/O automata.

Executions are embedded in Isabelle using the sequence model defined in §6.3.

Definition 8.3.1 (Type of Executions)
Executions of an I/O automaton A are modeled by a pair of a start state and a sequence of action/state pairs:

$$(\alpha,\sigma)\,\mathsf{execution} = \sigma \times (\alpha \times \sigma)\,\mathsf{sequence}$$

□

The additional start state is necessary because otherwise the first transition starts from an unknown state and, more importantly, because a state has to be associated with the empty execution. The latter is required for simulation steps, where the empty execution is used to simulate an internal step of the implementation. Here it would not be possible with an empty execution without state (i.e. nil) to keep track of the connection to the state of the preceding simulation step.

Several alternative formalizations of executions have been studied as well, the most important of them will briefly be discussed in the following:

- At first glance, a sequence of transition triples seems to be more natural:

  $$(\alpha,\sigma)\,\mathsf{execution} = \sigma \times (\sigma \times \alpha \times \sigma)\,\mathsf{sequence}$$

  The advantage is that $(\sigma \times \alpha \times \sigma)$ triples are already part of the automaton definition. But an important drawback is the redundancy of this representation. It has to be guaranteed that the transitions coincide on the intermediate states: the state/sequence pair $(s_1, [(s_2, a_1, s_3), (s_4, a_2, s_5)!])$, for example, would be an execution only if $s_1 = s_2$ and $s_3 = s_4$.

- In [NS95] a pair of sequences, one for actions and one for states, has been chosen, which in our setting would correspond to:

  $$(\alpha,\sigma)\,\mathsf{execution} = (\sigma)\,\mathsf{sequence} \times (\alpha)\,\mathsf{sequence}$$

  This is a natural representation for sequences as functions on natural numbers, but in our setting it is rather inconvenient, as it allows the two sequences to be of different length, which has to be ruled out explicitly. This, however, contradicts the philosophy in our approach to avoid indexes as much as possible. For an argumentation against a formalization of sequences as functions, see the discussion in §7.7.

- Of course, it is also possible to stay as near as possible to the informal description: An alternating sequence of states and actions could be formalized as

  $$(\alpha,\sigma)\,\mathsf{execution} = ((\alpha,\sigma)\,\mathsf{element})\,\mathsf{sequence}$$

  where elements are defined by

  $$\textbf{datatype}\quad (\alpha,\sigma)\,\mathsf{element} = \mathsf{act}(\alpha) \mid \mathsf{state}(\sigma)$$

A further predicate has to guarantee that states and actions really alternate. Such a predicate has to express three requirements, namely that the sequence begins with a state and ends with a state if it is finite, and that every action (state) is followed by an state (action). In our solution, however, all these requirements are coded already into the type. To reduce the number of requirements one could instead determine states or actions by checking if the corresponding indexes are odd or even. This would again introduce indexes, which is not adequate in our setting. Furthermore, note that in both cases additional reasoning is needed to show that the predicate holds after a concatenation. All in all, although this solution is nearest to the set theoretic characterization, it turns out to be the most complicated in our typed setting.

In contrast to the preceding three solutions, our formalization has the following advantages:

- Most of what it means to be an execution is already coded into the type.
- There is no redundancy in the representation, every element is represented once.
- The representation is tailored for induction proofs instead of index reasoning.
- The definition of the concatenation of two executions $exec_1$ and $exec_2$ is natural, as eliminating the duplicate occurrence of the first state of $exec_2$ is just pair projection:

$$exec_1 \oplus_{\mathsf{ex}} exec_2 \quad \equiv \quad (\mathsf{fst}\ exec_1, (\mathsf{snd}\ exec_1) \oplus (\mathsf{snd}\ exec_2))$$

- The projection of an execution of $A \parallel B$ to A means deleting every action not in $\mathsf{act}(A)$ together with its following state. This can be defined in a natural way, as actions are already paired with their following state.

Notation. In the sequel $exec$ stands for variables of type $(\alpha, \sigma)\,\mathsf{execution}$, whereas s and ex denote the the start state and the sequence of action/state pair, if the execution is explicitly written as a pair.

Definition 8.3.2 (Executions)
The predicate is-exec-frag identifies sequences of type (α, σ) execution that are a proper execution fragment of an I/O automaton A:

$$\begin{array}{lcl} \mathsf{is\text{-}exec\text{-}frag} & :: & (\alpha,\sigma)\,\mathsf{ioa} \rightarrow (\alpha,\sigma)\,\mathsf{execution} \rightarrow \mathsf{bool} \\ \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, ex) & \equiv & \mathsf{is\text{-}exec\text{-}frag_c}\ A\ \text{`}ex\ s \neq \mathsf{FF} \end{array}$$

It is realized by a computable function, the continuous predicate

$$\mathsf{is\text{-}exec\text{-}frag_c} \;::\; (\alpha,\sigma)\,\mathsf{ioa} \rightarrow (\alpha \times \sigma)\,\mathsf{sequence} \rightarrow_c \sigma \rightarrow \mathsf{tr}$$

which "runs down" the sequence checking if all of its transitions are transitions of the automaton A. The predicate is-exec-frag A (s, ex) is true if the operation terminates and returns TT (for finite ex) or if the operation does not terminate, thus returning $\perp$ (for infinite ex). The operation $\mathsf{is\text{-}exec\text{-}frag_c}$ is defined as a fixpoint. The following equations follow immediately from its definition:

$$\begin{array}{lcl}
\mathsf{is\text{-}exec\text{-}frag_c}\ A\ \text{`}\perp s & = & \perp \\
\mathsf{is\text{-}exec\text{-}frag_c}\ A\ \text{`}\mathsf{nil}\ s & = & \mathsf{TT} \\
\mathsf{is\text{-}exec\text{-}frag_c}\ A\ \text{`}((a,t)\hat{\ }ex)\ s & = & (\mathsf{Def}\ \ s \xrightarrow{a}_A t\ )\ \mathsf{andalso}\ \mathsf{is\text{-}exec\text{-}frag_c}\ A\ \text{`}ex\ t
\end{array}$$

Executions are execution fragments beginning with a start state.

$$\mathsf{executions}(A) \ \equiv\ \{(s,ex).\ s \in \mathsf{starts\text{-}of}(A) \wedge \mathsf{is\text{-}exec\text{-}frag}\ A\ (s,ex)\}$$

□

Corollary 8.3.3
The following equations have been derived for $\mathsf{is\text{-}exec\text{-}frag}$ using those for $\mathsf{is\text{-}exec\text{-}frag_c}$:

$$\begin{array}{lcl}
\mathsf{is\text{-}exec\text{-}frag}\ A\ (s,\perp) & = & \mathsf{True} \\
\mathsf{is\text{-}exec\text{-}frag}\ A\ (s,\mathsf{nil}) & = & \mathsf{True} \\
\mathsf{is\text{-}exec\text{-}frag}\ A\ (s,(a,t)\hat{\ }ex) & = & s \xrightarrow{a}_A t\ \wedge \mathsf{is\text{-}exec\text{-}frag}\ A\ (t,ex)
\end{array}$$

See §6.3 for the schema to prove such recursive rules for boolean predicates.

Differently from most accounts on I/O automata and our introduction in §2 we do not directly define traces as originating from executions, but introduce *schedules* in an intermediate step. This contributes to a more structured and simpler proof of the compositionality result. With this decision we follow [LT87], the only article where compositionality is dealt with to some extent. Alternatively, we provide an operator that generates traces directly from executions.

Definition 8.3.4 (Schedules and Traces)
The actions of an execution may be extracted by projecting every pair in the sequence onto the action component.

$$\begin{array}{lcl}
\mathsf{Filter\text{-}act} & :: & (\alpha \times \sigma)\,\mathsf{sequence} \rightarrow_c \alpha\,\mathsf{sequence} \\
\mathsf{Filter\text{-}act} & \equiv & \mathsf{Map\ fst}
\end{array}$$

The resulting subsequence of actions is called a *schedule* of A.

$$\mathsf{schedules}(A) \ \equiv\ \{\mathsf{Filter\text{-}act}\ \text{`}ex\ |_{ex}\ \exists s.\ (s,ex) \in \mathsf{executions}\ A\}$$

Traces represent the visible behaviour of an I/O automaton A. They are obtained by removing all internal actions from the schedules of A.

$$\mathsf{traces}(A) \ \equiv\ \{\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ A)\ \text{`}sch\ |_{sch}\ sch \in \mathsf{schedules}\ A\}$$

The operation $\mathsf{mk\text{-}trace}$ transforms an execution of an I/O automaton A into the corresponding trace.

$$\begin{array}{lcl} \mathsf{mk\text{-}trace} & :: & (\alpha,\sigma)\,\mathsf{ioa} \to (\alpha\times\sigma)\,\mathsf{sequence} \to_c \alpha\,\mathsf{sequence} \\ \mathsf{mk\text{-}trace}(A) & \equiv & \Lambda ex.\ \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ A)\ \text{`}(\mathsf{Filter\text{-}act}\ \text{`}ex) \end{array}$$

□

Corollary 8.3.5
The following rewrite rules for $\mathsf{mk\text{-}trace}$ follow immediately from those for Filter and Map:

$$\begin{array}{lcl} \mathsf{mk\text{-}trace}\ A\ \text{`}\bot & = & \bot \\ \mathsf{mk\text{-}trace}\ A\ \text{`}\mathsf{nil} & = & \mathsf{nil} \\ \mathsf{mk\text{-}trace}\ A\ \text{`}(a,t)\hat{}\,ex & = & \textbf{if}\ a \in \mathsf{ext}\ A\ \textbf{then}\ a\hat{}\,\mathsf{mk\text{-}trace}\ A\ \text{`}ex \\ & & \qquad\qquad\qquad\ \ \textbf{else}\quad \mathsf{mk\text{-}trace}\ A\ \text{`}ex \end{array}$$

Lemma 8.3.6 (Alternative Characterization of Traces)
Using $\mathsf{mk\text{-}trace}$ the set of traces of an I/O automaton A is characterized as:

$$\mathsf{traces}(A) = \{\mathsf{mk\text{-}trace}\ A\ \text{`}ex\ |_{ex}\ \exists s.\ (s, ex) \in \mathsf{executions}\ A\}$$

Proof.
Trivial, but not fully automatic due to the $\exists$ in the definition of $|_x$. □

To formalize the notion of fair traces, which follows next, we need to talk about the last state of an execution. Obviously, there is no last state of an infinite execution. However, a last state will only be needed when dealing with finite executions. Therefore an arbitrary default definition is chosen for the partial and infinite case.

Definition 8.3.7 (Last State)
The last state of an execution (s, ex) is defined to be the last state of ex if ex is finite, otherwise it is the first state s.

$$\begin{array}{lcl} \mathsf{last\text{-}state} & :: & (\alpha,\sigma)\,\mathsf{execution} \to \sigma \\ \mathsf{last\text{-}state}\ (s, ex) & \equiv & \textbf{case}\ \mathsf{Last}\ \text{`}ex\ \textbf{of} \\ & & \quad\ \ \bot \quad\ \ \Rightarrow\ s \\ & & \quad\mid \mathsf{Def}\ at \ \Rightarrow\ \mathsf{snd}(at) \end{array}$$

□

Although it makes no sense to speak about the continuity of $\mathsf{last\text{-}state}$ because σ is of class term, every function that refers to $\mathsf{last\text{-}state}$ (and has a codomain with a partial order defined on it), will not be continuous in ex.

Corollary 8.3.8
For the finite case, $\mathsf{last\text{-}state}$ can be characterized inductively:

$$\begin{array}{rrcl} & \mathsf{last\text{-}state}\ (s, \mathsf{nil}) & = & s \\ \mathsf{Finite}(ex) \Rightarrow & \mathsf{last\text{-}state}\ (s, (a,t)\hat{\ } ex) & = & \mathsf{last\text{-}state}\ (t, ex) \end{array}$$

Proof.
Case distinction: For $ex = \mathsf{nil}$ unfolding of the definition and rewriting yields the result. For $ex \neq \mathsf{nil}$ the lemma $\mathsf{Finite}(ex) \wedge ex \neq \mathsf{nil} \Rightarrow \mathsf{Last}`ex \neq \perp$ shows the desired property. □

Definition 8.3.9 (Fair Executions and Traces)
To define fairness we first have to capture the notion that a predicate P holds finitely or infinitely often in a sequence s. This is done by checking the finiteness of s after having filtered it w.r.t. P:

$$\begin{array}{lcl} \mathsf{inf\text{-}often}, \mathsf{fin\text{-}often} & :: & (\alpha \to \mathsf{bool}) \to (\alpha)\,\mathsf{sequence} \to \mathsf{bool} \\ \mathsf{inf\text{-}often} & \equiv & \mathsf{Infinite}\ (\mathsf{Filter}\ P\ `s) \\ \mathsf{fin\text{-}often} & \equiv & \neg \mathsf{inf\text{-}often}\ P\ s \end{array}$$

Fairness is divided into weak fairness and strong fairness , which are defined by the predicates $\mathsf{is\text{-}wfair}$ and $\mathsf{is\text{-}sfair}$ on executions (s, ex) of I/O automata A:

$$\begin{array}{lcl} \mathsf{is\text{-}wfair}, \mathsf{is\text{-}sfair}, \mathsf{is\text{-}fair} & :: & (\alpha,\sigma)\,\mathsf{ioa} \to (\alpha,\sigma)\,\mathsf{execution} \to \mathsf{bool} \\ \mathsf{is\text{-}wfair}\ A\ (s, ex) & \equiv & \forall as \in \mathsf{wfair\text{-}of}(A). \\ & & \quad \mathbf{if}\ \mathsf{Finite}(s)\ \mathbf{then}\ \neg \mathsf{enabled}\ A\ as\ (\mathsf{last\text{-}state}\ (s, ex)) \\ & & \quad\quad\quad \mathbf{else}\ (\mathsf{inf\text{-}often}\ (\lambda(a,t).\ a \in as)\ ex\ \vee \\ & & \quad\quad\quad\quad \mathsf{inf\text{-}often}\ (\lambda(a,t).\ \neg \mathsf{enabled}\ A\ as\ ex)\ ex) \\ \mathsf{is\text{-}sfair}\ A\ (s, ex) & \equiv & \forall as \in \mathsf{sfair\text{-}of}(A). \\ & & \quad \mathbf{if}\ \mathsf{Finite}(s)\ \mathbf{then}\ \neg \mathsf{enabled}\ A\ as\ (\mathsf{last\text{-}state}\ (s, ex)) \\ & & \quad\quad\quad \mathbf{else}\ (\mathsf{inf\text{-}often}\ (\lambda(a,t).\ a \in as)\ ex\ \vee \\ & & \quad\quad\quad\quad \mathsf{fin\text{-}often}\ (\lambda(a,t).\ \mathsf{enabled}\ A\ as\ ex)\ ex) \\ \mathsf{is\text{-}fair}\ A\ exec & \equiv & \mathsf{is\text{-}wfair}\ A\ exec \wedge \mathsf{is\text{-}sfair}\ A\ exec \end{array}$$

Then, the set of fair executions can be characterized as those executions that admissibilitysatisfy $\mathsf{is\text{-}fair}$, and the set of fair traces is obtained by the $\mathsf{mk\text{-}trace}$ operator as in the safe case:

$$\begin{array}{lcl} \mathsf{fair\text{-}executions}(A) & \equiv & \{exec.\ exec \in \mathsf{executions}(A) \wedge \mathsf{is\text{-}fair}\ A\ exec\} \\ \mathsf{fair\text{-}traces}(A) & \equiv & \{\mathsf{mk\text{-}trace}\ A\ `(\mathsf{snd}\ exec)\ |_{exec}\ ex \in \mathsf{fair\text{-}executions}(A)\} \end{array}$$

□

As examples, two lemmas concerning behaviours of I/O automata are shown below.

Lemma 8.3.10
Execution fragments are prefix closed w.r.t. the two prefix notions $\sqsubseteq$ and $\ll$.

$$\begin{array}{lcl} \textsf{is-exec-frag}\ A\ (s, ex_2) \wedge ex_1 \sqsubseteq ex_2 & \Rightarrow & \textsf{is-exec-frag}\ A\ (s, ex_1) \\ \textsf{is-exec-frag}\ A\ (s, ex_2) \wedge ex_1 \ll ex_2 & \Rightarrow & \textsf{is-exec-frag}\ A\ (s, ex_1) \end{array}$$

Proof.
Both propositions are essentially proved by structural induction on ex_2 and subsequent case splitting of ex_1 (for $\perp$, nil, $x\hat{\ }xs$). Note that for the first proposition the ex_1 and ex_2 could not be interchanged, as otherwise the automatic check would fail. $\square$

Lemma 8.3.11
Actions of executions, schedules, and traces originate from the signature of A.

$$\begin{array}{lcl} \textsf{is-trans-of}(A) \wedge exec \in \textsf{executions}(A) & \Rightarrow & \textsf{Forall}\ (\lambda a.\ a \in \textsf{act}\ A)\ `(\textsf{Filter-act}\ `(\textsf{snd}\ exec)) \\ \textsf{is-trans-of}(A) \wedge sch \in \textsf{schedules}(A) & \Rightarrow & \textsf{Forall}\ (\lambda a.\ a \in \textsf{act}(A))\ `sch \\ tr \in \textsf{traces}(A) & \Rightarrow & \textsf{Forall}\ (\lambda a.\ a \in \textsf{act}(A))\ `tr \end{array}$$

Proof.
The first proposition is proved by structural induction on (snd $exec$). The second is directly implied by the first. The third proposition follows from the general sequence lemma $(\forall x.\ P(x) \Rightarrow Q(x)) \Rightarrow \textsf{Forall}\ Q\ (\textsf{Filter}\ P\ `xs)$ and from $a \in \textsf{ext}(A) \Rightarrow a \in \textsf{act}(A)$. Therefore it does not need the is-trans-of assumption. $\square$

In the previous section we have introduced invariants as a means to express state properties of an I/O automaton A. Sometimes it is more naturally to formulate properties of A as properties of its behaviours. Thus, we introduce, analogous to [Lyn96], sets of behaviours that describe such properties.

Definition 8.3.12 (Properties of Behaviours)
Execution (resp. *schedule* and *trace*) properties are pairs of an action signature and a set of executions (resp. schedules and traces) which comply with this signature.

$$\begin{array}{lcl} (\alpha, \sigma)\,\textsf{exec-prop} & = & (\alpha)\,\textsf{signature} \times ((\alpha, \sigma)\,\textsf{execution})\,\textsf{set} \\ (\alpha)\,\textsf{sched-prop} & = & (\alpha)\,\textsf{signature} \times ((\alpha)\,\textsf{sequence})\,\textsf{set} \\ (\alpha)\,\textsf{trace-prop} & = & (\alpha)\,\textsf{signature} \times ((\alpha)\,\textsf{sequence})\,\textsf{set} \end{array}$$

Every I/O automaton A induces its canonical execution, schedule, and trace property.

$$\begin{array}{lcl} \textsf{Execs}(A) & \equiv & (\textsf{sig-of}(A), \textsf{executions}(A)) \\ \textsf{Scheds}(A) & \equiv & (\textsf{sig-of}(A), \textsf{schedules}(A)) \\ \textsf{Traces}(A) & \equiv & (\textsf{sig-of}(A), \textsf{traces}(A)) \end{array}$$

$\square$

In §10 composition operators shall be defined for each type of behaviour property. This will allow us to formulate compositionality results in a succinct way.

Definition 8.3.13 (Safe and Fair Implementation)
Safe and fair implementation are defined via trace inclusion and fair trace inclusion, respectively. Furthermore, the external actions have to be the same.

$$\begin{array}{lcl} C \preceq_S A & \equiv & \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A) \wedge \\ & & \mathsf{traces}(C) \subseteq \mathsf{traces}(A) \\ C \preceq_F A & \equiv & \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A) \wedge \\ & & \mathsf{fair\text{-}traces}(C) \subseteq \mathsf{fair\text{-}traces}(A) \end{array}$$

□

Note that $\mathsf{ext}(C) = \mathsf{ext}(A)$ would not be sufficient: it could be the case that input and output actions are exactly interchanged. This rather pathological case has to be excluded.

8.4 Refinement Notions

In our I/O automata formalization implementation relations between automata are shown by the use of simulations or by the simpler refinement mappings (cf. §2). A refinement mapping f is a function from the state space of a concrete automaton C to the state space of a more abstract automaton A, which has to fulfill two requirements. First, start states of C have to be mapped to start states of A. Second, for every step $s \xrightarrow{a}_C t$ of the automaton C there has to be a corresponding move of A, i.e. a finite execution fragment with first state $f(s)$, last state $f(t)$, and external behavior a. This property of f is depicted in Figure 8.3. Simulations use a relation instead of a function to relate concrete and abstract states.

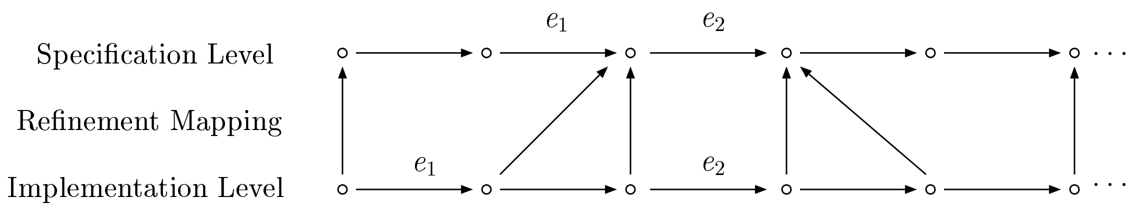

Figure 8.3: Refinement mapping: e_i are external actions, internal actions are omitted.

To formalize the notion of a simulation we first need the definition of a move:

Definition 8.4.1 (Moves)
A move (s, ex) from a state s to a state t with external behavior a is characterized by the

predicate $\mathsf{is\text{-}move}\ A\ ex\ (s, a, t)$:

$$
\begin{array}{lcl}
\mathsf{is\text{-}move} & :: & (\alpha,\sigma)\,\mathsf{ioa} \rightarrow (\alpha\times\sigma)\,\mathsf{sequence} \rightarrow (\sigma\times\alpha\times\sigma) \rightarrow \mathsf{bool} \\
\mathsf{is\text{-}move}\ A\ ex\ (s,a,t) & \equiv & \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, ex) \wedge \mathsf{Finite}(ex) \wedge \\
 & & \mathsf{last\text{-}state}\ (s, ex) = t \wedge \\
 & & \mathsf{mk\text{-}trace}\ A\ \text{`}ex = (\mathbf{if}\ a \in \mathsf{ext}(A)\ \mathbf{then}\ [a!]\ \mathbf{else}\ \mathsf{nil})
\end{array}
$$

□

By defining $\mathsf{is\text{-}move}$ over sequences of type $(\alpha\times\sigma)\mathsf{sequence}$ and not over executions of type $(\alpha,\sigma)\,\mathsf{execution}$ we can skip the requirement on the first state, as $\mathsf{fst}\ (s, ex) = s$ is trivially fulfilled. See §9.2 for a further advantage of defining $\mathsf{is\text{-}move}$ this way.

Definition 8.4.2 (Simulations and Refinement Mappings)
Forward simulations are defined by the predicate $\mathsf{is\text{-}simulation}$:

$$
\begin{array}{lcl}
\mathsf{is\text{-}simulation} & :: & (\sigma_1\times\sigma_2)\,\mathsf{set} \rightarrow (\alpha,\sigma_1)\,\mathsf{ioa} \rightarrow (\alpha,\sigma_2)\,\mathsf{ioa} \rightarrow \mathsf{bool} \\
\mathsf{is\text{-}simulation}\ R\ C\ A & \equiv & (\forall s_0 \in \mathsf{starts\text{-}of}(C).\ R[s_0] \cap \mathsf{starts\text{-}of}(A) \neq \{\}) \wedge \\
 & & (\forall s\ s'\ t\ a. \mathsf{reachable}\ C\ s \wedge\ s \stackrel{a}{\rightarrow}_C t\ \wedge (s, s') \in R \\
 & & \qquad \Rightarrow \exists t'\ ex.\ (t, t') \in R \wedge \mathsf{is\text{-}move}\ A\ ex\ (s', a, t')\,)
\end{array}
$$

Thus, a corresponding move from a given state is postulated. Backward simulations postulate a corresponding move towards a given state.

$$
\begin{array}{lcl}
\mathsf{is\text{-}back\text{-}simulation} & :: & (\sigma_1\times\sigma_2)\,\mathsf{set} \rightarrow (\alpha,\sigma_1)\,\mathsf{ioa} \rightarrow (\alpha,\sigma_2)\,\mathsf{ioa} \rightarrow \mathsf{bool} \\
\mathsf{is\text{-}back\text{-}simulation}\ R\ C\ A & \equiv & (\forall s_0 \in \mathsf{starts\text{-}of}(C).\ R[s_0] \subseteq \mathsf{starts\text{-}of}(A) \wedge \\
 & & (\forall s\ s'\ t\ a. \mathsf{reachable}\ C\ s \wedge\ s \stackrel{a}{\rightarrow}_C t\ \wedge (t, t') \in R \\
 & & \qquad \Rightarrow \exists s'\ ex.\ (s, s') \in R \wedge \mathsf{is\text{-}move}\ A\ ex\ (s', a, t')\,)
\end{array}
$$

Refinement mappings are defined by the predicate $\mathsf{is\text{-}ref\text{-}map}$:

$$
\begin{array}{lcl}
\mathsf{is\text{-}ref\text{-}map} & :: & (\sigma_1 \rightarrow \sigma_2) \rightarrow (\alpha,\sigma_1)\,\mathsf{ioa} \rightarrow (\alpha,\sigma_2)\,\mathsf{ioa} \rightarrow \mathsf{bool} \\
\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A & \equiv & (\forall s_0 \in \mathsf{starts\text{-}of}(C).\ f(s_0) \in \mathsf{starts\text{-}of}(A)) \wedge \\
 & & (\forall s\ t\ a. \mathsf{reachable}\ C\ s \wedge\ s \stackrel{a}{\rightarrow}_C t \\
 & & \qquad \Rightarrow \exists ex.\ \mathsf{is\text{-}move}\ A\ ex\ (f\ s, a, f\ t)\,)
\end{array}
$$

□

In several practical cases a weaker notion of refinement suffices to establish the implementation relation between automata. Therefore *weak refinement mappings* are introduced. They allow to simulate a step of C not by an execution fragment, but by at most a single step of A only. Two corresponding executions induced by a weak refinement mapping are depicted exemplarily in Fig. 8.4.

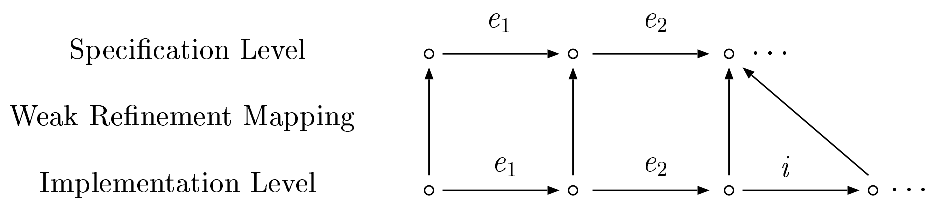

Figure 8.4: Weak Refinement Mapping: e_i are external actions, i is internal.

Definition 8.4.3 (Weak Refinement Mapping)
Weak refinement mappings are defined by the predicate $\mathsf{is\text{-}weak\text{-}ref\text{-}map}$:

$$
\begin{array}{lcl}
\mathsf{is\text{-}weak\text{-}ref\text{-}map} & :: & (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\mathsf{ioa} \to (\alpha, \sigma_2)\,\mathsf{ioa} \to \mathsf{bool} \\
\mathsf{is\text{-}weak\text{-}ref\text{-}map}\, f\ C\ A & \equiv & (\forall s_0 \in \mathsf{starts\text{-}of}(C).\ f(s_0) \in \mathsf{starts\text{-}of}(A)) \wedge \\
 & & (\forall s\ t\ a. \mathsf{reachable}\ C\ s \wedge s \xrightarrow{a}_C t \\
 & & \qquad \Rightarrow\ \mathbf{if}\ a \in \mathsf{ext}(C)\ \mathbf{then}\, f(s) \xrightarrow{a}_A f(t) \\
 & & \qquad\qquad\qquad\qquad \mathbf{else}\ f(s) = f(t)\,)
\end{array}
$$

□

Note, however, that the use of weak refinement mappings is severely limited. For example, they do not generally allow to show that an automaton refines itself: The predicate $\mathsf{is\text{-}weak\text{-}ref\text{-}map}\,(\lambda x.\, x)\ C\ C$ only holds if C has no internal actions. This applies more generally: weak refinement mappings always assume that the abstract automaton A has no internal actions. This may be acceptable if A is the initial specification, but in a hierarchy of layered implementations it is inappropriate. See §12.3 for an example. Note that because of this strong limitations weak refinement mappings permit to work with sequences modeled as functions of type $\mathsf{nat} \to (\alpha)\,\mathsf{option}$ without normalizing, which has been done in [NS95]. This is explained in §7.7.

The relation between several of the refinement notions is easily established.

Lemma 8.4.4 (Relation between Refinement Notions)
Weak refinement mappings are refinement mappings. Refinement mappings induce forward simulations in the canonical way.

$$
\frac{\mathsf{ext}(C) = \mathsf{ext}(A) \quad \mathsf{is\text{-}weak\text{-}ref\text{-}map}\, f\ C\ A}{\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A}
\qquad
\frac{\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A}{\mathsf{is\text{-}simulation}\ \{(i, o).\ f(i) = o\}\ C\ A}
$$

Proof.

1. Case distinction on $a \in \mathsf{ext}(A)$. Use the lemmas

$$
\frac{s \xrightarrow{a}_A t}{\exists ex.\ \mathsf{is\text{-}move}\ A\ ex\ (s, a, t)}
\qquad
\frac{a \notin \mathsf{ext}(A) \quad s = t}{\exists ex.\ \mathsf{is\text{-}move}\ A\ ex\ (s, a, t)}
$$

for the two cases, respectively. The assumption $\mathsf{ext}(C) = \mathsf{ext}(A)$ is needed, as is-move is defined on A, whereas weak refinement mappings make a case distinction on $a \in \mathsf{ext}(C)$.

2. Using the lemma $(R[x] \cap S \neq \{\}) = (\exists y.\ (x, y) \in R \wedge y \in S)$ about sets it is easy to show the implication of the start state conditions. The implication of the step condition is fulfilled, because the existence of some t' with $(t, t') \in R$ can easily be shown by defining it to be $f(t)$. □

Definition 8.4.5 (Fair Simulations and Refinement Mappings)
Refinement mappings that transfer fairness from every execution to its corresponding one are called *fair refinement mappings*[2]:

$$
\begin{array}{lcl}
\mathsf{is\text{-}fair\text{-}ref\text{-}map} & :: & (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\mathsf{ioa} \to (\alpha, \sigma_2)\,\mathsf{ioa} \to \mathsf{bool} \\
\mathsf{is\text{-}fair\text{-}ref\text{-}map}\, f\ C\ A & \equiv & \mathsf{is\text{-}ref\text{-}map}\, f\ C\ A \wedge \\
 & & \forall exec \in \mathsf{executions}(C).\ \mathsf{is\text{-}fair}\ C\ exec \\
 & & \qquad\qquad \Rightarrow \mathsf{is\text{-}fair}\ A\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ exec)
\end{array}
$$

Forward (resp. backward) simulations with the analogous behavior are called *fair forward (resp. backward) simulations*:

$$
\begin{array}{lcl}
\mathsf{is\text{-}fair\text{-}simulation} & :: & (\sigma_1 \times \sigma_2)\,\mathsf{set} \to (\alpha, \sigma_1)\,\mathsf{ioa} \to (\alpha, \sigma_2)\,\mathsf{ioa} \to \mathsf{bool} \\
\mathsf{is\text{-}fair\text{-}simulation}\ R\ C\ A & \equiv & \mathsf{is\text{-}simulation}\ R\ C\ A \wedge \\
 & & \forall exec \in \mathsf{executions}(C).\ \mathsf{is\text{-}fair}\ C\ exec \\
 & & \qquad\qquad \Rightarrow \mathsf{is\text{-}fair}\ A\ (\mathsf{corresp}^{\mathsf{sim}}\ A\ R\ exec) \\
\mathsf{is\text{-}fair\text{-}back\text{-}simulation} & :: & (\sigma_1 \times \sigma_2)\,\mathsf{set} \to (\alpha, \sigma_1)\,\mathsf{ioa} \to (\alpha, \sigma_2)\,\mathsf{ioa} \to \mathsf{bool} \\
\mathsf{is\text{-}fair\text{-}back\text{-}simulation}\ R\ C\ A & \equiv & \mathsf{is\text{-}back\text{-}simulation}\ R\ C\ A \wedge \\
 & & \forall exec \in \mathsf{executions}(C).\ \mathsf{is\text{-}fair}\ C\ exec \\
 & & \qquad\qquad \Rightarrow \mathsf{is\text{-}fair}\ A\ (\mathsf{corresp}^{\mathsf{sim}}\ A\ R\ exec)
\end{array}
$$

□

8.5 Proof Infrastructure and Methodology

An important methodological point of this formalization is the separation of meta-theory and theory of I/O automata w.r.t. the employed logic: whereas the meta-theory makes heavy use of the advanced concepts of HOLCF, the user can stay within the simpler sublogic HOL. Subsequently, we present the technical prerequisites for this division.

[2] The notions $\mathsf{corresp}^{\mathsf{ref}}$ and $\mathsf{corresp}^{\mathsf{sim}}$ referred to below are defined in §9.

The meta-theoretic notion of safe implementation is trace inclusion which uses HOLCF to formalize the notion of traces. Simulations and refinement mappings, however, which represent the working vehicle for the user, can be formalized completely in plain HOL. For weak refinement mappings this is obvious, as the predicate $\mathsf{is\text{-}weak\text{-}ref\text{-}map}$ does not involve any variable of type $(\alpha)\,\mathsf{sequence}$. For refinement mappings and simulations this is not immediately clear, as both postulate the existence of a move, i.e. a state/sequence pair representing a finite execution fragment. Since the sequence has to be finite, however, its existence will be shown by explicitly specifying the steps that build the sequence. Such an approach is supported by schemes which are offered to the user. For example, for two steps the following lemmas are provided:

$$\frac{s_1 \xrightarrow{a}_A s_2 \quad s_2 \xrightarrow{i}_A s_3 \quad i \notin \mathsf{ext}(A)}{\exists ex.\ \mathsf{is\text{-}move}\ A\ ex\ (s_1, a, s_3)} \qquad \frac{s_1 \xrightarrow{i}_A s_2 \quad s_2 \xrightarrow{a}_A s_3 \quad i \notin \mathsf{ext}(A)}{\exists ex.\ \mathsf{is\text{-}move}\ A\ ex\ (s_1, a, s_3)}$$

Analogous schemes are provided for three (resp. four) steps, of which two (resp. three) are internal[3]. Theoretically, simulating moves may consist of arbitrarily many steps, although in a recent paper [GV98] it has been shown that a maximal length for every simulating move can be determined a priori without loosing completeness. In practice, the length should not exceed four steps, as otherwise the refinement proof itself becomes too complicated. Instead one should rather add a further implementation level in the refinement hierarchy.

This methodology allows to reuse the well-established proof infrastructure provided by HOL. In particular, proof strategies can be specialized to deal with the specific needs of our automaton model. We mention merely two examples. The tactic `ioa_invariant_tac` applies the invariant rule $(InvI)$, performs induction on the occurring actions and simplifies the remaining goals as far as possible. The tactic `ioa_condition_tac` discharges verification obligations which are typically generated by the application of meta-theoretic rules like compositionality. These obligations are mainly subproperties of the predicate $\mathsf{is\text{-}safe\text{-}IOA}$. The tactic proves them for every basic automaton and uses the following structural rules to extend the result to composed automata.

Lemma 8.5.1 (Structural Rules for Discharging I/O Automata Conditions)

$$\frac{}{\mathsf{is\text{-}trans\text{-}of}\ (A \parallel B)} \quad \frac{\mathsf{is\text{-}trans\text{-}of}(A)}{\mathsf{is\text{-}trans\text{-}of}\ (\mathsf{restrict}\ A\ as)} \quad \frac{\mathsf{is\text{-}trans\text{-}of}(A)}{\mathsf{is\text{-}trans\text{-}of}\ (\mathsf{rename}\ A\, f)}$$

$$\frac{\mathsf{is\text{-}sig\text{-}of}(A) \quad \mathsf{is\text{-}sig\text{-}of}(B) \quad \mathsf{compatible}\ A\ B}{\mathsf{is\text{-}sig\text{-}of}\ (A \parallel B)} \quad \frac{\mathsf{is\text{-}sig\text{-}of}(A)}{\mathsf{is\text{-}sig\text{-}of}\ (\mathsf{restrict}\ A\ as)} \quad \frac{\mathsf{is\text{-}sig\text{-}of}(A)}{\mathsf{is\text{-}sig\text{-}of}\ (\mathsf{rename}\ A\, f)}$$

$$\frac{\mathsf{compatible}\ A\ B \quad \mathsf{compatible}\ A\ C}{\mathsf{compatible}\ A\ (B \parallel C)} \quad \frac{\mathsf{compatible}\ A\ B \quad (\mathsf{ext}(B) \setminus as) \cap \mathsf{ext}(A) = \emptyset}{\mathsf{compatible}\ A\ (\mathsf{restrict}\ B\ as)}$$

$$\frac{\mathsf{compatible}\ A\ C \quad \mathsf{compatible}\ B\ C}{\mathsf{compatible}\ (A \parallel B)\ C} \quad \frac{\mathsf{input\text{-}enabled}(A) \quad \mathsf{input\text{-}enabled}(B) \quad \mathsf{compatible}\ A\ B}{\mathsf{input\text{-}enabled}\ (A \parallel B)}$$

□

[3] A tactic chooses the appropiate scheme for any given pair consisting of the number of steps and the pattern of internal/external actions.

8.6 Evaluation and Related Work

This chapter described the foundation for the verification environment we provide for I/O automata. In the sequel we summarize the benefits of this framework in general. First, we discuss the advantages which are due to the choice of I/O automata, Isabelle, and higher-order logic.

Confidence in Specification The expressiveness of higher-order logic and Isabelle's facilities for structuring specifications and proofs (theories, lemmas, hierarchic simplification sets, etc.) guarantee a neat correspondence between the formal description and the actual specification on the one hand (cf. the direct translation of I/O automata into Isabelle in §8.2), and the proof scripts and the informal, intuitive arguments on the other hand. This increases the confidence in really specifying and proving what one actually had in mind.

Confidence in Verification Computer-assisted proofs are usually more reliable than manual proofs, as they force the user to supply details for all cases. There are two reasons why this confidence in machine-checked proofs is even higher in our case than e.g. for proofs performed in PVS [ORR+96] or LP [GG91]. First, Isabelle itself is built according to the LCF system approach [Pau87], which means that every proof is broken down to a small and clear set of primitive inferences. Second, we introduce new theories only in a definitional way, which ensures that no inconsistencies, for example caused by contradictory axioms, can occur.

Scalability Interactive provers like Isabelle in general scale up better than fully automatic proof tools like model checkers. In addition, we believe that our I/O automata formalization is in particular qualified for full-scale applications, because of the following reason. Proofs of both simulations and invariants are essentially performed by case-splitting on actions and state components. If the number of actions and state components increases, the number of cases that have to be considered, increases as well, but each case appears to be no more complicated than in simple examples. This conjecture will be confirmed by the case studies in §12.

Readability Isabelle provides syntactical facilities like mixfix operators and powerful translations. Furthermore, most of the standard mathematical symbols like $\forall$, $\wedge$, $\in$ are supported[4]. Together with the expressiveness of higher-order logic this increases readability and thus considerably speeds up interactive proofs and the search for errors in specifications.

Rechecking During the course of carrying out a large proof, it is likely that definitions or lemmas have to be modified. Furthermore, one likes to polish already completed proofs. This can much safer and easier be done with computer-assistance.

[4] Actually, the syntax employed in this thesis represents only a slight modification, concerning subscripts and further special symbols.

There are further aspects which in particular profit from our methodology which combines HOL and HOLCF in such a way that both meta-theory and system verification are handled in the adequate logic, respectively. First, we discuss the rôle of HOL, i.e. the benefits for the user.

Automation Isabelle's automatic proof procedures permit to discharge the large amount of trivial, intermediate steps or simple cases which usually appear in refinement proofs. This applies in particular to the simplifier and the classical reasoner, which are tailored for the use in HOL (cf. §6.3). Furthermore, user-defined tactics like the aforementioned `ioa_invariant_tac` and `ioa_condition_tac` increase the degree of automation.

Reuse HOL provides large data type libraries, which can be reused and need not be redone for each application.

Simplicity The conceptually simple HOL is much easier to use than HOLCF, which incorporates the entire complexity of Scott's domain theory (cf. §5.2).

Second, we discuss the rôle of HOLCF, i.e. the advantages for establishing meta-theory.

Expressiveness HOLCF provides infinite datatypes and arbitrary recursion. This allows to define runs of automata and powerful recursive functions like infinite concatenation or sophisticated merge functions, which will turn out to be crucial for proving meta-theoretic results (cf. §9 and §10).

Extensibility Having the meta-theory at our disposal, we have a greater degree of flexibility because we do not need to hardwire certain proof methods but can derive new ones at any point. A remarkable example for such an extension is the temporal logic which will be build on top of the sequence model in §11.2.

There is, however, one drawback of our framework, at least w.r.t. the status of this chapter. While the safety part can be proved by reasoning about automaton steps, the fairness part has to take entire system runs into account. Thus, the user has to employ the more complicated logic HOLCF for this part. The reason is that we directly encoded the usual definition of fairness [Lyn96, RV96] into Isabelle. In §11, however, we will provide further infrastructure for fairness (and general liveness) using temporal logic. This will result in a framework, where the user treats liveness with standard rules of temporal logic or by the use of HOL only, whereas the HOLCF-specific parts are proved once and for all as derived rules of the meta-theory.

Related Work: There are several other works in the area of tool supported I/O automata verification.

The MIT distributed systems group, which originally developed I/O automata, has done substantial efforts in verifying simulations between I/O automata using the Larch Prover

(LP) [SAGG+93, PPG+96]. A number of case studies have been performed, involving timing based systems as well (e.g. [LSGL94, Gri95]). Current work [GLV97] aims at a formal language for I/O automata which allows to develop tools like static type checkers, simulators and code generators. The distinguishing feature to our work is is the fact that LP is a theorem prover for first-order logic. This means that the abstract notions of I/O automata incorporating composition operators and behaviours cannot be expressed within the logic. Instead, results on paper are used to extract a set of proof obligations from the formal description of the system. In particular, reasoning about meta-theory is impossible.

Archer and Heitmeyer [AH97a, AH97b] verified several benchmark problems modeled as Lynch/Vaandrager timed automata [LV96] in PVS [ORR+96]. Their goal is to build a customized prover on top of PVS, which is designed to process proof steps that resemble in style and size the typical steps in hand proofs. This is accomplished by tailored proof strategies, which resemble pretty much our specialized Isabelle tactics. However, their framework is restricted to invariant proofs, simulations are not taken into account until now. Furthermore, they do not consider meta-theory, although the logic of PVS would be powerful enough.

Further case studies have been performed with Coq [DFH+93] in the area of communication protocols [HSV94, BPV94]. Again, the works rely much more on unformalized meta-theory than we do.

Inspired by our work Griffioen and Devillers [GD98] formalized the meta-theory of I/O automata in PVS [ORR+96] and proved the correctness of refinement mappings. The framework has been used to verify a small part of the software of a multimedia bus protocol. For more details see §9.4.

Chapter 9

Soundness of Refinement Notions

In this chapter we establish the soundness of (fair) refinement mappings and (fair) forward simulations within Isabelle. Surprisingly, the proof scripts are relatively short compared with the corresponding paper proofs in the literature. This is due to the algebraic sequence model, which in particular allows to define and reason about infinite concatenation very easily.

9.1 Introduction

In this chapter we prove the soundness of refinement mappings and forward simulations (Thm. 2.4.6) and their fair counterparts in Isabelle.

The key idea of the proofs is the fact, that there is a certain correspondence between the *executions* of the involved automata and not only between their *traces*. If a forward simulation R exists between an implementation automaton C and a specification automaton A, then for every execution $exec_1$ in C there is a corresponding execution $exec_2$ in A that has the same trace as $exec_1$. More specifically, every step $s \xrightarrow{a}_C t$ in $exec_1$ corresponds to a subsequence of $exec_2$ that is a move of A. This means that the corresponding execution $exec_2$ represents the *infinite concatenation* of all the moves that correspond to the single steps of $exec_1$ under R (see Fig. 8.3).

Thus, it is essential that defining and reasoning about infinite concatenation is easily dealt with. As the definition of the Flatten operator in §6.3 shows, this is the case in our algebraic sequence model. In fact it will turn out that the proofs are very concise and, in addition, easy to automate.

9.2 Soundness of Refinement Mappings

We start with the definition of the desired corresponding execution (see Fig. 9.1).

Definition 9.2.1 (Corresponding Execution)
Given an execution (s, ex) of C and a function f between the states of C and A the function $\mathsf{corresp}^{\mathsf{ref}}$ builds a state/sequence pair of A that corresponds to (s, ex) under f.

$$\begin{array}{lcl} \mathsf{corresp}^{\mathsf{ref}} & :: & (\alpha, \sigma_2)\,\mathsf{ioa} \to (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\mathsf{execution} \to (\alpha, \sigma_2)\,\mathsf{execution} \\ \mathsf{corresp}^{\mathsf{ref}}\ A\,f\ (s, ex) & \equiv & (f\ s, \mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}\ A\,f\ `ex\ s) \end{array}$$

The start state s is mapped to the corresponding start state $f(s)$. Furthermore, the infinite concatenation of the moves of A is realized by the following operation:

$$\mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}\ :: \ (\alpha, \sigma_2)\,\mathsf{ioa} \to (\sigma_1 \to \sigma_2) \to (\alpha \times \sigma_1)\,\mathsf{sequence} \to_c \sigma_1 \to (\alpha \times \sigma_2)\,\mathsf{sequence}$$

The following equations follow immediately from the definition:

$$\begin{array}{lcl} \mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}\ A\,f\ `\bot\ s & = & \bot \\ \mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}\ A\,f\ `\mathsf{nil}\ s & = & \mathsf{nil} \\ \mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}\ A\,f\ `((a,t)\hat{\ }ex_1)\ s & = & (\varepsilon\ ex_2.\ \mathsf{is\text{-}move}\ A\ ex_2\ (f\ s, a, f\ t)) \\ & & \oplus\ \mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}\ A\,f\ `ex_1\ t \end{array}$$

□

Corollary 9.2.2
The following equations for $\mathsf{corresp}^{\mathsf{ref}}$ follow immediately from those for $\mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}$:

$$\begin{array}{lcl} \mathsf{corresp}^{\mathsf{ref}}\ A\,f\ (s, \bot) & = & (f\ s, \bot) \\ \mathsf{corresp}^{\mathsf{ref}}\ A\,f\ (s, \mathsf{nil}) & = & (f\ s, \mathsf{nil}) \\ \mathsf{corresp}^{\mathsf{ref}}\ A\,f\ (s, (a,t)\hat{\ }ex_1) & = & (f\ s, (\varepsilon\ ex_2.\ \mathsf{is\text{-}move}\ A\ ex_2\ (f\ s, a, f\ t)) \\ & & \qquad \oplus\ \mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\,f\ (t, ex_1)) \end{array}$$

□

Intuitively, $\mathsf{corresp}^{\mathsf{ref}}$ works as follows: every step in the execution ex of C is mapped to the corresponding move of A, then these infinitely many moves are concatenated yielding the corresponding execution. Technically however, $\mathsf{corresp}^{\mathsf{ref}}$ cannot be realized by Map and $\mathsf{Flatten}$, as ex contains only pairs (a, t), whereas entire steps (s, a, t) are needed to construct the corresponding move. Therefore, the state s has to be carried over from the last pair using a fourth argument for $\mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}$.

Furthermore, note that $\mathsf{corresp}^{\mathsf{ref}}$ does not take executions as argument, but merely their second projections, i.e. their sequence parts. This avoids to eliminate t when concatenating (s, ex_1) and (t, ex_2). In addition, it matches the definition of $\mathsf{is\text{-}move}$ very well, which also takes only the sequence part of an execution as argument. Nevertheless, the first states s and t are not neglected in $\mathsf{is\text{-}move}$ and $\mathsf{corresp}_{\mathsf{c}}^{\mathsf{ref}}$, but are taken care of implicitly. This is necessary, as otherwise it would not be true that $(s, ex_1 \oplus ex_2)$ is an execution. To

$$\begin{array}{l|ccccc}
\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ exec & (f(s_0), & [\,(e_0, s_0'), (i_2, f(s_1))\,!] & \oplus\,\mathsf{nil} & \oplus\,[\,(e_1, f(s_3))\,!]) \\
 & \uparrow & \uparrow & \nwarrow & \uparrow \\
exec & (s_0, & [\,(e_0, s_1), & (i_1, s_2), & (e_1, s_3)\,!])
\end{array}$$

Figure 9.1: A corresponding execution: e_i denote external, i_j internal actions, $f(s_1) = f(s_2)$.

prove this, it is required that t is the last state of ex_1. See the example in Fig. 9.1, which illustrates these facts.

Note that the use of the description operator ε does not complicate reasoning in this context, as the desired ex_2 always exists: the definition of is-ref-map exactly states the existence of a simulation move for every reachable state of C. Thus, the description operator ε can always be eliminated: the only property we need about $\varepsilon\ ex_2$, namely that it is a move of A, is always guaranteed. This is expressed in the following lemma.

Lemma 9.2.3 (Existence of Move)
Given a refinement mapping f there is a corresponding move of A for every reachable state s and step $s \stackrel{a}{\rightarrow}_C t$.

$$\frac{\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A \quad \mathsf{reachable}\ C\ s \quad s \stackrel{a}{\rightarrow}_C t}{\mathsf{is\text{-}move}\ A\ (\varepsilon\ ex.\ \mathsf{is\text{-}move}\ A\ ex\ (f\ s, a, f\ t))\ (f\ s, a, f\ t)}$$

Proof.
Simple consequence of the definition of is-ref-map and $(\exists x.\ P\ x) \Rightarrow P\ (\varepsilon\ y.\ P\ y)$. □

Corollary 9.2.4 (Move Properties)
The following theorems are simple implications of the preceding lemma:

$$\frac{\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A \quad \mathsf{reachable}\ C\ s \quad s \stackrel{a}{\rightarrow}_C t}{\mathsf{is\text{-}exec\text{-}frag}\ A\ (f\ s, \varepsilon\ ex.\ \mathsf{is\text{-}move}\ A\ ex\ (f\ s, a, f\ t))} \qquad (1)$$

$$\frac{\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A \quad \mathsf{reachable}\ C\ s \quad s \stackrel{a}{\rightarrow}_C t}{\mathsf{Finite}\ (\varepsilon\ ex.\ \mathsf{is\text{-}move}\ A\ ex\ (f\ s, a, f\ t))} \qquad (2)$$

$$\frac{\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A \quad \mathsf{reachable}\ C\ s \quad s \stackrel{a}{\rightarrow}_C t}{\mathsf{last\text{-}state}\ (f\ s, \varepsilon\ ex.\ \mathsf{is\text{-}move}\ A\ ex\ (f\ s, a, f\ t)) = (f\ t)} \qquad (3)$$

$$\frac{\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A \quad \mathsf{reachable}\ C\ s \quad s \stackrel{a}{\rightarrow}_C t}{\begin{array}{l}\mathsf{mk\text{-}trace}\ A\ `(\varepsilon\ ex.\ \mathsf{is\text{-}move}\ A\ ex\ (f\ s, a, f\ t)) = \\ (\mathbf{if}\ a \in \mathsf{ext}\ A\ \mathbf{then}\ [\,a\,!]\ \mathbf{else}\ \mathsf{nil})\end{array}} \qquad (4)$$

□

The main correctness result can be divided into two steps. Given a refinement mapping f from C to A, we have to show

- **Trace Equality**: the traces of $exec$ and ($\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ exec$) coincide and
- **Execution Property**: ($\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ exec$) is an execution of A.

The two properties will be shown consecutively.

9.2.1 Trace Equality.

The $\mathsf{is\text{-}move}$ property guarantees that every move of A and its simulated step of C have the same trace. These local trace equalities have to be extended to the concatenation of moves, which is essentially possible because of the following trivial lemma:

Lemma 9.2.5
Generating traces distributes over concatenation.

$$\mathsf{mk\text{-}trace}\ A\ `(ex_1 \oplus ex_2) = (\mathsf{mk\text{-}trace}\ A\ `ex_1) \oplus (\mathsf{mk\text{-}trace}\ A\ `ex_2)$$

Proof.
Follows immediately from the fact, that both Filter and Map distribute over $\oplus$. □

This lemma is now used to show trace equivalence not only for the binary concatenation of moves, but for their infinite concatenation produced by $\mathsf{corresp}^{\mathsf{ref}}$.

Lemma 9.2.6 (Trace Equality)
The traces of the execution (s, ex) and $\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (s, ex)$ coincide if f is a refinement mapping from C to A and the external actions are the same.

$$\frac{\mathcal{A}_1\text{: } \mathsf{is\text{-}ref\text{-}map}\, f\ C\ A \quad \mathcal{A}_2\text{: } \mathsf{ext}(C) = \mathsf{ext}(A)}{\begin{array}{l}\mathcal{C}_1\text{: } \forall s. \mathsf{reachable}\ C\ s \wedge \mathsf{is\text{-}exec\text{-}frag}\ C\ (s, ex) \\ \qquad \Rightarrow \mathsf{mk\text{-}trace}\ A\ `(\mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (s, ex))) = \mathsf{mk\text{-}trace}\ C\ `ex\end{array}}$$

Proof.
The proof is by **structural induction** on ex. The **admissibility** condition is discharged automatically and the **base cases** $ex = \bot$, $ex = \mathsf{nil}$ are trivial.

Inductive step $ex = (a, t)\hat{}\, ex_1$**:** Assume $\mathcal{C}_1$ as induction hypothesis. Thus, we get a new goal which has to be shown under three further assumptions.

$$\begin{array}{ll}\mathcal{A}_3\text{:} & \mathsf{reachable}\ C\ s \qquad \mathcal{A}_4\text{: } \ s \stackrel{a}{\rightarrow}_C t \qquad \mathcal{A}_5\text{: } \ \mathsf{is\text{-}exec\text{-}frag}\ C\ (t, ex_1) \\ \mathcal{C}_2\text{:} & \mathsf{mk\text{-}trace}\ A\ `(\mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (s, (a, t)\hat{}\, ex_1))) = \mathsf{mk\text{-}trace}\ C\ `((a, t)\hat{}\, ex_1)\end{array}$$

Case $a \in \mathsf{ext}(A)$**:**

$$
\begin{array}{ll}
& \mathsf{mk\text{-}trace}\ A\ `(\mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (s, (a,t)\hat{\ }ex_1))) \\
= & \quad \{\mathit{Cor.\ 9.2.2:\ (Equations\ for}\ \mathsf{corresp}^{\mathsf{ref}})\} \\
& \mathsf{mk\text{-}trace}\ A\ `(\varepsilon\ ex_2.\ \mathsf{is\text{-}move}\ A\ ex_2\ (f\ s, a, f\ t)\ \oplus\ (\mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (s, ex_1)))) \\
= & \quad \{\mathit{Lemma\ 9.2.5:\ (Distributivity\ of}\ \mathsf{mk\text{-}trace}\ \mathit{over}\ \oplus)\} \\
& \mathsf{mk\text{-}trace}\ A\ `(\varepsilon\ ex_2.\ \mathsf{is\text{-}move}\ A\ ex_2\ (f\ s, a, f\ t)) \\
& \oplus\ \mathsf{mk\text{-}trace}\ A\ `(\mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (s, ex_1))) \\
= & \quad \{\mathit{IH\ with}\ s := t,\ ex := ex_1,\ \mathcal{A}_1 - \mathcal{A}_3,\ \mathit{and\ Def.\ 8.2.7}\ (\mathsf{reachable}).\} \\
& \mathsf{mk\text{-}trace}\ A\ `(\varepsilon\ ex_2.\ \mathsf{is\text{-}move}\ A\ ex_2\ (f\ s, a, f\ t))\ \oplus\ \mathsf{mk\text{-}trace}\ C\ `ex_1 \\
= & \quad \{\mathit{Lemma\ 9.2.4\ (Move\ property}\ (4)),\ \mathcal{A}_1,\ \mathcal{A}_3,\ a \in \mathsf{ext}(A)\} \\
& [a!]\ \oplus\ \mathsf{mk\text{-}trace}\ C\ `ex_1 \\
= & \quad \{\mathit{Def.\ 6.2.5}\ (\oplus),\ \mathit{Cor.\ 8.3.5:\ (Equations\ for}\ \mathsf{mk\text{-}trace}),\ \mathcal{A}_2\} \\
& \mathsf{mk\text{-}trace}\ C\ `((a,t)\hat{\ }ex_1)
\end{array}
$$

Case $a \notin \mathsf{ext}(A)$**:** Analogous. □

The assumption $\mathsf{ext}(A) = \mathsf{ext}(C)$ is needed, as $\mathsf{mk\text{-}trace}$ generates traces w.r.t. the signature of A on the *lhs* and traces w.r.t. the signature of C on the *rhs*. The other assumptions ($\mathsf{is\text{-}ref\text{-}map}\, f\ C\ A$) and ($\mathsf{reachable}\ C\ s$) are needed to ensure the existence of a move, i.e. to apply Lemma 9.2.4.

9.2.2 Execution Property.

Just as before, the $\mathsf{is\text{-}move}$ property yields already the property of being an execution-fragment for every simulation move. To prove the property for the entire corresponding execution, we need lemmas that propagate it from single executions to their finite and infinite concatenation. The next lemma treats binary concatenation first.

Lemma 9.2.7
The predicate $\mathsf{is\text{-}exec\text{-}frag}$ propagates from single executions (s, ex_1) and (u, ex_2) to their concatenation $(s, ex_1\ \oplus\ ex_2)$ provided that u is the last state of ex_1.

$$
\begin{array}{ll}
\mathcal{C}_1: & \mathsf{Finite}(ex_1) \\
& \Rightarrow\ \forall s. \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, ex_1) \wedge \mathsf{is\text{-}exec\text{-}frag}\ A\ (u, ex_2) \wedge \\
& \qquad\quad \mathsf{last\text{-}state}\ (s, ex_1) = u \\
& \qquad\quad \Rightarrow \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, ex_1\ \oplus\ ex_2)
\end{array}
$$

Proof.
The proof is by **finite structural induction** on ex_1: The **base case** $ex_1 = \mathsf{nil}$ is trivial.

Inductive step $ex_1 = (a,t)\hat{\ } ex_1'$**:** Assume $\mathcal{C}_1$ as induction hypothesis. Thus, we get a new goal which has to be shown under three further assumptions.

$$\begin{array}{ll}
\mathcal{A}_1: & s \xrightarrow{a}_A t \wedge \textsf{is-exec-frag}\ A\ (t, ex_1') \\
\mathcal{A}_2: & \textsf{is-exec-frag}\ A\ (u, ex_2) \\
\mathcal{A}_3: & \textsf{last-state}\ (s, (a,t)\hat{\ } ex_1') = \textsf{last-state}\ (t, ex_1') = u \\
\mathcal{C}_2: & \textsf{is-exec-frag}\ A\ (s, ((a,t)\hat{\ } ex_1') \oplus ex_2)
\end{array}$$

The following equalities yield the result.

$$\begin{array}{ll}
 & \textsf{is-exec-frag}\ A\ (s, ((a,t)\hat{\ } ex_1') \oplus ex_2) \\
= & \quad \{\textit{Def. 8.3.3, Def. 6.2.5, (Equations for } \textsf{is-exec-frag} \textit{ and } \oplus\textit{)}\} \\
 & s \xrightarrow{a}_A t \wedge \textsf{is-exec-frag}\ A\ (t, ex_1' \oplus ex_2) \\
= & \quad \{\mathcal{A}_1\} \\
 & \textsf{is-exec-frag}\ A\ (t, ex_1' \oplus ex_2) \\
= & \quad \{\textit{Induction hypothesis with } s := t,\ ex_1 := ex_1',\ \mathcal{A}_1 - \mathcal{A}_3\} \\
 & \textsf{True}
\end{array}$$

□

Notice that the assumption $\textsf{Finite}(ex_1)$ is not necessary, as the goal $\textsf{is-exec-frag}\ A\ (s, ex_1 \oplus ex_2)$ reduces to $\textsf{is-exec-frag}\ A\ (s, ex_1)$ if ex_1 is partial or infinite. In our context, however, we need the lemma only under this assumption, as we argue about moves, and the $\textsf{is-move}$ property includes the finiteness requirement. Furthermore, $\textsf{last-state}\ (s, ex)$ is not continuous in ex, therefore the lemma cannot be proven to be admissible using the syntactic admissibility test. Thus, infinite structural induction cannot be applied immediately. Finite structural induction, however, can be applied very easily, as $\textsf{last-state}$ has been characterized inductively for finite sequences (Lemma 8.3.8).

Now we propagate the execution property from single sequences to their *infinite* concatenation.

Lemma 9.2.8 (Execution Property)
Given a refinement mapping f from C to A, $\textsf{corresp}^{\textsf{ref}}\ A\, f\ (s, ex)$ is an execution fragment of A, provided (s, ex) is an execution fragment of C.

$$\frac{\mathcal{A}_1:\ \textsf{is-ref-map}\ f\ C\ A}{\begin{array}{l}\mathcal{C}_1:\ \forall s. \textsf{reachable}\ C\ s \wedge \textsf{is-exec-frag}\ C\ (s, ex) \\ \qquad\qquad \Rightarrow \textsf{is-exec-frag}\ A\ (\textsf{corresp}^{\textsf{ref}}\ A\, f\ (s, ex))\end{array}}$$

Proof.
The proof is by **structural induction** on ex. The **admissibility** is discharged automatically and the **base cases** $ex = \bot$, $ex = \mathsf{nil}$ are trivial.

Inductive step $ex = (a,t)\hat{\ } ex_1$**:** Assume $\mathcal{C}_1$ as induction hypothesis. Thus, we get a new goal which has to be shown under three further assumptions.

$$\mathcal{A}_2\colon\ \mathsf{reachable}\ C\ s \qquad \mathcal{A}_3\colon\ s \xrightarrow{a}_C t \qquad \mathcal{A}_4\colon\ \mathsf{is\text{-}exec\text{-}frag}\ C\ (t, ex_1)$$
$$\mathcal{C}_2\colon\ \mathsf{is\text{-}exec\text{-}frag}\ A\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (s, (a,t)\hat{\ } ex_1))$$

The following equalities yield the result.

$$
\begin{array}{cl}
 & \mathsf{is\text{-}exec\text{-}frag}\ A\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (s, (a,t)\hat{\ } ex_1)) \\
= & \{\textit{Cor. 9.2.2 (Equations for } \mathsf{corresp}^{\mathsf{ref}}\textit{)}\} \\
 & \mathsf{is\text{-}exec\text{-}frag}\ A\ (f\ s, \varepsilon\ ex_2.\ \mathsf{is\text{-}move}\ A\ ex_2\ (f\ s, a, f\ t)\ \oplus\ \mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (t, ex_1))) \\
= & \{\textit{Lemma 9.2.7 (}\mathsf{is\text{-}exec\text{-}frag}\textit{ propagates to } \oplus\textit{), } t := f(t)\ \} \\
 & \mathsf{is\text{-}exec\text{-}frag}\ A\ (f\ s, \varepsilon\ ex_2.\ \mathsf{is\text{-}move}\ A\ ex_2\ (f\ s, a, f\ t)) \wedge \\
 & \mathsf{is\text{-}exec\text{-}frag}\ A\ (f\ t, \mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (t, ex_1))) \wedge \\
 & \mathsf{last\text{-}state}\ (f\ s, \varepsilon\ ex_2.\ \mathsf{is\text{-}move}\ A\ ex_2\ (f\ s, a, f\ t)) = (f\ t) \\
= & \{\textit{2}\times\textit{ Lemma 9.2.4 (Move properties (1) and (3)), } \mathcal{A}_1 - \mathcal{A}_3\} \\
 & \mathsf{is\text{-}exec\text{-}frag}\ A\ (f\ t, \mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (t, ex_1))) \\
= & \{\textit{Unfold, simplify and fold Def. 9.2.1: (}\mathsf{corresp}^{\mathsf{ref}}\textit{)}\} \\
 & \mathsf{is\text{-}exec\text{-}frag}\ A\ (\mathsf{corresp}^{\mathsf{ref}}\ A\, f\ (t, ex_1)) \\
= & \{\textit{IH with } s := t,\ ex := ex_1,\ \mathcal{A}_2 - \mathcal{A}_4,\ \textit{Def. 8.2.7 (reachable-n)}\} \\
 & \mathsf{True}
\end{array}
$$

□

9.2.3 Main Soundness Results

Using the two important properties about $\mathsf{corresp}^{\mathsf{ref}}$ that have been proved in the previous subsections, it is now straight-forward to derive the main soundness result for refinement mappings.

Theorem 9.2.9 (Soundness of Refinement Mappings)
A implements C safely if there is a refinement mapping from C to A and the external actions are the same.

$$\frac{\mathcal{A}_1\colon\ \mathsf{is\text{-}ref\text{-}map}\, f\ C\ A \quad \mathcal{A}_2\colon\ \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A)}{\mathcal{C}_1\colon\ C \preceq_S A}$$

Proof.
Because of $\mathcal{A}_2$ and by the definition of $\preceq_C$, $\mathcal{C}_1$ reduces to $\mathcal{C}_2$: $\mathsf{traces}(C) \subseteq \mathsf{traces}(A)$ which reduces further by elementary set equalities, some propositional logic, and the characterization of traces (Lemma 8.3.6) to

$$\begin{aligned} \mathcal{C}_3\colon\quad & exec_1 \in \mathsf{executions}(C) \\ & \Rightarrow \exists exec_2 \in \mathsf{executions}(A).\ \mathsf{mk\text{-}trace}\ C\ \text{‘}(\mathsf{snd}\ exec_1) = \mathsf{mk\text{-}trace}\ A\ \text{‘}(\mathsf{snd}\ exec_2) \end{aligned}$$

Thus, the existence of an execution $exec_2 \in \mathsf{executions}(A)$ has to be shown, that has the same trace as $exec_1$. Such an execution is exactly the corresponding execution of $exec_1$ given by $exec_2 := \mathsf{corresp}^{\mathsf{ref}}\ A f\ exec_1$. Therefore, by writing (s_1, ex_1) for $exec_1$, using the equations for $\mathsf{is\text{-}exec\text{-}frag}$, and unfolding the definition of $\mathsf{executions}$, $\mathcal{C}_3$ reduces to the propositions

$$\begin{aligned} \mathcal{C}_4\colon\quad & \mathsf{corresp}^{\mathsf{ref}}\ A\ f\ (s_1, ex_1) \in \mathsf{executions}\ A \\ \mathcal{C}_5\colon\quad & \mathsf{mk\text{-}trace}\ C\ \text{‘}ex_1 = \mathsf{mk\text{-}trace}\ A\ \text{‘}(\mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\ f\ (s_1, ex_1))) \end{aligned}$$

which have to be shown under the assumption

$$\mathcal{A}_3\colon\quad s_1 \in \mathsf{starts\text{-}of}(C) \wedge \mathsf{is\text{-}exec\text{-}frag}\ C\ (s_1, ex_1)$$

For $\mathcal{C}_5$, Lemma 9.2.6 (Trace equality) can immediately be applied using $\mathcal{A}_1$–$\mathcal{A}_3$ and the inductive rule *reachable*-0 (Def. 8.2.7). For $\mathcal{C}_4$, it has to be shown that $f(s_1) \in \mathsf{starts\text{-}of}(A)$, which is an implication of $\mathcal{A}_1$ together with $\mathcal{A}_3$. Then Lemma 9.2.8 (Execution Property) can be applied using $\mathcal{A}_1$ and $\mathcal{A}_3$. □

For fair refinement mappings an analogous soundness result is obtainable, which uses the same two important lemmas concerning trace equality and execution correspondence.

Theorem 9.2.10 (Soundness of Fair Refinement Mappings)
A implements *C* fairly if there is a fair refinement mapping from *C* to *A* and the external actions are the same.

$$\frac{\mathcal{A}_1\colon\ \mathsf{is\text{-}fair\text{-}ref\text{-}map}\,f\ C\ A \qquad \mathcal{A}_2\colon\ \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A)}{\mathcal{C}_1\colon\ C \preceq_F A}$$

Proof.
Because of $\mathcal{A}_2$ and by the definition of $\preceq_F$, $\mathcal{C}_1$ reduces to $\mathcal{C}_2$: $\mathsf{fair\text{-}traces}(C) \subseteq \mathsf{fair\text{-}traces}(A)$ which reduces further by elementary set equalities, some propositional logic, and the definition of $\mathsf{fair\text{-}traces}$ to

$$\begin{aligned} \mathcal{C}_3\colon\quad & exec_1 \in \mathsf{executions}(C) \wedge \mathsf{is\text{-}fair}\ C\ apexec_1 \\ & \Rightarrow \exists exec_2 \in \mathsf{executions}(A).\ \mathsf{mk\text{-}trace}\ C\ \text{‘}(\mathsf{snd}\ exec_1) = \mathsf{mk\text{-}trace}\ A\ \text{‘}(\mathsf{snd}\ exec_2) \wedge \\ & \qquad\qquad \mathsf{is\text{-}fair}\ A\ exec_2 \end{aligned}$$

For $exec_2$ we choose as before the corresponding execution of $exec_1$ given by $exec_2 :=$ $\mathsf{corresp}^{\mathsf{ref}}\ A\ f\ exec_1$. Therefore, by writing (s_1, ex_1) for $exec_1$, using the equations for is-exec-frag, and unfolding the definition of executions, $\mathcal{C}_3$ reduces to the propositions

$$\mathcal{C}_4:\ \ \mathsf{corresp}^{\mathsf{ref}}\ A\,f\ (s_1, ex_1) \in \mathsf{executions}\ A$$
$$\mathcal{C}_5:\ \ \mathsf{mk\text{-}trace}\ C\ `ex_1 = \mathsf{mk\text{-}trace}\ A\ `(\mathsf{snd}\ (\mathsf{corresp}^{\mathsf{ref}}\ A\,f\ (s_1, ex_1)))$$
$$\mathcal{C}_6:\ \ \mathsf{is\text{-}fair}\ A\ (\mathsf{corresp}^{\mathsf{ref}}\ A\,f\ (s_1, ex_1))$$

which have to be shown under the assumption

$$\mathcal{A}_3:\ \ (s_1, ex_1) \in \mathsf{executions}(C) \wedge \mathsf{is\text{-}fair}\ C\ exec_1$$

$\mathcal{C}_4$ and $\mathcal{C}_5$ are discharged in the same manner as in the soundness proof for the safe case. $\mathcal{C}_6$ follows from $\mathcal{A}_1$ and $\mathcal{A}_3$ with the definition of a fair refinement mapping. □

9.3 Soundness of Forward Simulations

The correctness of forward simulations is a stronger result than the correctness of refinement mappings, as every refinement mapping induces a forward simulation. Therefore, it would be sufficient to show only the correctness of forward simulations. However, we decided to perform both proofs, as although they follow along the same lines, the simulation correctness proof is distinctly more subtle in detail. Thus, for sake of simplicity we will describe the simulation proof in terms of what changes in the proof of the previous section.

Forward simulations differ from refinement mappings in using relations R instead of functions f between the states of C and A. This mainly implies that the construction of the corresponding execution has to consider more underspecification, which is accomplished by further uses of the choice operator. In detail, the following changes to $\mathsf{corresp}^{\mathsf{ref}}$ have to be made to construct a corresponding execution $\mathsf{corresp}^{\mathsf{sim}}$ for simulations (cf. Fig. 9.2 and the definition below):

- For every start state s of C the corresponding state s' of A has to be chosen in such a way that $(s, s') \in R$ and $s' \in \mathsf{starts\text{-}of}(A)$.

- Consider a step $u \xrightarrow{b}_C v$, its succeeding step $s \xrightarrow{a}_C t$, and their corresponding moves of A. First, note that from $v = s$ it does not follow in general that their corresponding states are equal, as $f(v) = f(s)$ is true for a function f, but not for all values $v' \in R[v]$ and $s' \in R[s]$ for an arbitrary relation R. Thus, for $\mathsf{corresp}^{\mathsf{sim}}$ it does not suffice to hand over the concrete state of the last step as further argument, but its corresponding state is needed as well. Actually, the concrete state can be neglected then.
  Second, the state t' that corresponds to t has to be chosen in such a way that $(t, t') \in R$ and that there is a move from s' to that t'.

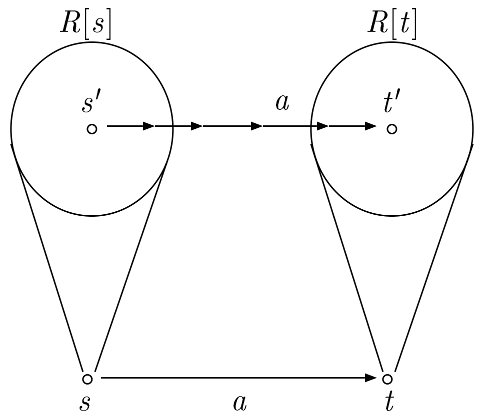

Figure 9.2: A Step of a Forward Simulation

Therefore two more applications of the choice operator are needed, where one is nested with the choice operator already used for $\mathsf{corresp}^{\mathsf{ref}}$:

Definition 9.3.1 (Corresponding Execution for Forward Simulations)
Given an execution (s, ex) of C and a relation R between the states of C and A the function $\mathsf{corresp}^{\mathsf{sim}}$ constructs a state/sequence pair of A that corresponds to (s, ex) under R.

$$\begin{array}{lcl} \mathsf{corresp}^{\mathsf{sim}} & :: & (\alpha, \sigma_2)\,\mathsf{ioa} \to (\sigma_1 \times \sigma_2)\,\mathsf{set} \to (\alpha, \sigma_1)\,\mathsf{execution} \to (\alpha, \sigma_2)\,\mathsf{execution} \\ \mathsf{corresp}^{\mathsf{sim}}\ A\ R\ (s, ex) & \equiv & \textbf{let}\ \ s' = \varepsilon\, s'.\, (s, s') \in R \wedge s' \in\ \mathsf{starts\text{-}of}(A) \\ & & \textbf{in}\ \ \ (s', \mathsf{corresp}^{\mathsf{sim}}_{\mathsf{c}}\ A\ R\ `ex\ s') \end{array}$$

For the start state s a corresponding start state s' is chosen. Furthermore, the infinite concatenation of the moves of A is realized by the following operation:

$$\mathsf{corresp}^{\mathsf{sim}}_{\mathsf{c}} :: (\alpha, \sigma_2)\,\mathsf{ioa} \to (\sigma_1 \times \sigma_2)\,\mathsf{set} \to (\alpha \times \sigma_1)\,\mathsf{sequence} \to_c \sigma_2 \to (\alpha \times \sigma_2)\,\mathsf{sequence}$$

The following equations follow immediately from the definition:

$$\begin{array}{lcl} \mathsf{corresp}^{\mathsf{sim}}_{\mathsf{c}}\ A\ R\ `\bot\ s & = & \bot \\ \mathsf{corresp}^{\mathsf{sim}}_{\mathsf{c}}\ A\ R\ `\mathsf{nil}\ s & = & \mathsf{nil} \\ \mathsf{corresp}^{\mathsf{sim}}_{\mathsf{c}}\ A\ R\ `((a,t)\hat{}\, ex_1)\ s' & = & \textbf{let}\ \ (ex_2, t') = \varepsilon\,(ex_2, t').(t, t') \in R \wedge \\ & & \qquad\qquad\qquad\qquad \mathsf{is\text{-}move}\ A\ ex_2\ (s', a, t') \\ & & \textbf{in}\ \ \ ex_2 \oplus \mathsf{corresp}^{\mathsf{sim}}_{\mathsf{c}}\ A\ R\ `ex_1\ t' \end{array}$$

□

Note that rewrite rules like those for $\mathsf{corresp}^{\mathsf{ref}}$ are not derivable for $\mathsf{corresp}^{\mathsf{sim}}$, as the requirement on the start state $\varepsilon\, s'.\, (s, s') \in R \wedge s' \in \mathsf{starts\text{-}of}(A)$ does not allow a recursive characterization. We work with $\mathsf{corresp}^{\mathsf{sim}}_{\mathsf{c}}$ instead, which enables us to talk about this requirement merely for the main correctness result.

As for refinement mappings, the choice operators do not complicate reasoning because of the following lemma.

Lemma 9.3.2 (Existence of Move)
Given a forward simulation R there is a corresponding move of A for every reachable state s and step $s \xrightarrow{a}_C t$.

$$\frac{\textsf{is-simulation}\ R\ C\ A \quad \textsf{reachable}\ C\ s \quad s \xrightarrow{a}_C t \quad (s,s') \in R}{\begin{array}{ll}\textbf{let} & (ex_2, t') = \varepsilon\,(ex_2, t').\ (t,t') \in R \wedge \textsf{is-move}\ A\ ex_2\ (s', a, t') \\ \textbf{in} & (t,t') \in R \wedge \textsf{is-move}\ A\ ex_2\ (s', a, t')\end{array}}$$

Proof.
Follows essentially by applying $(\exists x.\ P\ x) \Rightarrow P\ (\varepsilon\, y.\ P\ y)$. □

Similar to the previous section, lemmas can directly be derived which describe this result for every subproperty of is-move. They will not be displayed here, but implicitly assumed when referring to the result above.

The facts that trace generation and the execution property propagate from executions to their *binary* concatenation can be reused. We merely have to redo the proof for the *infinite* concatenation, i.e. for $\textsf{corresp}^{\textsf{sim}}$ instead of $\textsf{corresp}^{\textsf{ref}}$. In turn, only the changes to the corresponding proofs for $\textsf{corresp}^{\textsf{ref}}$ are explained.

Lemma 9.3.3 (Trace Equality)
Given a forward simulation R between C and A, the traces of the execution (s, ex) and $\textsf{corresp}^{\textsf{sim}}\ A\ R\ (s, ex)$ coincide.

$$\frac{\textsf{is-simulation}\ R\ C\ A \quad \textsf{ext}(C) = \textsf{ext}(A)}{\begin{array}{l}\forall s\ s'.\textsf{reachable}\ C\ s \wedge \textsf{is-exec-frag}\ C\ (s, ex) \wedge (s,s') \in R \\ \qquad \Rightarrow \textsf{mk-trace}\ A\ `(\textsf{corresp}^{\textsf{sim}}_{\textsf{c}}\ A\ R\ `ex\ s') = \textsf{mk-trace}\ C\ `ex\end{array}}$$

Proof.
Similar to the proof for $\textsf{corresp}^{\textsf{ref}}$. Merely s' has to be instantiated by

$$\textsf{snd}\ (\varepsilon\,(ex_2, t').\ (t,t') \in R \wedge \textsf{is-move}\ A\ ex_2\ (s', a, t'))$$

for applying the induction hypothesis. Such a value t' exists because of Lemma 9.3.2. □

Lemma 9.3.4 (Execution Property)
Given a forward simulation R between C and A, $\textsf{corresp}^{\textsf{sim}}\ A\ R\ (s, ex)$ is an execution fragment of A, provieded that (s, ex) is an execution fragment of C.

$$\frac{\textsf{is-simulation}\ R\ C\ A}{\begin{array}{l}\forall s\ s'.\textsf{reachable}\ C\ s \wedge \textsf{is-exec-frag}\ C\ (s, ex) \wedge (s,s') \in R \\ \qquad \Rightarrow \textsf{is-exec-frag}\ A\ (s', \textsf{corresp}^{\textsf{sim}}_{\textsf{c}}\ A\ R\ `ex\ s')\end{array}}$$

Proof.
Similar to the proof for $\mathsf{corresp}^{\mathsf{ref}}$. Merely s' has to be instantiated by

$$\mathsf{snd}\ (\varepsilon\ (ex_2, t').\ (t, t') \in R \wedge \mathsf{is\text{-}move}\ A\ ex_2\ (s', a, t'))$$

for applying the induction hypothesis. Such a value t' exists because of Lemma 9.3.2. The same instantiation has to be used to apply Lemma 9.2.7 (execution fragments are preserved by concatenation). □

There is one more theorem needed than in the previous section, because the existence of a corresponding start state has to be established. This is guaranteed by the is-simulation property. Reasoning about the third additional use of the choice operator, which chooses the corresponding start state, is facilitated by the following lemma.

Lemma 9.3.5 (Existence of Corresponding Start State)
Given a forward simulation relation R between C and A and a start state s of C, there is always a start state s' of A with $(s, s') \in R$.

$$\frac{\mathsf{is\text{-}simulation}\ R\ C\ A \quad s \in \mathsf{starts\text{-}of}(C)}{\begin{array}{ll}\mathbf{let} & s' = \varepsilon\ s'.\ (s, s') \in R \wedge s' \in\ \mathsf{starts\text{-}of}(A) \\ \mathbf{in} & (s, s') \in R \wedge s' \in\ \mathsf{starts\text{-}of}(A)\end{array}}$$

Proof.
Follows essentially from $(\exists x.\ P\ x) \wedge (\forall x.\ P(x) \Rightarrow Q(x)) \Rightarrow Q\ (\varepsilon\ y.\ P\ y)$. □

Now the main soundness result for forward simulations can be established.

Theorem 9.3.6 (Soundness of Forward Simulations)
The traces of A include those of C if there is a forward simulation from C to A and the external actions are the same.

$$\frac{\mathcal{A}_1\text{: } \mathsf{is\text{-}simulation}\ R\ C\ A \quad \mathcal{A}_2\text{: } \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A)}{\mathcal{C}_1\text{: } C \preceq_S A}$$

Proof.
By the definition of $\preceq_S$ $\mathcal{C}_1$ reduces to $\mathcal{C}_2$: $\mathsf{traces}(C) \subseteq \mathsf{traces}(A)$ which reduces further by elementary set equalities, some propositional logic, and the characterization of traces (Lemma 8.3.6) to

$$\begin{array}{ll}\mathcal{C}_3\text{:} & exec_1 \in \mathsf{executions}\ C \\ & \Rightarrow \exists exec_2 \in \mathsf{executions}\ A.\ \mathsf{mk\text{-}trace}\ C\ \text{`}(\mathsf{snd}\ exec_1) = \mathsf{mk\text{-}trace}\ A\ \text{`}(\mathsf{snd}\ exec_2)\end{array}$$

Thus, the existence of an execution $exec_2 \in \mathsf{executions}(A)$ has to be shown, that has the same trace as $exec_1$. Such an execution is exactly the corresponding execution of $exec_1$ given by $exec_2 := \mathsf{corresp}^{\mathsf{sim}}\, A f\; exec_1$. Therefore, by writing (s_1, ex_1) for $exec_1$, using the equations for is-exec-frag, and unfolding the definition of executions, $\mathcal{C}_3$ reduces to the propositions

$$\begin{aligned}
&\mathcal{C}_4\!: \quad \mathsf{corresp}^{\mathsf{sim}}\, A\, f\ (s_1, ex_1) \in \mathsf{executions}\ A\\
&\mathcal{C}_5\!: \quad \mathsf{mk\text{-}trace}\ C\ `ex_1 = \mathsf{mk\text{-}trace}\ A\ `(\mathsf{snd}\ (\mathsf{corresp}^{\mathsf{sim}}\, A\, f\ (s_1, ex_1)))
\end{aligned}$$

which have to be shown under the assumption

$$\mathcal{A}_3\!: \quad s_1 \in \mathsf{starts\text{-}of}(C) \wedge \mathsf{is\text{-}exec\text{-}frag}\ C\ (s_1, ex_1)$$

Until now, the proof is analogous to the one for $\mathsf{corresp}^{\mathsf{ref}}$. But for the application of the Lemmas 9.3.3 (Trace Equality) and 9.3.4 (Execution Property) the preceding Lemma 9.3.5 has to be employed at three places. It is used to guarantee the existence of a corresponding start state and to justify the additional assumption $(s, s') \in R$ for the two main Lemmas 9.3.3 and 9.3.4. In detail:

For $\mathcal{C}_5$, Lemma 9.3.3 can be applied using $\mathcal{A}_1$–$\mathcal{A}_3$, *reachable*-0 (Def. 8.2.7), and Lemma 9.3.5. For $\mathcal{C}_4$, it has to be shown that

$$(\varepsilon\, s'.\ (s, s') \in R \wedge s' \in\ \mathsf{starts\text{-}of}\ A) \in \mathsf{starts\text{-}of}\ A$$

which is implied by $\mathcal{A}_1$ together with $\mathcal{A}_3$ and Lemma 9.3.5. Then Lemma 9.3.4 can be applied using $\mathcal{A}_1$, $\mathcal{A}_3$, and once more Lemma 9.3.5. □

Again a soundness result for the fair case is obtainable in an analogous way.

Theorem 9.3.7 (Soundness of Fair Forward Simulations)
A implements C fairly if there is a fair forward simulation from C to A and the external actions are the same.

$$\frac{\mathsf{is\text{-}fair\text{-}simulation}\, f\ C\ A \quad \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A)}{C \preceq_F A}$$

Proof.
The proof is analogous to the soundness proof of fair refinement mappings (Lemma 9.2.10). For the main lemmas concerning trace equality and correspondence of executions Lemmas 9.3.3 and 9.3.4 are used instead of Lemmas 9.2.6 and 9.2.8. □

9.4 Conclusion and Related Work

The meta-theoretic proofs of this chapter provide a refinement framework which has been verified completely within the theorem prover. Note that the proofs had only to be done once and for all. Concerning the completeness of the verified refinement notions merely backward simulations have not been considered. As backward simulations have the same expressiveness as refinement mappings together with prophecy variables [LV95], merely the correctness of prophecy variables has not been verified.

As the description in this chapter showed, the proofs are rather succinct. In addition, they are easy to automate because they mainly employ structural induction on sequences which appears to be especially suitable for automatic simplification. The soundness proof of forward simulations includes 104 proof commands on 4 pages and therefore seems to be very concise compared to the paper proof of [GSSL93] of about 5 pages (only counting the relevant parts, as backward simulations have been considered there as well).

This seems to be surprising, and in the following chapter we will indeed make the opposite experience. However, it is due to the different sequence models which underly the respective approaches. Whereas we use domain-theoretic lazy lists, in [GSSL93] partial functions on natural numbers are employed. In [GSSL93] the correspondence between two executions of C and A is given *descriptively* by a (rather sophisticated) index mapping that relates the indexes of a step and its corresponding move (Def. 2.4.5). The corresponding execution is then calculated as a limit construction of index intervals. We, however, define the corresponding execution *constructively* as the infinite concatenation of the corresponding moves. Thus, indexes do not have to be considered at all.

In addition, our algebraic sequence model is in particular suitable to treat infinite concatenation, which is demonstrated by the simple fixpoint definition of $\mathsf{corresp}^{\mathsf{ref}}$. This is especially true compared to models that formalize sequences as functions, as discussed in §7.

Related Work: Inspired by our work, Griffioen and Devillers [GD98] formalized the meta-theory of I/O automata in PVS [ORSH95] and proved the correctness of safe refinement mappings, but not of forward simulations. However, defining and reasoning about infinite concatenation turned out to be very awkward in their functional sequence model [DG97] (cf. the discussion in §7 and [DGM97]). Thus, they are currently investigating whether they should switch to a coalgebraic sequence model [HJ97a].

Our verification framework is a modification and significant extension of [NS95], where sequences are described as functions of type $\mathsf{nat} \to (\alpha)\,\mathsf{option}$ without normalizing (cf. §7.2). As explained in §7.6, such a sequence model does not permit to prove refinement concepts which are more general than weak refinement mappings. This applies similarly to the restricted notion of refinement supported in the tool AutoFocus [HSSS96], which is planned to be integrated with Isabelle.

Chapter 10

Compositionality and Non-Interference

In this chapter we prove compositionality and non-interference for safe I/O automata with Isabelle. In contrast to the short proof in the literature, which is based on an informal sequence model, our rigorous proof shows considerable complexity. This is due to noncontinuous functions and admissibility problems, i.e. complications which are caused by the formal sequence model and thereby are neglected in the informal proof. In particular, the overall proof outline has to be modified slightly in order to circumvent a noncontinuous fair merge function. Because of its complexity the proof enables a meaningful evaluation of the usability of HOLCF.

10.1 Introduction

In this chapter we establish compositionality (Thm. 2.4.10) and non-interference (Thm. 2.4.11) for safe I/O automata within Isabelle. Compositional reasoning is a major requirement for every reasonable verification formalism (cf. [dRe97]) and means in the context of I/O automata that it suffices to show the existence of implementation relations for the *components* of a system. This already implies that there is an implementation relation for the *entire* system. More precisely we will prove in §10.5 that for compatible safe I/O automata A_1, A_2, B_1, B_2 holds:

$$\frac{A_1 \preceq_S A_2 \qquad\qquad B_1 \preceq_S B_2}{(A_1 \parallel B_1) \preceq_S (A_2 \parallel B_2)}$$

As the implementation relation $\preceq_S$ is defined via trace inclusion this theorem boils down to the fact that the traces of a parallel composition are uniquely determined by the traces of its components. Therefore, in §10.4 the following theorem will be proved

$$\mathsf{Traces}\,(A \parallel B) \;\;=\;\; \mathsf{Traces}(A) \parallel_{tr} \mathsf{Traces}(B)$$

where $\|_{tr}$ essentially means set intersection and A and B are compatible I/O automata. This theorem in turn relies on similar theorems for schedules and executions

$$\begin{array}{lcl} \mathsf{Scheds}\,(A \parallel B) & = & \mathsf{Scheds}(A) \parallel_{sch} \mathsf{Scheds}(B) \\ \mathsf{Execs}\,(A \parallel B) & = & \mathsf{Execs}(A) \parallel_{ex} \mathsf{Execs}(B) \end{array}$$

for appropriate operators $\|_{sch}$ and $\|_{ex}$. These theorems will be proved in §10.3 and §10.2, respectively. In the sequel these properties are referred to as *compositionality for trace/ schedule/execution properties*.

In §10.6 we prove non-interference for safe I/O automata, which means that parallel composition preserves the property that a system controls the performance of its local actions. Clearly, this property is a consequence of the compositionality for schedules and the input-enabledness of I/O automata.

The proofs represent a nontrivial challenge for HOLCF, as we face a noncontinuous fair merge function and have to deal with proof obligations which can hardly be proved to be admissible. The first problem forced us to modify the overall proof outline of the informal compositionality proof sketch in [LT87], the second was the motivation for the coalgebraic infrastructure for sequences developed in §6.5 and §6.6. In more detail:

Fair Merge The key idea of the compositionality proofs is to paste together executions (resp. schedules) of components in such a way, that an execution (resp. schedule) of the composition is obtained. This can be accomplished by some kind of fair merge function, which is not continuous in general. Fortunately, for executions an oracle can be carried along which guarantees the fairness and therefore, makes the merge process continuous. For schedules, however, such an oracle can only predict the membership of *external* actions. This implies that possibly occurring internal actions after the last external one have to be merged without oracle and therefore noncontinuously. Our solution is to modify the proof in such a way, that it is possible to work with schedules whose internal rests have been cut off. This cut-off operator is exactly the Chop operator introduced in §6.6, and therefore demonstrates the need for functions, which are more naturally characterized corecursively.

Admissibility In the compositionality proofs for schedules and traces propositions occur whose admissibility cannot be discharged automatically by the admissibility tactic. Furthermore, proving admissibility by its definition would boil down to essentially proving the entire proposition. For schedules we get along with tricky reformulations of the initial goal. For traces, however, this is not possible. Instead, we take advantage of the infrastructure for take lemma proofs concerning recursively defined operators provided in §6.5.

In §10.7 we report on the experiences with these proofs and draw conclusions w.r.t. the power and applicability of HOLCF.

10.2 Compositionality for Executions

This section describes the compositionality result on the execution level. First, some operations on executions have to be introduced.

Definition 10.2.1 (Execution Projections)
Executions of parallel compositions may be projected onto their first or second component. This is done by ProjA and ProjB, respectively:

$$
\begin{array}{lcl}
\mathsf{ProjA} & :: & (\alpha, \sigma \times \tau)\,\mathsf{execution} \to (\alpha, \sigma)\,\mathsf{execution} \\
\mathsf{ProjA}\,(s, ex) & \equiv & (\mathsf{fst}\; s, \mathsf{ProjA_c}\,\text{‘}ex) \\[4pt]
\mathsf{ProjA_c} & :: & (\alpha \times (\sigma \times \tau))\,\mathsf{sequence} \to_c (\alpha \times \sigma)\,\mathsf{sequence} \\
\mathsf{ProjA_c} & \equiv & \mathsf{Map}\,(\lambda(a, t).\,(a, \mathsf{fst}\; t)) \\[4pt]
\mathsf{ProjB} & :: & (\alpha, \sigma \times \tau)\,\mathsf{execution} \to (\alpha, \tau)\,\mathsf{execution} \\
\mathsf{ProjB}\,(s, ex) & \equiv & (\mathsf{snd}\; s, \mathsf{ProjB_c}\,\text{‘}ex) \\[4pt]
\mathsf{ProjB_c} & :: & (\alpha \times (\sigma \times \tau))\,\mathsf{sequence} \to_c (\alpha \times \tau)\,\mathsf{sequence} \\
\mathsf{ProjB_c} & \equiv & \mathsf{Map}\,(\lambda(a, t).\,(a, \mathsf{snd}\; t))
\end{array}
$$

□

Definition 10.2.2 (Filtering Executions)
The operation Filter-ex removes from an execution each action not in a given signature together with its following state:

$$
\begin{array}{lcl}
\mathsf{Filter\text{-}ex} & :: & (\alpha, \sigma)\,\mathsf{signature} \to (\alpha, \sigma)\,\mathsf{execution} \to (\alpha, \sigma)\,\mathsf{execution} \\
\mathsf{Filter\text{-}ex}\; sig\; (s, ex) & \equiv & (s, \mathsf{Filter\text{-}ex_c}\,\text{‘}ex) \\[4pt]
\mathsf{Filter\text{-}ex_c} & :: & (\alpha, \sigma)\,\mathsf{signature} \to (\alpha \times \sigma)\,\mathsf{sequence} \to_c (\alpha \times \sigma)\,\mathsf{sequence} \\
\mathsf{Filter\text{-}ex_c}\; sig & \equiv & \mathsf{Filter}\,(\lambda(a, s).\; a \in \mathsf{actions}(sig))
\end{array}
$$

□

Corollary 10.2.3
The following rewrite rules for $\mathsf{Filter\text{-}ex_c}$ follow immediately from those for Filter:

$$
\begin{array}{lcl}
\mathsf{Filter\text{-}ex_c}\; sig\,\text{‘}\bot & = & \bot \\
\mathsf{Filter\text{-}ex_c}\; sig\,\text{‘}\mathsf{nil} & = & \mathsf{nil} \\
\mathsf{Filter\text{-}ex_c}\; sig\,\text{‘}(a, t)\hat{}\,ex & = & \textbf{if}\; (\mathsf{fst}\; a) \in \mathsf{actions}(sig)\; \textbf{then}\; (a, t)\hat{}\,\mathsf{Filter\text{-}ex_c}\; sig\,\text{‘}ex \\
 & & \qquad\qquad\qquad\qquad\quad\;\; \textbf{else}\qquad\;\; \mathsf{Filter\text{-}ex_c}\; sig\,\text{‘}ex
\end{array}
$$

The operation Filter-ex is defined over a signature instead of an I/O automaton. This allows us to use it for sets of executions, which are not necessarily generated by I/O automata, as well. See Def. 10.3.9 where Filter-ex is used in the context of general execution properties.

Furthermore, note that $\mathsf{Filter\text{-}ex_c}$ can directly be defined using Filter because executions are formalized using sequences of action/state *pairs*. In §10.3.3 this allows us to derive nice commutation properties for $\mathsf{Filter\text{-}ex_c}$ and Filter. For a similar reason ProjA and $\mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ A)$ are defined as separate operations in contrast to the literature (cf. §2): more abstract properties can be proven about ProjA without connection to $\mathsf{Filter\text{-}ex}$. Furthermore, ProjA will as well be used in other contexts, for example in connection with the following predicate:

Definition 10.2.4 (Stuttering)
The predicate $\mathsf{Stutter}$ checks for all transitions $s \xrightarrow{a} t$ occurring in an execution whether $s = t$ provided a is not in a given signature:

$$\begin{array}{lcl}\mathsf{Stutter} & :: & (\alpha,\sigma)\,\mathsf{signature} \to (\alpha,\sigma)\,\mathsf{execution} \to \mathsf{bool} \\ \mathsf{Stutter}\ sig\ (s, ex) & \equiv & \mathsf{Stutter_c}\ sig\ `ex\ s \neq \mathsf{FF}\end{array}$$

The $\mathsf{Stutter}$ predicate is realized by the computable operation

$$\mathsf{Stutter_c}\ ::\ (\alpha,\sigma)\,\mathsf{signature} \to (\alpha \times \sigma)\,\mathsf{sequence} \to_c \sigma \to \mathsf{tr}$$

which runs down the sequence examining every transition w.r.t. the desired property. The following equations characterize $\mathsf{Stutter_c}$:

$$\begin{array}{lcl}\mathsf{Stutter_c}\ sig\ `\bot\ s & = & \bot \\ \mathsf{Stutter_c}\ sig\ `\mathsf{nil}\ s & = & \mathsf{TT} \\ \mathsf{Stutter_c}\ sig\ `((a,t)\hat{\ }ex)\ s & = & (\mathbf{if}\ a \notin \mathsf{actions}(sig)\ \mathbf{then}\ \mathsf{Def}(s = t)\ \mathbf{else}\ \mathsf{TT}) \\ & & \mathsf{andalso}\ \ \mathsf{Stutter_c}\ sig\ `ex\ t\end{array}$$

□

Corollary 10.2.5
The following rewrite rules for $\mathsf{Stutter}$ follow immediately from those for $\mathsf{Stutter_c}$:

$$\begin{array}{lcl}\mathsf{Stutter}\ sig\ (s,\bot) & = & \mathsf{True} \\ \mathsf{Stutter}\ sig\ (s,\mathsf{nil}) & = & \mathsf{True} \\ \mathsf{Stutter}\ sig\ (s,(a,t)\hat{\ }ex) & = & (a \notin \mathsf{actions}(sig) \Rightarrow s = t) \wedge \mathsf{Stutter}\ sig\ (t, ex)\end{array}$$

Proof Outline. The following lemmas all contribute to the main theorem of this section, Thm. 10.2.8, which expresses the compositionality of executions (see an example in Fig. 10.1): executions of a composition induce executions of the components by projecting them onto the components' states ($\mathsf{ProjA}, \mathsf{ProjB}$) and filtering out action/state pairs unknown to the components ($\mathsf{Filter\text{-}ex}$). Conversely, suppose *exec* is of type $\mathsf{execution}$. Then *exec* is an execution of $A \parallel B$, if $\mathsf{Filter\text{-}ex}$ and $\mathsf{ProjA}, \mathsf{ProjB}$ induce executions of the components and the $\mathsf{Stutter}$ predicate holds for the projections onto the components. Furthermore, it is required that *exec* contains only actions of the signature of $A \parallel B$.

$exec$	$((s_1^A, s_1^B), [(a_1, s_2^A, s_1^B),(a_2, s_2^A, s_2^B)!])$
Filter-ex (sig-of A) (ProjA $exec$)	$(s_1^A \quad , [(a_1, s_2^A)!] \qquad)$
Filter-ex (sig-of B) (ProjB $exec$)	$(s_1^B \quad , \qquad [(a_2, s_2^B)!] \quad)$

Figure 10.1: An Execution of $A \parallel B$ and its Projections on A and B.

Proof Automation. The proofs in the sequel could be automated very well, as negations cannot make the admissibility test fail: all involved predicates (is-exec-frag, Stutter, and Forall) are defined via the common scheme $f\,`ex \neq FF$. As explained in §6.3 terms satisfying this scheme are always admissible.

Therefore, the proofs are essentially done by a tactic, called `ex_induct_tac`, which applies structural induction and simplifies all subgoals — base cases, admissibility, and inductive case. For the inductive case elements in executions are split into explicit pairs before simplification. The simplifier is enriched by a set of lemmas that describe the definition of $\parallel$ in detail. The tactic is followed by some predicate-logical reasoning, case-splitting of **if then else** constructs and further tailored simplification, which in most cases finishes the proof.

As the actual proof scripts are very short, the next proof is given exemplarily in more detail, whereas the subsequent lemmas are presented without proof.

Lemma 10.2.6
Executions of $A \parallel B$ imply executions on the components A and B:

$$
\begin{aligned}
\mathcal{C}_1\text{:}\quad & \forall s.\textsf{is-exec-frag}\ (A \parallel B)\ (s, ex) \\
& \quad \Rightarrow \textsf{is-exec-frag}\ A\ (\textsf{Filter-ex}\ (\textsf{sig-of}\ A)\ (\textsf{ProjA}\ (s, ex))) \wedge \\
& \quad\quad \textsf{is-exec-frag}\ B\ (\textsf{Filter-ex}\ (\textsf{sig-of}\ B)\ (\textsf{ProjB}\ (s, ex)))
\end{aligned}
$$

Proof.
Rewriting with Filter-ex and ProjA, ProjB reduces $\mathcal{C}_1$ to

$$
\begin{aligned}
\mathcal{C}_2\text{:}\quad & \forall s.\textsf{is-exec-frag}\ (A \parallel B)\ (s, ex) \\
& \quad \Rightarrow \textsf{is-exec-frag}\ A\ (\textsf{fst}\ s, \textsf{Filter-ex}_{\textsf{c}}\ (\textsf{sig-of}\ A)\ `(\textsf{ProjA}_{\textsf{c}}\ `ex)) \wedge \\
& \quad\quad \textsf{is-exec-frag}\ B\ (\textsf{snd}\ s, \textsf{Filter-ex}_{\textsf{c}}\ (\textsf{sig-of}\ B)\ `(\textsf{ProjB}_{\textsf{c}}\ `ex))
\end{aligned}
$$

This rewriting is necessary, because Filter-ex and ProjA, ProjB cannot be characterized recursively, as the start state precludes a recursive definition.

Now, we apply **structural induction** on ex. The **admissibility** condition is discharged automatically, the **base cases** $ex =$ nil, $ex = \perp$ are trivial.

Inductive step $ex = (a,t)\hat{\ } ex_1$: Assume $\mathcal{C}_2$ as induction hypothesis. Therefore, we get a new goal which has to be shown under a further assumption.

$$\begin{array}{ll} \mathcal{A}_1\!: & s \stackrel{a}{\rightarrow}_{A\|B} t \wedge \textsf{is-exec-frag}\ (A \parallel B)\ (t, ex_1) \\ \mathcal{C}_3\!: & \textsf{is-exec-frag}\ A\ (\textsf{fst}\ s, \textsf{Filter-ex}_\textsf{c}\ (\textsf{sig-of}\ A)\ \text{`}(\textsf{ProjA}_\textsf{c}\ \text{`}(a,t)\hat{\ } ex)) \wedge \\ & \textsf{is-exec-frag}\ B\ (\textsf{snd}\ s, \textsf{Filter-ex}_\textsf{c}\ (\textsf{sig-of}\ B)\ \text{`}(\textsf{ProjB}_\textsf{c}\ \text{`}(a,t)\hat{\ } ex)) \end{array}$$

We prove only the first part of $\mathcal{C}_3$, the second follows by an analogous argument.

- **Case $a \in \textsf{act}(A)$:** In this case, the component A can take a transition, as the projection of every step in the composition $A \parallel B$ is at the same time a step of A provided that the respective action is in the signature of A. Formally, the problem is reduced by the following equalities:

$$\begin{array}{ll} & \textsf{is-exec-frag}\ A\ (\textsf{fst}\ s, \textsf{Filter-ex}_\textsf{c}\ (\textsf{sig-of}\ A)\ \text{`}(\textsf{ProjA}_\textsf{c}\ \text{`}(a,t)\hat{\ } ex)) \\ & \quad \{\textit{Equations for}\ \textsf{ProjA}_\textsf{c}\ \textit{and}\ \textsf{Filter-ex}_\textsf{c}\} \\ = & \textsf{is-exec-frag}\ A\ (\textsf{fst}\ s, (a, \textsf{fst}\ t)\hat{\ }\textsf{Filter-ex}_\textsf{c}\ (\textsf{sig-of}\ A)\ \text{`}(\textsf{ProjA}_\textsf{c}\ \text{`}ex)) \\ & \quad \{\textit{Equations for}\ \textsf{is-exec-frag}\} \\ = & \textsf{fst}\ s \stackrel{a}{\rightarrow}_A \textsf{fst}\ t \wedge \textsf{is-exec-frag}\ A\ (\textsf{fst}\ t, \textsf{Filter-ex}_\textsf{c}\ (\textsf{sig-of}\ A)\ \text{`}(\textsf{ProjA}_\textsf{c}\ \text{`}ex)) \end{array}$$

  The first conjunct of the remaining proposition follows from the definition of $\parallel$ using $s \stackrel{a}{\rightarrow}_{A\|B} t$, which holds by $\mathcal{A}_1$, and $a \in \textsf{act}(A)$. The second conjunct can be discharged by the induction hypothesis by instantiating ex with ex_1 and s with t.

- **Case $a \notin \textsf{act}(A)$:** In this case, the component A stutters according to the definition of the composition $A \parallel B$ in the case where the component does not know the respective action. Therefore, the state is just passed on for the next pair in ex to be checked. Formally, the problem is reduced by the following equalities:

$$\begin{array}{ll} & \textsf{is-exec-frag}\ A\ (\textsf{fst}\ s, \textsf{Filter-ex}_\textsf{c}\ (\textsf{sig-of}\ A)\ \text{`}(\textsf{ProjA}_\textsf{c}\ \text{`}(a,t)\hat{\ } ex)) \\ & \quad \{\textit{Equations for}\ \textsf{ProjA}_\textsf{c}\ \textit{and}\ \textsf{Filter-ex}_\textsf{c}\} \\ = & \textsf{is-exec-frag}\ A\ (\textsf{fst}\ s, \textsf{Filter-ex}_\textsf{c}\ (\textsf{sig-of}\ A)\ \text{`}(\textsf{ProjA}_\textsf{c}\ \text{`}ex)) \end{array}$$

  From $s \stackrel{a}{\rightarrow}_{A\|B} t$, which holds because of $\mathcal{A}_1$, and $a \notin \textsf{act}(A)$ follows $(\textsf{fst}\ s) = (\textsf{fst}\ t)$. Therefore the induction hypothesis can be applied by instantiating ex with ex_1 and s with t.

□

Lemma 10.2.7
Executions of $A \parallel B$ imply the Stutter predicate on the components' executions. Furthermore, they contain only actions of the signature of $A \parallel B$, and represent specific

compositions of executions of the components A and B.

$$
\begin{array}{l}
\forall s. \textsf{is-exec-frag}\ (A \parallel B)\ (s, ex) \\
\quad \Rightarrow \textsf{Stutter}\ (\textsf{sig-of}\ A)\ (\textsf{ProjA}\ (s, ex)) \wedge \\
\quad\quad \textsf{Stutter}\ (\textsf{sig-of}\ B)\ (\textsf{ProjB}\ (s, ex)) \wedge \\
\quad\quad \textsf{Forall}\ (\lambda x.\ \textsf{fst}\ x \in \textsf{act}\ (A \parallel B))\ ex
\end{array}
$$

$$
\begin{array}{l}
\forall s. \textsf{is-exec-frag}\ A\ (\textsf{Filter-ex}\ (\textsf{sig-of}\ A)\ (\textsf{ProjA}\ (s, ex))) \wedge \\
\quad \textsf{is-exec-frag}\ B\ (\textsf{Filter-ex}\ (\textsf{sig-of}\ B)\ (\textsf{ProjB}\ (s, ex))) \wedge \\
\quad \textsf{Stutter}\ (\textsf{sig-of}\ A)\ (\textsf{ProjA}\ (s, ex)) \wedge \\
\quad \textsf{Stutter}\ (\textsf{sig-of}\ B)\ (\textsf{ProjB}\ (s, ex)) \wedge \\
\quad \textsf{Forall}\ (\lambda x.\ \textsf{fst}\ x \in \textsf{act}\ (A \parallel B))\ ex \\
\quad \Rightarrow \textsf{is-exec-frag}\ (A \parallel B)\ (s, ex)
\end{array}
$$

Proof.
Essentially by `ex_induct_tac`. □

Theorem 10.2.8 (Compositionality for Executions)
Executions of $A \parallel B$ are related to the executions of A and B via the following theorem:

$$
\begin{array}{l}
exec \in \textsf{executions}\ (A \parallel B) = \\
\quad \textsf{Filter-ex}\ (\textsf{sig-of}\ A)\ (\textsf{ProjA}\ exec) \in \textsf{executions}(A) \wedge \\
\quad \textsf{Filter-ex}\ (\textsf{sig-of}\ B)\ (\textsf{ProjB}\ exec) \in \textsf{executions}(B) \wedge \\
\quad \textsf{Stutter}\ (\textsf{sig-of}\ A)\ (\textsf{ProjA}\ exec) \wedge \\
\quad \textsf{Stutter}\ (\textsf{sig-of}\ B)\ (\textsf{ProjB}\ exec) \wedge \\
\quad \textsf{Forall}\ (\lambda x.\ \textsf{fst}\ x \in \textsf{act}\ (A \parallel B))\ (\textsf{snd}\ exec)
\end{array}
$$

Proof.
Basic application of Lemma 10.2.7 and Lemma 10.2.6. □

This theorem implicitly contains a construction that describes how executions of a composition can be generated from the executions of the components. To make this explicit, we define a composition operator for sets of executions with a given signature.

Definition 10.2.9 (Execution Property Composition)
Parallel composition on execution properties is defined as follows:

$$\parallel_{ex} \quad : \quad (\alpha, \sigma)\,\textsf{exec-prop} \to (\alpha, \tau)\,\textsf{exec-prop} \to (\alpha, \sigma \times \tau)\,\textsf{exec-prop}$$

$$
\begin{array}{l}
(sig_A, execs_A) \parallel_{ex} (sig_B, execs_B) \equiv \\
\quad \{\textsf{sig-comp}\ sig_A\ sig_B, \\
\qquad \quad \{exec \mid \textsf{Filter-ex}\ sig_A\ (\textsf{ProjA}\ exec) \in execs_A\} \\
\qquad \cap\ \{exec \mid \textsf{Filter-ex}\ sig_B\ (\textsf{ProjB}\ exec) \in execs_B\} \\
\qquad \cap\ \{exec \mid \textsf{Stutter}\ sig_A\ (\textsf{ProjA}\ exec)\} \\
\qquad \cap\ \{exec \mid \textsf{Stutter}\ sig_B\ (\textsf{ProjB}\ exec)\} \\
\qquad \cap\ \{exec \mid \textsf{Forall}\ (\lambda x.\ \textsf{fst}\ x \in (\textsf{actions}\ sig_A \cup \textsf{actions}\ sig_A))\ (\textsf{snd}\ exec)\}\}
\end{array}
$$

This composition operator allows us to rephrase Theorem 10.2.8 in a concise way.

Corollary 10.2.10 (Compositionality of Execution Properties)
Execution properties are compositional:

$$\textsf{Execs}\ (A \parallel B) = \textsf{Execs}(A) \parallel_{ex} \textsf{Execs}(B)$$

10.3 Compositionality for Schedules

In order to guide the reader through the subsequent proofs, we start with the main theorem in a top down fashion. Its proof will reveal the main verification tasks and thereby motivate the subsequent lemmas.

Theorem 10.3.1 (Compositionality for Schedules)
Schedules of $A \parallel B$ are related to those of A and B via the following theorem:

$$
\begin{array}{l}
sch \in \textsf{schedules}\ (A \parallel B) = \\
\quad \textsf{Filter}\ (\lambda a.\ a \in \textsf{act}\ A)\ \text{`}sch \in \textsf{schedules}(A) \wedge \\
\quad \textsf{Filter}\ (\lambda a.\ a \in \textsf{act}\ B)\ \text{`}sch \in \textsf{schedules}(B) \wedge \\
\quad \textsf{Forall}\ (\lambda a.\ a \in \textsf{act}\ (A \parallel B))\ sch
\end{array}
$$

Proof.
See also Fig. 10.2 and Fig. 10.3, which illustrate the main proof ideas.

"$\Rightarrow$": The assumption $sch \in \textsf{schedules}\ (A \parallel B)$ guarantees the existence of an $exec$ with

$$\mathcal{A}_1: \quad exec \in \textsf{executions}\ (A \parallel B) \wedge sch = \textsf{Filter-act}\ \text{`}(\textsf{snd}\ exec)$$

There are three conjuncts to prove. The third one is trivial, as a schedule of $A \parallel B$ contains only actions from the signature of $A \parallel B$. Formally, that is proved by Lemma 10.2.7, part 1, and the sequence lemma $\textsf{Forall}\ (P \circ f)\ s = \textsf{Forall}\ P\ (\textsf{Map}\, f\ \text{`}s)$.

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
 & \text{Executions} & \text{Schedules} \\
A & exec_A \arrow[r, "\mathsf{Filter\text{-}act} \circ \mathsf{snd}"] & sch_A \\
A \parallel B & exec \arrow[u, "\mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ A) \circ \mathsf{ProjA}"] \arrow[r, "\mathsf{Filter\text{-}act} \circ \mathsf{snd}"] \arrow[d, "\mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ B) \circ \mathsf{ProjB}"] & sch \arrow[u, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)"] \arrow[d, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B)"] \\
B & exec_B \arrow[r, "\mathsf{Filter\text{-}act} \circ \mathsf{snd}"] & sch_B
\end{tikzcd}
$$

Figure 10.2: Compositionality for Schedules

Therefore, we restrict our intention to the first conjunct; the second one is proved analogously. It has to be shown that there is an $exec_A$ with

$$
\begin{array}{ll}
\mathcal{C}_1: & exec_A \in \mathsf{executions}(A) \\
\mathcal{C}_2: & \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ exec_A) = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `sch
\end{array}
$$

We define $exec_A$ to be $\mathsf{Filter\text{-}ex}\,(\mathsf{sig\text{-}of}\ A)\,(\mathsf{ProjA}\ exec)$. Then, $\mathcal{C}_1$ is directly discharged by the compositionality result for executions (Thm. 10.2.8) using $\mathcal{A}_1$. $\mathcal{C}_2$ turns into

$$
\begin{array}{ll}
\mathcal{C}_3: & \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ (\mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ A)\ (\mathsf{ProjA}\ exec))) \\
 & = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `(\mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ exec))
\end{array}
$$

by rewriting with $sch = \mathsf{Filter\text{-}act}`(\mathsf{snd}\, exec)$ from $\mathcal{A}_1$ and with the definition of $exec_A$. The resulting proposition $\mathcal{C}_3$ means that the diagram in Fig. 10.2 commutes. This is essentially shown by Lemma 10.3.4 which will be proved later on. In detail, the proof proceeds as follows, where (s, ex) is written for *exec*:

$$
\begin{array}{cl}
 & \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ (\mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ A)\ (\mathsf{ProjA}\ (s, ex)))) \\
= & \quad \{\textit{Def. 10.2.2 (}\mathsf{Filter\text{-}ex}\textit{), Def. 10.2.1 (}\mathsf{ProjA}\textit{)}\} \\
 & \mathsf{Filter\text{-}act}\ `(\mathsf{Filter\text{-}ex_c}\ (\mathsf{sig\text{-}of}\ A)\ `(\mathsf{ProjA_c}\ `ex)) \\
= & \quad \{\textit{Lemma 10.3.4 } (1)\ \textit{(Filtering and action projection commutes)}\} \\
 & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `(\mathsf{Filter\text{-}act}\ `(\mathsf{ProjA_c}\ `ex)) \\
= & \quad \{\textit{Lemma 10.3.4 } (2)\ \textit{(State and action projections are independent)}\} \\
 & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `(\mathsf{Filter\text{-}act}\ `ex)
\end{array}
$$

"$\Leftarrow$": The assumptions imply that there are executions $exec_A$ and $exec_B$ with

$$
\begin{array}{ll}
\mathcal{A}_1: & exec_A \in \mathsf{executions}(A) \wedge \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ exec_A) = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `sch \\
\mathcal{A}_2: & exec_B \in \mathsf{executions}(B) \wedge \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ exec_B) = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B)\ `sch \\
\mathcal{A}_3: & \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ (A \parallel B))\ sch
\end{array}
$$

It has to be shown that there is an execution *exec* with

$$
\begin{array}{ll}
\mathcal{C}_1: & exec \in \mathsf{executions}\ (A \parallel B) \\
\mathcal{C}_2: & \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ exec) = sch
\end{array}
$$

A major challenge is to define such an *exec* appropriately. Basically $exec_A$ and $exec_B$ have to be pasted together in such a way, that the order of the actions is given by *sch*. The state components of $exec_A$ and $exec_B$ have to be paired, and when the order of actions of $exec_A$ or $exec_B$ does not fit with that of *sch*, stutter states have to be inserted. This is done by mkex via a recursive definition (Def. 10.3.2) which is explained in detail later on. Therefore, we define: $exec := \mathsf{mkex}\ A\ B\ sch\ exec_A\ exec_B$. To prove $\mathcal{C}_1$ we use the compositionality result for executions (Thm. 10.2.8) and get

$$
\begin{array}{ll}
\mathcal{C}_3: & \mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ A)\ (\mathsf{ProjA}\ exec) \in \mathsf{executions}(A) \\
\mathcal{C}_4: & \mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ B)\ (\mathsf{ProjB}\ exec) \in \mathsf{executions}(B) \\
\mathcal{C}_5: & \mathsf{Stutter}\ (\mathsf{sig\text{-}of}\ A)\ (\mathsf{ProjA}\ exec) \\
\mathcal{C}_6: & \mathsf{Stutter}\ (\mathsf{sig\text{-}of}\ B)\ (\mathsf{ProjB}\ exec) \\
\mathcal{C}_7: & \mathsf{Forall}\ (\lambda x.\ \mathsf{fst}\ x \in \mathsf{act}\ (A \parallel B))\ (\mathsf{snd}\ exec)
\end{array}
$$

Instead of $\mathcal{C}_3$ and $\mathcal{C}_4$ we prove

$$
\begin{array}{ll}
\mathcal{C}_8: & \mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ A)\ (\mathsf{ProjA}\ exec) = exec_A \\
\mathcal{C}_9: & \mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ B)\ (\mathsf{ProjB}\ exec) = exec_B
\end{array}
$$

which suffices because of the first conjuncts of $\mathcal{A}_1$ and $\mathcal{A}_2$. All remaining propositions $\mathcal{C}_2$ and $\mathcal{C}_5$ – $\mathcal{C}_9$ can be proved by structural induction using the assumptions

$\mathcal{A}_1$ – $\mathcal{A}_3$ after some tricky reformulations. This will be done in the Lemmas 10.3.5 – 10.3.8. Note that reformulations of several of the propositions $\mathcal{C}_2$ – $\mathcal{C}_7$ are necessary, as otherwise the second conjuncts of the assumptions $\mathcal{A}_2$ and $\mathcal{A}_3$ would not be dischargable by the admissibility test. To find the right reformulations was another major challenge of the proof. □

Therefore, two major challenges of the preceding proof have been identified: the definition of a function that merges two executions and a schedule in a specific way, and the circumvention of formulae which are not amenable for the admissibility test. The following description will mainly focus on these points by discussing them in the next subsections §10.3.1 and §10.3.2, respectively. The remaining lemmas are then presented in §10.3.3.

10.3.1 A specific Merge Function

Basically, a function mkex has to be defined which pastes two executions $exec_A$ of A and $exec_B$ of B together in such a way, that the order of the actions is given by a schedule sch. This schedule sch contains actions from both A and B and serves as some kind of oracle in the merge process.

Before giving the recursive definition in HOLCF, we present the semi-formal definition as formulated in [LT87], where a semiformal proof of compositionality is given. The syntax used for I/O automata and informal sequences stems from the introduction in §2.

> *Let $A = A_1 \parallel \cdots \parallel A_n$. Suppose $sch = a_1 a_2 \cdots$ is a schedule of A and $exec_i$ are executions of A_i which have the schedule $sch\lceil A_i$ for all i. Then we can write $exec_i = s^i_0 a_{i_1} s^i_1 a_{i_2} s^i_2 \cdots$ where $i_0 = 0$. Now the result of merging $exec_i$ and sch is defined as $exec := s_0 a_1 s_1 \cdots$ where the s_i are defined as follows: If $i_k \leq j \leq i_{k+1}$, then $s_j\lceil A_i = s^i_k$. That is, the automaton A_i remains in state s^i_k between the performance of actions a_{i_k} and $a_{i_{k+1}}$, and changes state to s^i_{k+1} upon the performance of $a_{i_{k+1}}$.*

There is, however, a significant problem with this definition, as it assumes that the $exec_i$ can be reformulated as $exec_i = s^i_0 a_{i_1} s^i_1 a_{i_2} s^i_2 \cdots$, i.e. that the elements can be reassigned to the order given by sch. It is by no means clear, how to ensure this possibility without giving an explicit merge function which builds the desired $exec$ constructively. In the following we define an operator mkex, which represents such a constructive merge function.

Informally, mkex merges (s_A, ex_A), (s_B, ex_B) and sch as follows (see also Fig. 10.3). First, the start states s_A and s_B are paired. Then, a possibly infinite recursive merging process starts, which is explained exemplary with the first step. Suppose a is the first action of sch and (a_A, t_A) and (a_B, t_B) are the first elements of ex_A and ex_B, respectively. Then the

$$\begin{tikzcd}
 & \text{Executions} & \text{Schedules} \\
A & s_A\ a\ t_A \arrow[r] & a \\
A \parallel B & (s_A, s_B)\ a\ (t_A, s_B)\ b\ (t_A, t_B) \arrow[u] \arrow[d] \arrow[r] & a\ b \arrow[u] \arrow[d] \\
B & s_B\ b\ t_B \arrow[r] & b
\end{tikzcd}$$

Figure 10.3: Compositionality for Schedules – An Example

first element of the merged sequence is (a, t_A, t_B) in the case that a is an action of both A and B. Otherwise, if a is not an action of, say A, then the state component of A has to stutter. This means, that instead of t_A the previous state in the recursive process s_A is taken.

Note the tricky point why mkex is computable and therefore expressible as a continuous function in contrast to an ordinary fair merge of ex_A and ex_B. The oracle sch is not only used to predict the order of the action components, but also to predict if one of the executions ex_A or ex_B has become empty or equals $\perp$. This is possible because of the following "environment assumptions" which can always be presumed when dealing with mkex:

$$\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(X))\ \text{`}sch = \mathsf{Filter\text{-}act}\ \text{`}ex_X \qquad (EA)$$

where X stands for A or B. This means, that the action subsequences of $exec_X$ and sch filtered onto the signature of X are the same. Therefore, a look on sch suffices to determine if a further action exists for ex_A or ex_B. Therefore, whereas a function *unfairmerge*, which merges ex_A and ex_B in an strict, unfair way, would always yield $unfairmerge\ ex_A \perp = \perp$, for mkex the following holds:

$$\frac{\mathsf{Filter\text{-}act}\ \text{`}ex_B = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B)\ \text{`}sch \quad sch \neq \perp}{\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch\ (s_A, ex_A)\ (s_B, \perp)) \neq \perp}$$

Informally speaking, the oracle sch guarantees that mkex is able to merge ex_A and ex_B in a fair way without "running on $\perp$" if applied in our setting. Consequently, all lemmas mentioning mkex will assume these environment assumptions, but in the following weaker form because of certain admissibility problems to be discussed later:

$$\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(X))\ \text{`}sch \sqsubseteq \mathsf{Filter\text{-}act}\ \text{`}ex_X$$

Note that this weaker form suffices to assure the same reasonable behavior for mkex, as we merely need the implication of the existence of actions in sch to that in ex_X and not vice versa.

Definition 10.3.2 (Merge of Executions)
Given two executions (s_A, ex_A) and (s_B, ex_B) of A and B and a schedule sch of $A \parallel B$ the function mkex merges them as just described informally.

$$\begin{aligned}\mathsf{mkex} \ :: \ & (\alpha,\sigma)\,\mathsf{ioa} \to (\alpha,\tau)\,\mathsf{ioa} \to \alpha\,\mathsf{sequence} \\ & \to (\alpha,\sigma)\,\mathsf{execution} \to (\alpha,\tau)\,\mathsf{execution} \to (\alpha,\sigma\times\tau)\,\mathsf{execution}\end{aligned}$$

The start states are paired. Furthermore, the possibly infinite merge of the sequence parts is accomplished by the operation $\mathsf{mkex_c}$

$$\mathsf{mkex}\ A\ B\ sch\ (s_A, ex_A)\ (s_B, ex_B) \ \equiv\ ((s_A, s_B), \mathsf{mkex_c}\ A\ B\ `sch\ `ex_A\ `ex_B\ s_A\ s_B)$$

which is defined as follows:

$$\begin{aligned}\mathsf{mkex_c} \ :: \ & (\alpha,\sigma)\,\mathsf{ioa} \to (\alpha,\tau)\,\mathsf{ioa} \to \alpha\,\mathsf{sequence} \\ & \to_c (\alpha\times\sigma)\,\mathsf{sequence} \to_c (\alpha\times\tau)\,\mathsf{sequence} \to \sigma \to \tau \to (\alpha\times\sigma\times\tau)\,\mathsf{execution}\end{aligned}$$

The following rewrite rules have been derived from the fixpoint definition:

$$\mathsf{mkex_c}\ A\ B\ `\bot\ `ex_A\ `ex_B\ s_A\ s_B = \bot$$

$$\mathsf{mkex_c}\ A\ B\ `\mathsf{nil}\ `ex_A\ `ex_B\ s_A\ s_B = \mathsf{nil}$$

$$\frac{a \in \mathsf{act}(A) \quad a \notin \mathsf{act}(B)}{\begin{array}{l}\mathsf{mkex_c}\ A\ B\ `(a\hat{\ }sch)\ `(at\hat{\ }ex_A)\ `ex_B\ s_A\ s_B = \\ \quad (a, \mathsf{snd}\ at, s_B)\hat{\ }\mathsf{mkex_c}\ A\ B\ `sch\ `ex_A\ `ex_B\ (\mathsf{snd}\ at)\ s_B\end{array}}$$

$$\frac{a \notin \mathsf{act}(A) \quad a \in \mathsf{act}(B)}{\begin{array}{l}\mathsf{mkex_c}\ A\ B\ `(a\hat{\ }sch)\ `ex_A\ `(at\ ex_B)\ s_A\ s_B = \\ \quad (a, s_A, \mathsf{snd}\ at)\hat{\ }\mathsf{mkex_c}\ A\ B\ `sch\ `ex_A\ `ex_B\ s_A\ (\mathsf{snd}\ at)\end{array}}$$

$$\frac{a \in \mathsf{act}(A) \quad a \in \mathsf{act}(B)}{\begin{array}{l}\mathsf{mkex_c}\ A\ B\ `(a\hat{\ }sch)\ `(at_1\hat{\ }ex_A)\ `(at_2\hat{\ }ex_B)\ s_A\ s_B = \\ \quad (a, \mathsf{snd}\ at_1, \mathsf{snd}\ at_2)\hat{\ }\mathsf{mkex_c}\ A\ B\ `sch\ `ex_A\ `ex_B\ (\mathsf{snd}\ at_1)\ (\mathsf{snd}\ at_2)\end{array}}$$

□

Corollary 10.3.3
The following rewrite rules for mkex follow immediately from those for $\mathsf{mkex_c}$:

$$\mathsf{mkex}\ A\ B\ \bot\ (s_A, ex_A)\ (s_B, ex_B) = ((s_A, s_B), \bot)$$

$$\mathsf{mkex}\ A\ B\ \mathsf{nil}\ (s_A, ex_A)\ (s_B, ex_B) = ((s_A, s_B), \mathsf{nil})$$

$$\frac{a \in \mathsf{act}(A) \quad a \notin \mathsf{act}(B)}{\begin{array}{l}\mathsf{mkex}\ A\ B\ (a\hat{\ }sch)\ (s_A, at\hat{\ }ex_A)\ (s_B, ex_B) = \\ \quad ((s_A, s_B), (a, \mathsf{snd}\ at, s_B)\hat{\ }\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch\ (\mathsf{snd}\ at, ex_A)\ (s_B, ex_B)))\end{array}}$$

$$\frac{a \notin \mathsf{act}(A) \quad a \in \mathsf{act}(B)}{\begin{array}{l}\mathsf{mkex}\ A\ B\ (a\hat{\ }sch)\ (s_A, ex_A)\ (s_B, at\hat{\ }ex_B) = \\ \quad ((s_A, s_B), (a, s_A, \mathsf{snd}\ at)\hat{\ }\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch\ (s_A, ex_A)\ (\mathsf{snd}\ at, ex_B)))\end{array}}$$

$$\frac{a \in \mathsf{act}(A) \quad a \in \mathsf{act}(B)}{\begin{array}{l}\mathsf{mkex}\ A\ B\ (a\hat{\ }sch)\ (s_A, at_1\hat{\ }ex_A)\ (s_B, at_2\hat{\ }ex_B) = \\ \quad ((s_A, s_B), (a, \mathsf{snd}\ at_1, \mathsf{snd}\ at_2)\hat{\ }\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch\ (\mathsf{snd}\ at_1, ex_A)\ (\mathsf{snd}\ at_2, ex_B)))\end{array}}$$

10.3.2 Admissibility Problems

Typical properties to be verified in this and the next section consist of predicates over sequences which hold only under certain sequence equalities. More precisely, properties $P\ x\ y$ have the form

$$f(x) = g(y) \Rightarrow A\ x\ y$$

where $x :: (\alpha)\,\mathsf{sequence}$ and $y :: (\beta)\,\mathsf{sequence}$ represent sequences. In our context, the equation $f(x) = g(y)$ may represent one of the environment assumptions (EA) for mkex. The admissibility of such predicates, however, cannot be discharged by the admissibility test as the syntactic rules simplify

$$\mathsf{adm}\ (\lambda x.\ P\ x\ y)$$

to

$$\mathsf{adm}\ (\lambda x.\ f(x) \neq g(y)) \vee \mathsf{adm}\ (\lambda x.\ A\ x\ y)$$

of which the first disjunct cannot be simplified further. Furthermore, proving admissibility manually by its definition would be very hard in this case, as it would boil down to establishing the property P itself for at least infinite streams x.

In this section the problems could be circumvented by the following two solutions:

- Often it was possible to strengthen the proposition P by weakening the assumption from $f(x) = g(y)$ to $f(x) \sqsubseteq g(y)$. Then the rule

$$\frac{\mathsf{cont}(f)}{\mathsf{adm}\ (\lambda x.\ f(x) \not\sqsubseteq c)}$$

which is part of the standard admissibility test (rule (6) in §5.1.2) can be used to automatically discharge the admissibility requirement. In our concrete application, this weakening is possible for the environment assumptions (EA).

- The other solution searches, roughly speaking, for the inverse function g^{-1} of g and proves $y = g^{-1}(f\,x) \Rightarrow A\,x\,y$ or even simpler $A\,x\,(g^{-1}(f\,x))$ instead of $P\,x\,y$. Obviously, in most cases there will not be an inverse function of g. In this section, however, g is always of the form $\mathsf{Map\,fst}$, and thus something similar to an inverse function can be constructed: we exploit the fact that

$$(\mathsf{Map\,fst}\text{ `}y \sqsubseteq z) \quad = \quad (y = \mathsf{Zip}\text{ `}z\text{ `}(\mathsf{Map\,snd}\text{ `}y))$$

which is a slightly stronger result than the intuitively clear statement

$$y = \mathsf{Zip}\text{ `}(\mathsf{Map\,fst}\text{ `}y)\text{ `}(\mathsf{Map\,snd}\text{ `}y)$$

Therefore, instead of

$$f(x) = \mathsf{Map\,fst}\text{ `}y \Rightarrow A\ x\ y$$

we prove

$$f(x) \sqsubseteq \mathsf{Map\,fst}\text{ `}y \Rightarrow A\ x\ (\mathsf{Zip}\text{ `}(f\ x)\text{ `}(\mathsf{Map\,snd}\text{ `}y)) \tag{10.1}$$

which is automatically discharged by the admissibility test. Note that this trick is specifically tailored for $g = \mathsf{Map\,fst}$. In particular, it does not work any longer for $g = \mathsf{Filter}\ h$, which will turn out to be the main admissibility problem in the next section.

10.3.3 Lemmas needed for the Main Theorem

As mentioned already, for the simple direction of Thm. 10.3.1 only two lemmas are needed.

Lemma 10.3.4
Projecting actions commutes with filtering actions, respectively action/state pairs. Furthermore, projecting states does not affect projecting actions.

$$\begin{array}{lcll}
\mathsf{Filter\text{-}act}\text{ `}(\mathsf{Filter\text{-}ex_c}\ (\mathsf{sig\text{-}of}\ A)\text{ `}ex) & = & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\text{ `}(\mathsf{Filter\text{-}act}\text{ `}ex) & (1)\\
\mathsf{Filter\text{-}act}\text{ `}(\mathsf{ProjA_c}\text{ `}ex) & = & \mathsf{Filter\text{-}act}\text{ `}ex & (2)\\
\mathsf{Filter\text{-}act}\text{ `}(\mathsf{ProjB_c}\text{ `}ex) & = & \mathsf{Filter\text{-}act}\text{ `}ex &
\end{array}$$

Proof.
The first one boils down to commutativity of Filter and Map, the others are easily proved by structural induction of ex. □

The more complicated direction of Thm. 10.3.1 requires several related proofs of properties of mkex. These proofs could be generalized in such a way, that a specifically designed tactic could be used for all of them. That is the reason why merely the proof of the following lemma is shown exemplarily.

Lemma 10.3.5
Given an execution which results from merging two executions and an oracle sch, projecting onto the action components yields sch again.

$$\begin{aligned}\mathcal{C}_1:\quad \forall ex_A\; ex_B\; s\; t. &\mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ sch\ \wedge\\ &\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}sch \sqsubseteq \mathsf{Filter\text{-}act}\ \text{`}ex_A\ \wedge\\ &\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}sch \sqsubseteq \mathsf{Filter\text{-}act}\ \text{`}ex_B\ \wedge\\ &\Rightarrow \mathsf{Filter\text{-}act}\ \text{`}(\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch\ (s, ex_A)\ (t, ex_B))) = sch\end{aligned}$$

Proof.
The proof is by **structural induction** on sch. The **admissibility** condition can be discharged by the admissibility test, as the assumptions have been weakened according to the considerations in §10.3.2. The **base cases** $sch = \bot$, $sch = \mathsf{nil}$ are trivial.

Inductive step $sch = a\hat{\ }sch'$: Let us take $\mathcal{C}_1$ as induction hypothesis. Therefore, we get a new goal which has to be proved under three further assumptions.

$$\begin{aligned}\mathcal{A}_1:&\quad (a \in \mathsf{act}(A) \vee a \in \mathsf{act}(B)) \wedge \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ sch'\\ \mathcal{A}_2:&\quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(A))\ \text{`}(a\hat{\ }sch') \sqsubseteq \mathsf{Filter\text{-}act}\ \text{`}ex_A\\ \mathcal{A}_3:&\quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(B))\ \text{`}(a\hat{\ }sch') \sqsubseteq \mathsf{Filter\text{-}act}\ \text{`}ex_B\\ \mathcal{C}_2:&\quad \mathsf{Filter\text{-}act}\ \text{`}(\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ (a\hat{\ }sch')\ (s, ex_A)\ (t, ex_B))) = a\hat{\ }sch'\end{aligned}$$

- **Case** $a \in \mathsf{act}(A)$**:**
  - **Case** $a \in \mathsf{act}(B)$**:**
    - **Case** $ex_A = \bot$**:** $\mathcal{A}_2$ rewrites to $a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}sch' \sqsubseteq \bot$ which is false. Therefore the entire proposition is true.
    - **Case** $ex_A = \mathsf{nil}$**:** $\mathcal{A}_2$ rewrites to $a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}sch' \sqsubseteq \mathsf{nil}$ which is false. Therefore the entire proposition is true.
    - **Case** $ex_A = at_1\hat{\ }ex_A'$**:**
      - **Case** $ex_B = \bot$**:** $\mathcal{A}_3$ rewrites to $a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}sch' \sqsubseteq \bot$ which is false. Therefore the entire proposition is true.
      - **Case** $ex_B = \mathsf{nil}$**:** $\mathcal{A}_3$ rewrites to $a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}sch' \sqsubseteq \mathsf{nil}$ which is false. Therefore the entire proposition is true.
      - **Case** $ex_B = at_2\hat{\ }ex_B'$**:** Using the rewrite rules for Filter and Filter-act, $\mathcal{A}_2$ rewrites to

$$\mathcal{A}_4:\quad a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(A))\ \text{`}sch' \sqsubseteq (\mathsf{fst}\ at_1)\hat{\ }\mathsf{Filter\text{-}act}\ \text{`}ex_A'$$

which simplifies further to

$$\mathcal{A}_5\text{:}\quad a = (\mathsf{fst}\ at_1) \wedge \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(A))\ \text{‘}sch' \sqsubseteq \mathsf{Filter\text{-}act}\ \text{‘}ex'_A$$

Similarly, $\mathcal{A}_3$ rewrites to

$$\mathcal{A}_6\text{:}\quad a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(B))\ \text{‘}sch' \sqsubseteq (\mathsf{fst}\ at_2)\hat{\ }\mathsf{Filter\text{-}act}\ \text{‘}ex'_B$$

which simplifies further to

$$\mathcal{A}_7\text{:}\quad a = (\mathsf{fst}\ at_2) \wedge \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(B))\ \text{‘}sch' \sqsubseteq \mathsf{Filter\text{-}act}\ \text{‘}ex'_B$$

Now $\mathcal{C}_2$ may be proved by the following equations:

$$\begin{array}{ll}
 & \mathsf{Filter\text{-}act}\ \text{‘}(\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ (a\hat{\ }sch')\ (s, ex_A)\ (t, ex_B))) \\
= & \{\mathit{Equations\ for}\ \mathsf{mkex}\ \mathit{and}\ \mathsf{Filter\text{-}act}\} \\
 & a\hat{\ }\mathsf{Filter\text{-}act}\ \text{‘}(\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch'\ (\mathsf{snd}\ at_1, ex'_A)\ (\mathsf{snd}\ at_2, ex'_B))) \\
= & \{\mathit{Induction\ hypothesis\ with}\ sch := sch',\ ex_A := ex'_A,\ ex_B := ex'_B,\ \} \\
 & \{s := \mathsf{snd}\ at_1,\ t := \mathsf{snd}\ at_2,\ \mathcal{A}_1,\ \mathcal{A}_5\ \mathit{and}\ \mathcal{A}_7\} \\
 & a\hat{\ }sch'
\end{array}$$

- **Case** $a \notin \mathsf{act}(B)$**:**
  - **Case** $ex_A = \bot$**:** $\mathcal{A}_2$ rewrites to $a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{‘}sch' \sqsubseteq \bot$ which is false. Therefore the entire proposition is true.
  - **Case** $ex_A = \mathsf{nil}$**:** $\mathcal{A}_2$ rewrites to $a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{‘}sch' \sqsubseteq \mathsf{nil}$ which is false. Therefore the entire proposition is true.
  - **Case** $ex_A = at\hat{\ }ex'_A$**:** Using the rewrite rules for Filter and Filter-act $\mathcal{A}_2$ rewrites to

    $$\mathcal{A}_4\text{:}\quad a\hat{\ }\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{‘}sch' \sqsubseteq (\mathsf{fst}\ at)\hat{\ }\mathsf{Filter\text{-}act}\ \text{‘}ex'_A$$

    and simplifies further to

    $$\mathcal{A}_5\text{:}\quad a = (\mathsf{fst}\ at) \wedge \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(A))\ \text{‘}sch' \sqsubseteq \mathsf{Filter\text{-}act}\ \text{‘}ex'_A$$

    Similarly, as $a \notin \mathsf{act}\ B$, $\mathcal{A}_3$ rewrites to

    $$\mathcal{A}_6\text{:}\quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(B))\ \text{‘}sch' \sqsubseteq \mathsf{Filter\text{-}act}\ \text{‘}ex_B$$

    Now $\mathcal{C}_2$ may be proved by the following equations:

    $$\begin{array}{ll}
     & \mathsf{Filter\text{-}act}\ \text{‘}(\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ (a\hat{\ }sch')\ (s, ex_A)\ (t, ex_B))) \\
    = & \{\mathit{Equations\ for}\ \mathsf{mkex}\ \mathit{and}\ \mathsf{Filter\text{-}act}\} \\
     & a\hat{\ }\mathsf{Filter\text{-}act}\ \text{‘}(\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch'\ (\mathsf{snd}\ at, ex'_A)\ (t, ex_B))) \\
    = & \{\mathit{Induction\ hypothesis\ with}\ sch := sch',\ ex_A := ex'_A,\ ex_B := ex_B,\ \} \\
     & \{s := \mathsf{snd}\ at,\ t := t,\ \mathcal{A}_1,\ \mathcal{A}_5,\ \mathit{and}\ \mathcal{A}_6\} \\
     & a\hat{\ }sch'
    \end{array}$$

- **Case** $a \notin \mathsf{act}(A)$**:**
  - **Case** $a \in \mathsf{act}(B)$**:** Analogous to the previous case.
  - **Case** $a \notin \mathsf{act}(B)$**:** This case is a contradiction to the first conjunct of $\mathcal{A}_1$, which finishes the proof. $\mathcal{A}_1$ is needed in the proposition merely because of this argument.

□

As already mentioned, a tactic has been written, called `mkex_induct_tac`, which generalizes this proof and can be applied in the subsequent proofs. The tactic basically applies structural induction, where in the inductive step it distinguishes between the 12 cases described in the previous proof. For every case the simplifier is invoked which is enriched with a series of specific lemmas needed in this context, among them the characterizing equations of possibly occurring recursively defined operations.

Lemma 10.3.6
The projection onto A of an execution, which results from merging two executions of A and B and an oracle sch, fulfills the $\mathsf{Stutter}$ predicate and contains only actions from the signature of $A \parallel B$ provided sch does the same.

$$\begin{array}{l}
\mathsf{Forall}\,(\lambda x.\, x \in \mathsf{act}\,(A \parallel B))\ sch \wedge \\
\mathsf{Filter}\,(\lambda x.\, x \in \mathsf{act}\ A)\ `sch \sqsubseteq \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ ex_A) \wedge \\
\mathsf{Filter}\,(\lambda x.\, x \in \mathsf{act}\ B)\ `sch \sqsubseteq \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ ex_B) \wedge \\
\Rightarrow \mathsf{Stutter}\ (\mathsf{sig\text{-}of}\ A)\ (\mathsf{ProjA}\ (\mathsf{mkex}\ A\ B\ sch\ ex_A\ ex_B)) \wedge \\
\quad \mathsf{Forall}\,(\lambda x.\, \mathsf{fst}\ x \in \mathsf{act}\,(A \parallel B))\ (\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch\ (s, ex_A)\ (t, ex_B)))
\end{array}$$

Proof.
By `mkex_induct_tac`. □

Lemma 10.3.7
Reformulation of Lemma 10.3.8 according to formula (10.1) in order to be amenable for the admissibility check:

$$\begin{array}{l}
\mathsf{Forall}\,(\lambda x.\, x \in \mathsf{act}\,(A \parallel B))\ sch \wedge \\
\mathsf{Filter}\,(\lambda x.\, x \in \mathsf{act}\ A)\ `sch \sqsubseteq \mathsf{Filter\text{-}act}\ `ex_A \wedge \\
\mathsf{Filter}\,(\lambda x.\, x \in \mathsf{act}\ B)\ `sch \sqsubseteq \mathsf{Filter\text{-}act}\ `ex_B \wedge \\
\Rightarrow \mathsf{Filter\text{-}ex_c}\ (\mathsf{sig\text{-}of}\ A)\ `(\mathsf{ProjA_c}\ `(\mathsf{snd}\ (\mathsf{mkex}\ A\ B\ sch\ (s, ex_A)\ (t, ex_B)))) \\
\quad = \mathsf{Zip}\ `(\mathsf{Filter}\,(\lambda x.\, x \in \mathsf{act}\ A)\ `sch)\ `(\mathsf{Map}\ \mathsf{snd}\ `ex_A)
\end{array}$$

Proof.
By `mkex_induct_tac`. Note that the reformulation does not match exactly formula (10.1),

as the equation

$$ex_A = \mathsf{Zip}\ `(\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `sch)\ `(\mathsf{Map}\,\mathsf{snd}\ `ex_A)$$

is not substituted for every occurrence of ex_A in the conclusion, but only on the *rhs* . This is sufficient to perform the inductive proof and, on the other side, does not blow up the statement. □

Lemma 10.3.8
Given an execution which results from merging two executions ex_A and ex_B and an oracle, projecting it onto the states of A and filtering out action/state pairs, whose action part is not in the signature of A, yields ex_A again.

$$\begin{array}{l}
\mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ sch \wedge \\
\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `sch = \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ ex_A) \wedge \\
\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ `sch = \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ ex_B) \wedge \\
\Rightarrow \mathsf{Filter\text{-}ex}\ (\mathsf{sig\text{-}of}\ A)\ (\mathsf{ProjA}\ (\mathsf{mkex}\ A\ B\ sch\ ex_A\ ex_B)) = ex_A
\end{array}$$

Proof.
By Lemma 10.3.7 using the admissibility considerations in §10.3.2. Note that the equality in the assumptions cannot be replaced by $\sqsubseteq$ in this case, as otherwise the statement would not hold for $sch = \perp$ and $sch = \mathsf{nil}$, which means that in the inductive proof the base cases fail. Intuitively speaking, the action subsequence of sch, ex_A (ex_B) must have the same length, as the merge result cannot be longer than the oracle sch, whereas its equality to ex_A has to be established. □

The result holds analogously for projection and filtering w.r.t. the signature of B, yielding ex_B instead of ex_A.

Definition 10.3.9 (Schedule Property Composition)
Parallel composition on schedule properties is defined as follows:

$$\|_{sch} \quad : \quad (\alpha)\,\mathsf{sched\text{-}prop} \to (\alpha)\,\mathsf{sched\text{-}prop} \to (\alpha)\,\mathsf{sched\text{-}prop}$$

$$\begin{array}{l}
(sig_A, scheds_A)\ \|_{sch}\ (sig_B, scheds_B) \equiv \\
\quad \{\mathsf{sig\text{-}comp}\ sig_A\ sig_B, \\
\qquad\quad \{sch \mid \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `sch \in scheds_A\} \\
\qquad \cap\ \{sch \mid \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `sch \in scheds_B\} \\
\qquad \cap\ \{sch \mid \mathsf{Forall}\ (\lambda a.\ a \in (\mathsf{actions}\ sig_A \cup \mathsf{actions}\ sig_A))\ sch\}\}
\end{array}$$

Again we can rephrase the main compositionality result (Theorem 10.3.1) in a concise way.

Corollary 10.3.10 (Compositionality of Schedule Properties)
Schedule properties are compositional.

$$\mathsf{Scheds}\ (A \parallel B) = \mathsf{Scheds}(A) \parallel_{sch} \mathsf{Scheds}(B)$$

10.4 Compositionality for Traces

As in the previous section, we start with the main compositionality theorem and its proof in a top-down manner. The aim is to motivate the subsequent lemmas, which are then reported in a bottom-up fashion according to the dependency graph depicted in Fig. 10.4.

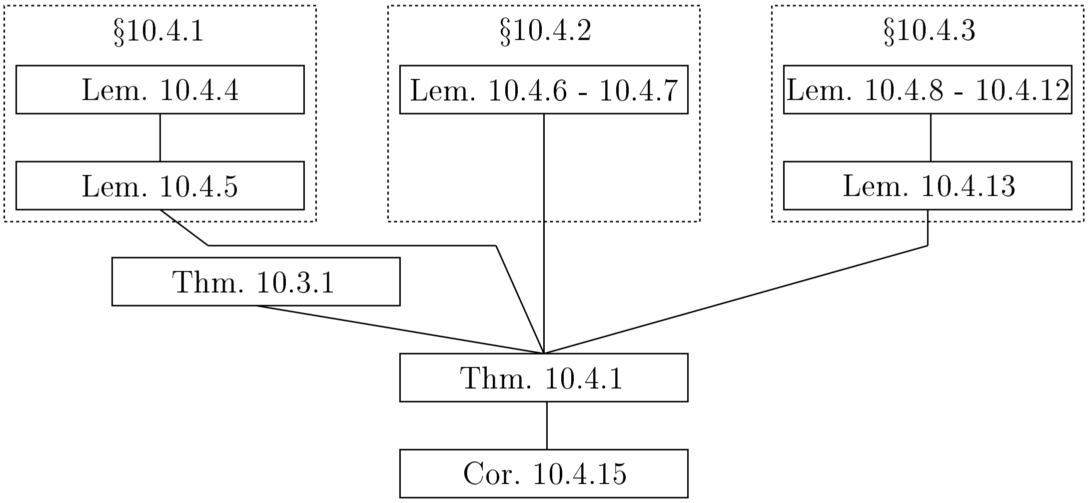

Figure 10.4: Dependency Graph of Lemmas proven in §10.4

The proof structure of the following theorem, which represents the formal counterpart of Thm. 2.4.9, part 2, is rather similar to that of Theorem 10.3.1. Furthermore, note the similarity of Fig. 10.2 and Fig. 10.5.

Theorem 10.4.1 (Compositionality for Traces)
Traces of $A \parallel B$ are related to those of A and B via the following theorem:

$$\begin{array}{ll}
\mathcal{A}_1: & \mathsf{compatible}\ A\ B \\
\mathcal{A}_2: & \mathsf{is\text{-}trans\text{-}of}(A) \wedge \mathsf{is\text{-}trans\text{-}of}(B) \\
\mathcal{A}_3: & \mathsf{is\text{-}sig\text{-}of}(A) \wedge \mathsf{is\text{-}sig\text{-}of}(B) \\
\hline
\mathcal{C}_1: & tr \in \mathsf{traces}\ (A \parallel B) = \\
& \quad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\ \text{`}tr \in \mathsf{traces}(A) \wedge \\
& \quad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(B))\ \text{`}tr \in \mathsf{traces}(B) \wedge \\
& \quad \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{ext}\ (A \parallel B))\ tr
\end{array}$$

$$
\begin{tikzcd}[column sep=large, row sep=large]
 & \text{Schedules} & \text{Traces} \\
A & sch_A \arrow[r, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ A)"] & tr_A \\
A \parallel B & sch \arrow[u, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)"] \arrow[d, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B)"'] \arrow[r, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ (A \parallel B))"] & tr \arrow[u, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)"'] \arrow[d, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B)"] \\
B & sch_B \arrow[r, "\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ B)"] & tr_B
\end{tikzcd}
$$

Figure 10.5: Compositionality for Traces

Proof.
See also Fig. 10.5 and Fig. 10.6, which illustrate the main proof ideas.

"⇒": The assumption $tr \in \mathsf{traces}\ (A \parallel B)$ guarantees the existence of sch with

$$\mathcal{A}_4: \quad sch \in \mathsf{schedules}\ (A \parallel B) \wedge tr = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ (A \parallel B))\ `sch$$

There are three conjuncts to prove. The third one is trivial, as a trace of $A \parallel B$ contains only external actions from the signature of $A \parallel B$. Formally, that is proved by the sequence lemma $\mathsf{Forall}\ P\ (\mathsf{Filter}\ P\ `s)$. Therefore, we restrict our intention to the first conjunct; the second one is proved analogously. It has to be shown that there is an sch_A with

$$\mathcal{C}_1: \quad sch_A \in \mathsf{schedules}(A)$$
$$\mathcal{C}_2: \quad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}(A))\ `sch_A = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\ `tr$$

We define sch_A to be $\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `sch$. Then, $\mathcal{C}_1$ is directly discharged by the compositionality result for schedules (Thm. 10.3.1) using $\mathcal{A}_1$. $\mathcal{C}_2$ turns into

$$\begin{aligned}\mathcal{C}_3: \quad & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}(A))\ `(\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\ `sch) \\ & = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\ `(\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ (A \parallel B))\ `sch)\end{aligned}$$

by rewriting with $tr = \mathsf{Filter}\,(\lambda a.\ a \in \mathsf{ext}\,(A \parallel B))$ from $\mathcal{A}_4$ and with the definition of sch_A. The resulting proposition $\mathcal{C}_3$ means that the diagram in Fig. 10.5 commutes.

This is essentially shown by properties about Filter and the compatibility of A and B. In detail, using the lemmas

$$\mathsf{Filter}\ P\ `(\mathsf{Filter}\ Q\ `s) = \mathsf{Filter}\ (\lambda a.\ P(a) \wedge Q(a))\ `s$$
$$a \in \mathsf{ext}\ (A \parallel B) = a \in \mathsf{ext}(A) \vee a \in \mathsf{ext}(B)$$

proposition $\mathcal{C}_3$ reduces to

$$\mathcal{C}_4\text{:}\quad a \in \mathsf{ext}(A) \wedge a \in \mathsf{act}(A) = a \in \mathsf{act}(A) \wedge (a \in \mathsf{ext}(A) \vee a \in \mathsf{ext}(B))$$

which is derived automatically from $\mathcal{A}_1$, the compatibility of A and B.

"$\Leftarrow$": The assumptions imply that there are schedules sch_A and sch_B with

$\mathcal{A}_4$: $sch_A \in \mathsf{schedules}(A) \wedge \mathsf{Filter}(\lambda a.\ a \in \mathsf{ext}(A))\ `sch_A = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\ `tr$
$\mathcal{A}_5$: $sch_B \in \mathsf{schedules}(B) \wedge \mathsf{Filter}(\lambda a.\ a \in \mathsf{ext}(B))\ `sch_B = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(B))\ `tr$
$\mathcal{A}_6$: $\mathsf{Forall}\ (\lambda a.\ a \in \mathsf{ext}\ (A \parallel B))\ tr$

It has to be shown that there is a schedule sch with

$\mathcal{C}_1$: $sch \in \mathsf{schedules}\ (A \parallel B)$
$\mathcal{C}_2$: $\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ (A \parallel B))\ sch = tr$

A major challenge is to define such a sch appropriately. Basically, sch_A and sch_B have to be merged in such a way, that the order of the external actions is given by tr, whereas the internal actions of sch_A and sch_B have to be interleaved in an appropriate way. At first sight, such a sch can be recursively defined quite in the spirit of mkex: the external action sequence serves as an oracle, which determines which internal actions of sch_A or sch_B have to be interleaved next. However, there is a severe complication for the case when sch_A or sch_B end with internal instead of external actions. The concrete problem will be explained later on. Fortunately, there is a remarkable solution to it: Lemma 10.4.5 will allow us to take two different sch'_A and sch'_B instead of sch_A and sch_B which necessarily end with an external action — a property expressed by the predicate Last-act-is-ext. Therefore, there are sch'_A and sch'_B with

$\mathcal{A}_7$: $sch'_A \in \mathsf{schedules}(A) \wedge \mathsf{Filter}(\lambda a.\ a \in \mathsf{ext}(A))\ `sch'_A = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\ `tr\ \wedge$
$\mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch'_A$
$\mathcal{A}_8$: $sch'_B \in \mathsf{schedules}(B) \wedge \mathsf{Filter}(\lambda a.\ a \in \mathsf{ext}(B))\ `sch'_B = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(B))\ `tr\ \wedge$
$\mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch'_A$

Using these sch'_A and sch'_B a merge function mksch can be defined recursively which is done in Def. 10.4.2 and will be explained in detail later on. We set:

$$sch := \mathsf{mksch}\ A\ B\ tr\ sch'_A\ sch'_B$$

To prove $\mathcal{C}_2$ structural induction is used together with weakening of the assumptions in order to fit the admissibility test (Lemma 10.4.6), analogous to the proofs for schedule composition. To prove $\mathcal{C}_1$ we use the compositionality result for schedules (Thm. 10.3.1) and get

$$\begin{aligned}
\mathcal{C}_3\!: &\quad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\ \text{`}sch \in \mathsf{schedules}(A)\\
\mathcal{C}_4\!: &\quad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(B))\ \text{`}sch \in \mathsf{schedules}(B)\\
\mathcal{C}_5\!: &\quad \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ (A \parallel B))\ sch
\end{aligned}$$

Here, $\mathcal{C}_5$ can be reduced to $\mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ (A \parallel B))\ tr$ by Lemma 10.4.7, which in turn is a consequence of $\mathcal{A}_6$ because of $a \in \mathsf{ext}(A) \Rightarrow a \in \mathsf{act}(A)$. Instead of $\mathcal{C}_3$ and $\mathcal{C}_4$ we prove

$$\begin{aligned}
\mathcal{C}_6\!: &\quad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(A))\ \text{`}sch = sch'_A\\
\mathcal{C}_7\!: &\quad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}(B))\ \text{`}sch = sch'_B
\end{aligned}$$

which suffices because of the first conjuncts of $\mathcal{A}_7$ and $\mathcal{A}_8$. Unfortunately, $\mathcal{C}_6$ and $\mathcal{C}_7$ cannot easily be proved by induction, as weakening of the assumptions or reformulations are not possible in order to fit the admissibility check. Therefore, the take lemma is employed instead (Lemma 10.4.13), using the proof infrastructure provided in §6.5. The proof of Lemma 10.4.13 represents — besides the definition of the merge function — the major challenge of the overall proof. □

This proof outline shows that the main challenges of the compositionality proof for traces are analogous to those of the proof for schedules. A complicated merge function has to be defined constructively, and admissibility problems due to sequence equalities in the premises have to be solved.

However, the task is more difficult this time, as the solutions provided in the last section do not apply here. First, it is true a merge function can be defined, but it does only operate properly, if the input sequences are modified, which requires a change of the overall proof, differing from that found in the literature [LT87]. This is discussed in §10.4.1.

Second, the workarounds to circumvent the admissibility problems proposed in §10.3.2, whose aim is to reformulate the proof goal in such a way that the syntactic admissibility test can be applied, does not always work here. The reason is, that the statements obey the form

$$\mathsf{Filter}\ h\ \text{`}y = f(x) \Rightarrow A\ x\ y$$

which does not allow us to reformulate them according to formula (10.1) developed in §10.3.2. Instead, we use the take lemma, which does not require admissibility, and take advantage of the proof infrastructure provided for its use in §6.5. Indeed, the necessary take lemma proof of this section was the initial motivation for building this infrastructure. Lemmas which can be treated by reformulations are discussed in §10.4.2, the remainder in §10.4.3.

10.4.1 A specific Merge Function

Basically, sch_A and sch_B have to be merged in such a way, that the order of the external actions is given by tr, whereas the internal actions of sch_A and sch_B have to be interleaved in an appropriate way. In particular, the common external action of sch_A and sch_B have to be synchronized according to the oracle tr.

First, the semi-formal definition is presented, as formulated in [LT87]. The syntax used for I/O automata and informal sequences stems from the introduction in §2.

> *Let $A = A_1 \parallel \cdots \parallel A_n$. Suppose $tr = a_1 a_2 \cdots$ is a trace of A and sch_i are schedules of A_i which have the trace $tr\lceil A_i$ for all i. Then we can write $sch_i = int_0^i\, ext_1^i\, int_1^i \cdots$ where int_j^i is a (possibly empty) sequence of internal actions of A_i, and ext_j^i is a_j if a_j is an external action of A_j, otherwise the empty sequence. Now the result of merging sch_i and tr is defined as $sch := \gamma_0 a_1 \gamma_1 a_2 \cdots$ where γ_i is an arbitrary interleaving of the actions appearing in int_j^i.*

Similar to the previous section, this definition is not constructive, as the sch_i are reformulated in a descriptive way. As before, it is by no means clear how to prove the existence of such a reformulation without a constructive merge function. However, this time it is not possible to come up with a recursive definition that matches the informal one. This is due to complications caused by internal actions possibly following the last external action in sch_A or sch_B. To explain that, let us first suppose, that sch_A and sch_B do not contain any internal action after the last external one. Then, the merge process can be formalized recursively as given in Definition 10.4.2. The definition of mksch is explained by the following example, which is illustrated in Figure 10.6.

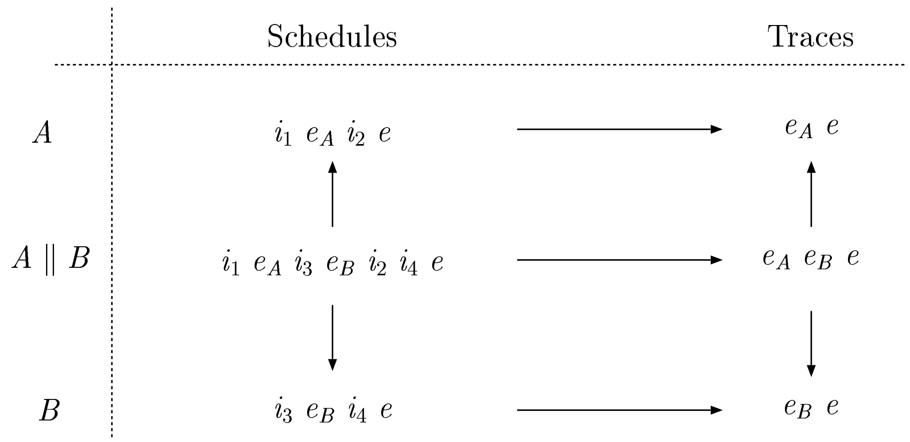

Figure 10.6: Compositionality for Traces – An Example

Let $tr = e_A\, e_B\, e$, $sch_A = i_1\, e_A\, i_2\, e$, and $sch_B = i_3\, e_B\, i_4\, e$, where e_A is an external action of A, e_B an external action of B, e a common external action, and the remaining actions represent internal ones.

The merge process is guided by tr: as the first element e_A is an action of A, the internal actions of sch_A in front of e_A are taken first, then e_A itself. Similarly for e_B, as it is an action of B, the internal actions of sch_B in front of e_B are taken, then e_B itself. Therefore, the result up to now equals $i_1\, e_A\, i_3\, e_B$. Now we have to synchronize on e, as it is an action of both A and B. This means that first all internal actions of sch_A in front of e are taken, then those of sch_B and then e itself. The final result is $sch := i_1\, e_A\, i_3\, e_B\, i_2\, i_4\, e$. Note that arbitrary interleaving of the internal actions would also be possible.

Definition 10.4.2 (Merge of Schedules)
Given two schedules sch_A and sch_B of A and B and a trace tr of $A \parallel B$ the function mksch merges them as just described informally.

$$\begin{aligned}\mathsf{mksch} \;::\; & (\alpha,\sigma)\,\mathsf{ioa} \rightarrow (\alpha,\tau)\,\mathsf{ioa} \rightarrow \alpha\,\mathsf{sequence}\\ & \rightarrow_c \alpha\,\mathsf{sequence} \rightarrow_c \alpha\,\mathsf{sequence} \rightarrow_c \alpha\,\mathsf{sequence} \rightarrow_c \alpha\,\mathsf{sequence}\end{aligned}$$

The following rewrite rules have been derived from the fixpoint definition:

$$\mathsf{mksch}\; A\; B\; `\bot\; `sch_A\; `sch_B = \bot$$

$$\mathsf{mksch}\; A\; B\; `\mathsf{nil}\; `sch_A\; `sch_B = \mathsf{nil}$$

$$\frac{a \in \mathsf{act}(A) \quad a \notin \mathsf{act}(B)}{\begin{aligned}&\mathsf{mksch}\; A\; B\; `(a\hat{}\,tr)\; `sch_A\; `sch_B =\\ &\quad (\mathsf{Takewhile}\;(\lambda a.\; a \in \mathsf{int}\; A)\; `sch_A)\\ &\quad\; \oplus\; (a\hat{}\,\mathsf{mksch}\; A\; B\; `tr\; `(\mathsf{TL}\; `(\mathsf{Dropwhile}\;(\lambda a.\; a \in \mathsf{int}\; A)\; `sch_A))\; `sch_B)\end{aligned}}$$

$$\frac{a \notin \mathsf{act}(A) \quad a \in \mathsf{act}(B)}{\begin{aligned}&\mathsf{mksch}\; A\; B\; `(a\hat{}\,tr)\; `sch_A\; `sch_B =\\ &\quad (\mathsf{Takewhile}\;(\lambda a.\; a \in \mathsf{int}\; B)\; `sch_B)\\ &\quad\; \oplus\; (a\hat{}\,\mathsf{mksch}\; A\; B\; `tr\; `sch_A\; `(\mathsf{TL}\; `(\mathsf{Dropwhile}\;(\lambda a.\; a \in \mathsf{int}\; B)\; `sch_B)))\end{aligned}}$$

$$\frac{a \in \mathsf{act}(A) \quad a \in \mathsf{act}(B)}{\begin{aligned}&\mathsf{mksch}\; A\; B\; `(a\hat{}\,tr)\; `sch_A\; `sch_B =\\ &\quad (\mathsf{Takewhile}\;(\lambda a.\; a \in \mathsf{int}\; A)\; `sch_A) \oplus\\ &\quad (\mathsf{Takewhile}\;(\lambda a.\; a \in \mathsf{int}\; B)\; `sch_B)\\ &\quad\; \oplus\; (a\hat{}\,\mathsf{mksch}\; A\; B\; `tr\; `(\mathsf{TL}\; `(\mathsf{Dropwhile}\;(\lambda a.\; a \in \mathsf{int}\; A)\; `sch_A))\\ &\qquad\qquad\qquad `(\mathsf{TL}\; `(\mathsf{Dropwhile}\;(\lambda a.\; a \in \mathsf{int}\; B)\; `sch_B)))\end{aligned}}$$

□

Note that, quite similar to the environment assumptions for mkex, mksch has to be used only under the assumptions

$$\mathsf{Filter}\;(\lambda x.\; x \in \mathsf{act}(A))\; `tr \sqsubseteq \mathsf{Filter}\;(\lambda x.\; x \in \mathsf{ext}(A))\; `sch_A$$
$$\mathsf{Filter}\;(\lambda x.\; x \in \mathsf{act}(B))\; `tr \sqsubseteq \mathsf{Filter}\;(\lambda x.\; x \in \mathsf{ext}(B))\; `sch_B$$

which guarantee that mksch does not expect elements from schedules which equal nil or $\perp$, and therefore behaves properly.

The problem with this continuous definition is that it cannot take care of internal actions appearing after the last external action in sch_A or sch_B, because the merge process terminates with the end of the oracle tr. At first sight, it seems that this could be compensated by a noncontinuous fairmerge function which merges both internal rests of sch_A and sch_B in a fair way. The fair merge result could then be appended to the result of mksch by a special noncontinuous concatenation function. Of course, this would complicate reasoning about the merge result significantly, as it does not match the inductive proof principles any longer and admissibility cannot be reduced to the continuity of the occurring functions.

But even worse, it does not produce the correct result, as it fails when one schedule contains infinitely many external actions whereas the other contains only finitely many and some internal actions afterwards.

To see this, consider the simple case where $sch_A = eee\cdots$, $sch_B = iii\cdots$, and $tr = eee\cdots$. This means that all actions e stem from A. As mksch follows tr it only looks at sch_A and does not consider sch_B at all. The problem is that there is no way to append sch_B to the result of mksch afterwards, as this result is already infinite, no matter whether a fairmerge function or a noncontinuous concatenation is used or not.

Therefore, the integration of the internal rests has to be incorporated already in the functionality of mksch. Unfortunately, this cannot be done in a continuous way, as we do not know, which action is the last external one, when looking along the schedules. Therefore a correct mksch allowing arbitrary schedule input could not take advantage of the powerful recursion mechanisms provided in HOLCF. It is by no means clear how to define mksch instead.

Therefore, the only way out is to circumvent schedules as input for mksch which contain internal actions after the last external one. Fortunately, this is possible as a deeper analysis of the proof of Theorem 10.4.1 shows that there is certain freedom in selecting the schedule inputs: sch_A, for example, has to be chosen in such a way that it is indeed a schedule of A and yields a given trace tr when filtering out internal actions. In the following we will show that if such a sch_A exists, there is always another sch'_A which satisfies in addition to these two requirements the restriction that it does not contain internal actions after the last external one. The idea is quite obvious: sch'_A is obtained from sch_A by deleting all internal actions after the last external one. Intuitively it is now clear that sch'_A is still an execution of A as internal "divergence" does not affect this property. Furthermore the trace of sch'_A will be tr as well, as sch_A and sch'_A differ only concerning internal actions.

The formal treatment of these facts starts with the definition of the desired property.

Definition 10.4.3 (External Last Action)
The predicate Last-act-is-ext identifies those schedules of an I/O automaton A which do not permit internal actions of A to follow the last external action, or, if no external actions

exist, do not consist of internal actions of A only.

$$\begin{array}{lcl}
\mathsf{Last\text{-}act\text{-}is\text{-}ext} & :: & (\alpha,\sigma)\,\mathsf{ioa} \rightarrow \alpha\,\mathsf{sequence} \rightarrow \mathsf{bool} \\
\mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch & \equiv & \mathsf{Chop}\ (\lambda x.\ x \in \mathsf{ext}(A))\ sch = sch
\end{array}$$

□

The question may arise why this definition is needed: instead of claiming for sch to satisfy Last-act-is-ext one could use $sch' := \mathsf{Chop}\ (\lambda x.\ x \in \mathsf{ext}(A))\ sch$ in place of sch. Then Last-act-is-ext would be satisfied implicitly for every occurrence of sch'. However, we prefer to encapsulate the property Last-act-is-ext by a predicate which can be used only when it is really needed.

The following result about Chop represents one of the main requirements for the next lemma.

Lemma 10.4.4
The execution fragment property is closed under the Chop operator.

$$\frac{\mathsf{is\text{-}exec\text{-}frag}\ A\ (s, ex)}{\mathsf{is\text{-}exec\text{-}frag}\ A\ (s, \mathsf{Chop}\ P\ ex)}$$

Proof.
The proof is by case distinction on the finiteness of ex:

- **Case** $\mathsf{Finite}(ex)$**:**
  We use Lemma 6.6.4(1) to conclude that there is a y such that $ex = \mathsf{Chop}\ P\ ex \oplus y$. Therefore, we may assume: $\mathsf{is\text{-}exec\text{-}frag}\ A\ (s, \mathsf{Chop}\ P\ ex \oplus y)$. The fact that execution fragments are prefix closed w.r.t. $\ll$ yields the result (Lemma 8.3.10(2)).

- **Case** $\neg\mathsf{Finite}(ex)$**:**
  We use Lemma 6.6.4(2) to conclude that $\mathsf{Chop}\ P\ ex$ is a prefix of ex. The fact, that execution fragments are prefix closed w.r.t. $\sqsubseteq$ yields the result (Lemma 8.3.10(1)).

□

Therefore, we can present the main result, which allows us to choose always such a schedule, which fulfills the Last-act-is-ext predicate and therefore guarantees a proper behaviour of mksch.

Lemma 10.4.5
For every schedule sch_1 of an I/O automaton A with given trace tr there is another schedule

sch_2 of A with the same trace tr but fulfilling the $\mathsf{Last\text{-}act\text{-}is\text{-}ext}$ predicate.

$$\frac{\begin{array}{ll}\mathcal{A}_1: & sch_1 \in \mathsf{schedules}(A) \\ \mathcal{A}_2: & tr = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ \text{`}sch_1\end{array}}{\begin{array}{ll}\mathcal{C}_1: & \exists sch_2 . sch_2 \in \mathsf{schedules}(A) \wedge tr = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ \text{`}sch_2 \wedge \\ & \quad \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch_2\end{array}}$$

Proof.
Interestingly, the lemma is a consequence of the following basic lemmas about Chop, which have all been proved using the take-lemma (Lemma 6.6.3):

$$\begin{array}{lcl}\mathsf{Chop}\ P\ (\mathsf{Chop}\ P\ x) & = & \mathsf{Chop}\ P\ x \\ \mathsf{Filter}\ P\ \text{`}x & = & \mathsf{Filter}\ P\ \text{`}(\mathsf{Chop}\ P\ x) \\ \mathsf{Map}\, f\ \text{`}(\mathsf{Chop}\ (P \circ f)\ \text{`}x) & = & \mathsf{Chop}\ P\ (\mathsf{Map}\, f\ \text{`}x)\end{array}$$

In detail, the proof proceeds as follows. Unfolding the definition of schedules, $\mathcal{A}_1$ and $\mathcal{A}_2$ guarantee the existence of an $exec_1$ with

$$\begin{array}{ll}\mathcal{A}_3: & exec_1 \in \mathsf{executions}(A) \\ \mathcal{A}_4: & sch_1 = \mathsf{Filter\text{-}act}\ \text{`}(\mathsf{snd}\ exec_1) \\ \mathcal{A}_5: & tr = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ \text{`}sch_1\end{array}$$

Accordingly, proposition $\mathcal{C}_1$ reduces to

$$\begin{array}{lll}\mathcal{C}_2: & \exists sch_2\ exec_2. & exec_2 \in \mathsf{executions}(A) \wedge sch_2 = \mathsf{Filter\text{-}act}\ \text{`}(\mathsf{snd}\ exec_2) \wedge \\ & & tr = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ \text{`}sch_2 \wedge \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch_2\end{array}$$

by the definition of schedules and some predicate-logical reasoning. Intuitively, $exec_2$ may be gained from $exec_1$ by eliminating all action/state pairs with internal actions occurring after the last external action. Writing (s, ex) for $exec_1$ this is done by the Chop operator:

$$\begin{array}{lcl}exec_2 & := & (s, \mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}\ A)\ ex) \\ sch_2 & := & \mathsf{Filter\text{-}act}\ \text{`}(\mathsf{snd}\ exec_2)\end{array}$$

By instantiating these definitions and rewriting with $\mathcal{A}_4$ and $\mathcal{A}_5$ $\mathcal{C}_2$ divides into three subgoals:

$$\begin{array}{ll}\mathcal{C}_3: & (s, \mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}(A))\ ex) \in \mathsf{executions}(A) \\ \mathcal{C}_4: & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ \text{`}(\mathsf{Filter\text{-}act}\ \text{`}ex) \\ & = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ \text{`}(\mathsf{Filter\text{-}act}\ \text{`}(\mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}(A))\ ex) \\ \mathcal{C}_5: & \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ (\mathsf{Filter\text{-}act}\ \text{`}(\mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}(A))\ ex))\end{array}$$

Note that the second conjunct of $\mathcal{C}_2$ disappeared, because it has become trivial due to the definition of sch_2. As $\mathcal{A}_3$ assures $s \in \mathsf{starts\text{-}of}(A)$, $\mathcal{C}_3$ together with $\mathcal{A}_1$ turn into:

$$\mathcal{C}_6\text{:}\quad \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, ex) \Rightarrow \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, \mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}\ A)\ ex)$$

which is a consequence of Lemma 10.4.4.

Using the fact, that Chop commutes with Map (Lema 6.6.3), and unfolding the definition of Last-act-is-ext, $\mathcal{C}_4$ and $\mathcal{C}_5$ turn into:

$$\begin{aligned}
\mathcal{C}_7\text{:}\quad & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ \text{`}(\mathsf{Filter\text{-}act}\ \text{`}ex) \\
& = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ \text{`}(\mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}\ A)\ (\mathsf{Filter\text{-}act}\ \text{`}ex)) \\
\mathcal{C}_8\text{:}\quad & \mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}\ A)\ (\mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}\ A)\ (\mathsf{Filter\text{-}act}\ \text{`}ex) \\
& = \mathsf{Chop}\ (\lambda a.\ (\mathsf{fst}\ a) \in \mathsf{ext}\ A)\ (\mathsf{Filter\text{-}act}\ \text{`}ex)
\end{aligned}$$

This transformation reduced the remaining propositions to some standard properties about the Chop operator: $\mathcal{C}_7$ is an implication of $\mathsf{Filter}\ P\ \text{`}x = \mathsf{Filter}\ P\ \text{`}(\mathsf{Chop}\ P\ x)$ (Lemma 6.6.3), and $\mathcal{C}_8$ expresses that Chop is idempotent (Lemma 6.6.3). □

10.4.2 Lemmas using Structural Induction

The following lemmas needed for Theorem 10.4.1 can be proved using structural induction. They do not require schedule inputs that satisfy the Last-act-is-ext predicate.

Lemma 10.4.6
Given a schedule which results from merging two schedules and a trace tr of A, filtering it onto the external actions of A yields tr again.

$$\frac{\mathcal{A}_1\text{: }\mathsf{compatible}\ A\ B \quad \mathcal{A}_2\text{: }\mathsf{is\text{-}sig\text{-}of}(A) \wedge \mathsf{is\text{-}sig\text{-}of}(B)}{\begin{aligned}
\mathcal{C}_1\text{: } \forall sch_A\ sch_B. & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ tr\ \wedge \\
& \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ A)\ sch_A \wedge \\
& \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ B)\ sch_B \wedge \\
& \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}tr \sqsubseteq \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ A)\ \text{`}sch_A \wedge \\
& \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}tr \sqsubseteq \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ \text{`}sch_B \wedge \\
& \Rightarrow \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ \text{`}(\mathsf{mksch}\ A\ B\ \text{`}tr\ \text{`}sch_A\ \text{`}sch_B) = tr
\end{aligned}}$$

Proof.
The proof is by **structural induction** on tr. The **admissibility** condition is discharged by the admissibility test, as the assumptions have been weakened according to the explanations above. The **base cases** $tr = \perp$, $tr = \mathsf{nil}$ are trivial.

Inductive step $tr = a\hat{}tr'$: Take $\mathcal{C}_1$ as induction hypothesis. Therefore, we get a new goal which has to be proved under four further assumptions:

$$\begin{array}{ll}
\mathcal{A}_3: & (a \in \mathsf{ext}(A) \vee a \in \mathsf{ext}(B)) \wedge \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ tr' \\
\mathcal{A}_4: & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(A))\ sch_A \wedge \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(B))\ sch_B \\
\mathcal{A}_5: & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(A))\ `(a\hat{}tr') \sqsubseteq \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ `sch_A \\
\mathcal{A}_6: & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(B))\ `(a\hat{}tr') \sqsubseteq \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(B))\ `sch_B \\
\mathcal{C}_2: & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ `(\mathsf{mksch}\ A\ B\ `(a\hat{}tr')\ `sch_A\ `sch_B) = a\hat{}tr'
\end{array}$$

We distinguish between four cases according to $a \in \mathsf{act}(A)$ and $a \in \mathsf{act}(B)$. The case $a \notin \mathsf{act}(A) \wedge a \notin \mathsf{act}(B)$ is ruled out because of $\mathcal{A}_3$ and $a \in \mathsf{ext}(A) \Rightarrow a \in \mathsf{act}(A)$. The other three cases are rather similar, we restrict our attention to the simplest case to make the proof idea clear:

Case $a \in \mathsf{act}(A) \wedge a \notin \mathsf{act}(B)$**:** Lemma 6.5.2 is used to derive

$$\begin{array}{ll}
\mathcal{A}_7: & sch_A = \mathsf{Takewhile}\ (\lambda x.\ x \notin \mathsf{ext}(A))\ `sch_A \oplus a\hat{}\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \notin \mathsf{ext}(A))\ `sch_A) \\
& \wedge\ \mathsf{Finite}\ (\mathsf{Takewhile}\ (\lambda x.\ x \notin \mathsf{ext}(A))\ `sch_A \wedge a \in \mathsf{ext}(A)
\end{array}$$

from $\mathcal{A}_5$. Because of $\mathcal{A}_2$ it is possible to rewrite the resulting assumption with

$$\mathsf{is\text{-}sig}\ S \Rightarrow (a \notin \mathsf{externals}\ S) = (x \in \mathsf{internals}\ S \vee x \notin \mathsf{actions}\ S)$$

Subsequently, the assumption is rewritten with

$$\mathsf{Forall}\ P\ s \Rightarrow \mathsf{Takewhile}\ (\lambda x.\ Q(x) \vee (\neg P(x)))\ `s = \mathsf{Takewhile}\ Q\ `s$$

and the analogous lemma for Dropwhile, which is possible because of $\mathcal{A}_4$. Therefore, $\mathcal{A}_7$ turns into

$$\begin{array}{ll}
\mathcal{A}_8: & sch_A = \mathsf{Takewhile}\ (\lambda x.\ x \in \mathsf{int}(A))\ `sch_A \oplus a\hat{}\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}(A))\ `sch_A) \\
& \wedge\ \mathsf{Finite}\ (\mathsf{Takewhile}\ (\lambda x.\ x \in \mathsf{int}(A))\ `sch_A \wedge a \in \mathsf{ext}(A)
\end{array}$$

Note that $\mathcal{A}_4$ is needed only here to establish that non-external actions of the schedules are internal actions, as expected. Now, the equation for sch_A in $\mathcal{A}_8$ is substituted in $\mathcal{A}_5$. The result is further simplified by the sequence lemma

$$\mathsf{Finite}\ (\mathsf{Takewhile}\ Q\ `s) \wedge (\forall x.\ Q(x) \Rightarrow P(x)) \Rightarrow \mathsf{Filter}\ P\ `(\mathsf{Takewhile}\ Q\ `s) = \mathsf{nil}$$

which is possible because of the last two conjuncts of $\mathcal{A}_8$, the lemma

$$\mathsf{is\text{-}sig}(S) \Rightarrow (a \in \mathsf{internals}(S) \Rightarrow a \notin \mathsf{externals}(S)$$

and $\mathcal{A}_2$. We get, together with the equations for Filter and the case information $a \in \mathsf{act}(A)$, the following:

$$\begin{array}{ll}
\mathcal{A}_9: & a\hat{}\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(A))\ `tr' \sqsubseteq \\
& a\hat{}\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ `(\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}(A))\ `sch_A))
\end{array}$$

The a in front can be eliminated as $\sqsubseteq$ on sequences expresses the prefix ordering. Similar to the transformation of $\mathcal{A}_5$ it is possible to rewrite $\mathcal{A}_4$ with the equation for sch_A in $\mathcal{A}_8$. Together with the fact that Forall distributes over $\oplus$ we derive from $\mathcal{A}_4$

$$\mathcal{A}_{10}\text{:}\quad \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(A))\ (\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}(A))\ `sch_A))$$

Using the case information $a \notin \mathsf{act}(B)$ and the Filter rewrite rules $\mathcal{A}_6$ simplifies to

$$\mathcal{A}_{11}\text{:}\quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(B))\ `tr' \sqsubseteq \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(B))\ `sch_B$$

Until now, we merely performed forward reasoning on the assumptions in order to make them applicable for the induction hypothesis later on. Now, we turn to the proposition to prove, $\mathcal{C}_2$, and rewrite it with the mksch rewrite rules:

$$\begin{aligned}\mathcal{C}_3\text{:}\quad & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\\ & `(\mathsf{Takewhile}\ (\lambda a.\ a \in \mathsf{int}(A))\ `sch_A)\\ & \ \oplus (a\hat{\ }\mathsf{mksch}\ A\ B\ `tr'\ `(\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda a.\ a \in \mathsf{int}(A))\ `sch_A))\ `sch_B) = a\hat{\ }tr'\end{aligned}$$

Intuitively, the internal actions from sch_A interleaved by mksch will again be filtered out when generating the external trace. Formally, $\mathcal{C}_3$ is simplified by the same sequence lemma as above about Filter and Takewhile, which is possible because of the last two conjuncts of $\mathcal{A}_8$, the facts

$$\begin{aligned}&\mathsf{is\text{-}sig}(S) \Rightarrow a \in \mathsf{internals}(S) \Rightarrow a \notin \mathsf{externals}(S))\\ &\mathsf{compatible}\ A\ B \Rightarrow x \in \mathsf{int}(A) \Rightarrow x \notin \mathsf{ext}(B)\\ &\mathsf{Filter}\ P\ `(s\ \oplus\ t) = \mathsf{Filter}\ P\ `s\ \oplus\ \mathsf{Filter}\ P\ `t\\ &a \in \mathsf{ext}\ (A \parallel B) = a \in \mathsf{ext}(A) \vee a \in \mathsf{ext}(B)\end{aligned}$$

and the assumptions $\mathcal{A}_2$ and $\mathcal{A}_1$. Therefore, we get:

$$\begin{aligned}\mathcal{C}_4\text{:}\quad & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\\ & `(a\hat{\ }\mathsf{mksch}\ A\ B\ `tr'\ `(\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda a.\ a \in \mathsf{int}(A))\ `sch_A))\ `sch_B) = a\hat{\ }tr'\end{aligned}$$

As $\mathcal{A}_8$ guarantees $a \in \mathsf{ext}(A)$, $\mathcal{C}_4$ further simplifies to

$$\begin{aligned}\mathcal{C}_5\text{:}\quad & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\\ & `\mathsf{mksch}\ A\ B\ `tr'\ `(\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda a.\ a \in \mathsf{int}(A))\ `sch_A))\ `sch_B = tr'\end{aligned}$$

which can be discharged using the induction hypothesis with the instantiations $sch_A :=$ $\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda a.\ a \in \mathsf{int}\ A)\ `sch_A)$ and $sch_B := sch_B$ using $\mathcal{A}_3$, the second conjunct of $\mathcal{A}_4$, $\mathcal{A}_9$, $\mathcal{A}_{10}$, and $\mathcal{A}_{11}$. □

Lemma 10.4.7
A schedule which results from merging two schedules of A and B and an oracle tr contains only actions from the signature of $A \parallel B$ provided tr does the same.

$$\begin{aligned}\mathcal{C}_1\text{:}\quad & \forall sch_A\ sch_B.\mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ tr\\ & \qquad\qquad \Rightarrow \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ (\mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `sch_B)\end{aligned}$$

Proof.
The proof is by **structural induction** on tr. The **admissibility** condition is discharged by the admissibility test. The **base cases** $tr = \bot$, $tr = \mathsf{nil}$ are trivial.

Inductive step $tr = a\hat{\ }tr'$: Take $\mathcal{C}_1$ it as induction hypothesis. Therefore, we get a new goal and a first assumption.

$$\begin{array}{ll} \mathcal{A}_1\colon & (a \in \mathsf{ext}(A) \vee a \in \mathsf{ext}(B)) \wedge \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ tr' \\ \mathcal{C}_2\colon & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ (\mathsf{mksch}\ A\ B\ \text{‘}(a\hat{\ }tr')\ \text{‘}sch_A\ \text{‘}sch_B) \end{array}$$

We distinguish between four cases according to $a \in \mathsf{act}(A)$ and $a \in \mathsf{act}(B)$. The case $a \notin \mathsf{act}(A) \wedge a \notin \mathsf{act}(B)$ is ruled out because of $\mathcal{A}_1$ and $a \in \mathsf{ext}(A) \Rightarrow a \in \mathsf{act}(A)$. The other three cases are similar, as in the previous proof we restrict us to the following case:

Case $a \in \mathsf{act}(A) \wedge a \notin \mathsf{act}(B)$**:** Rewriting $\mathcal{C}_2$ with the mksch rewrite rules yields:

$$\begin{array}{ll} \mathcal{C}_3\colon & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B)) \\ & \quad (\mathsf{Takewhile}\ (\lambda a.\ a \in \mathsf{int}\ A)\ \text{‘}sch_A \\ & \quad\ \oplus (a\hat{\ }\mathsf{mksch}\ A\ B\ \text{‘}tr'\ \text{‘}(\mathsf{TL}\ \text{‘}(\mathsf{Dropwhile}\ (\lambda a.\ a \in \mathsf{int}\ A)\ \text{‘}sch_A))\ \text{‘}sch_B)) \end{array}$$

Now, we apply the sequence lemma $\mathsf{Forall}\, P\ s \wedge \mathsf{Forall}\, P\ t \Rightarrow \mathsf{Forall}\, P\ (s \oplus t)$ to $\mathcal{C}_3$ and get:

$$\begin{array}{ll} \mathcal{C}_4\colon & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ (\mathsf{Takewhile}\ (\lambda a.\ a \in \mathsf{int}\ A)\ \text{‘}sch_A) \\ \mathcal{C}_5\colon & a \in \mathsf{act}\ (A \parallel B) \\ \mathcal{C}_6\colon & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B)) \\ & \quad (\mathsf{mksch}\ A\ B\ \text{‘}tr'\ \text{‘}(\mathsf{TL}\ \text{‘}(\mathsf{Dropwhile}\ (\lambda a.\ a \in \mathsf{int}\ A)\ \text{‘}sch_A))\ \text{‘}sch_B) \end{array}$$

Note that the sequence lemma above holds for the other direction as well, but only under the additional assumption $\mathsf{Finite}(s)$. It is the tricky point of the proof to prevent us from this finiteness assumption, as the four additional assumptions of Lemma 10.4.6 would have been necessary to satisfy it, whereas this lemma needs only one assumption.

$\mathcal{C}_4$ can be discharged using $a \in \mathsf{int}(A) \Rightarrow a \in \mathsf{act}(A)$ and

$$(\forall a.\ P(a) \Rightarrow Q(a) \Rightarrow \mathsf{Forall}\ Q\ (\mathsf{Filter}\ P\ \text{‘}s)$$

$\mathcal{C}_5$ follows from $\mathcal{A}_1$, and $\mathcal{C}_6$ can be discharged by the induction hypothesis using $sch_A :=$ $\mathsf{TL}\ \text{‘}(\mathsf{Dropwhile}\ (\lambda a.\ a \in \mathsf{int}\ A)\ \text{‘}sch_A)$, $sch_B := sch_B$, and $\mathcal{A}_1$. □

10.4.3 Lemmas using the Take Lemma

The aim of this subsection is to prove Lemma 10.4.13. As mentioned earlier, it cannot be proved by structural induction, as reformulations according to formula (10.1) in §10.3.2 in order to be amenable for the admissibility check are not possible. The take lemma is used instead. To apply it to Lemma 10.4.13, a number of auxiliary lemmas are needed, which are developed in a bottom up manner in the sequel (cf. Fig. 10.4).

Lemma 10.4.8
Schedules of A that satisfy Last-act-is-ext A and do not contain external actions of A, equal nil if they are finite, otherwise they equal $\perp$.

$$\frac{\mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch \quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ `sch = \mathsf{nil}}{sch = \mathsf{nil}}$$

$$\frac{\mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch \quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ `sch = \perp}{sch = \perp}$$

Proof.
The following sequence lemmas

$$\begin{array}{lll} \mathsf{Filter}\ P\ `s = \mathsf{nil} & \Rightarrow & (\mathsf{Forall}\ (\lambda a.\ \neg P(x))\ s \wedge \mathsf{Finite}\ s) \\ \mathsf{Filter}\ P\ `s = \perp & \Rightarrow & (\mathsf{Forall}\ (\lambda a.\ \neg P(x))\ s \wedge \neg\mathsf{Finite}\ s) \end{array}$$

are used for forward reasoning on the second conjunct of the assumption. Then, the characterizing equations for Chop (Lemma 6.6.2) yield the result immediately after unfolding he definition of Last-act-is-ext. □

Lemma 10.4.9
A schedule which results from merging two schedules of A and B and an oracle tr contains only actions from the signature of A and not of B, provided tr does the same.

$$\frac{\mathsf{compatible}\ A\ B}{\begin{array}{l}\forall sch_A\ sch_B. \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ A \wedge x \notin \mathsf{act}\ B)\ tr \\ \qquad \Rightarrow \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ A \wedge x \notin \mathsf{act}\ B)\ (\mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `sch_B)\end{array}}$$

Proof.
The proof is similar to that of Lemma 10.4.7. Therefore, we merely comment on the differences. First, there is only one case, namely $a \in \mathsf{act}(A) \wedge a \notin \mathsf{act}(B)$. Second, to apply the lemma $(\forall a.\ (P(a) \Rightarrow Q(a)) \Rightarrow \mathsf{Forall}\ Q\ (\mathsf{Filter}\ P\ `s)$ the fact compatible $A\ B \Rightarrow x \in \mathsf{int}(A) \Rightarrow x \notin \mathsf{ext}(B)$ is additionally needed, which makes the compatibility requirement necessary. □

Lemma 10.4.10
Every schedule which results from merging two schedules sch_A and sch_B and a trace tr is finite, provided tr is finite.

$$\frac{\mathsf{Finite}(tr) \quad \mathsf{compatible}\ A\ B \quad \mathsf{is\text{-}sig\text{-}of}(A) \wedge \mathsf{is\text{-}sig\text{-}of}(B)}{\begin{array}{l}\forall sch_A\ sch_B.\mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ tr\ \wedge \\ \qquad\qquad \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ A)\ sch_A \wedge \\ \qquad\qquad \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ B)\ sch_B \wedge \\ \qquad\qquad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ A)\ `sch_A = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `tr\ \wedge \\ \qquad\qquad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ `sch_B = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ `tr \\ \qquad\qquad \Rightarrow \mathsf{Finite}\ (\mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `sch_B)\end{array}}$$

Proof.
The proof is by finite structural induction on tr and follows along the lines of the proof of Lemma 10.4.6. Therefore, merely the characteristics will be outlined. Some notes on the necessity of the assumptions: the equations

$$\begin{array}{rcl}\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(A))\ `sch_A & = & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(A))\ `tr \\ \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}(B))\ `sch_B & = & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(B))\ `tr\end{array}$$

are needed, as a finite oracle does not suffice to yield a finite merge result. Rather the internal parts that are interleaved with this oracle have to be finite as well. This holds especially for the last internal actions after a possible last external action of sch_A or sch_B. Therefore, we really need the equality in these assumptions, weakening to a prefix relation would not suffice. Note that this does not lead to admissibility problems here, as finite structural induction is used.

The key lemma is $\mathsf{Finite}(s \oplus t) = (\mathsf{Finite}(s) \wedge \mathsf{Finite}(t))$ in contrast to Lemma 10.4.6, where the distribution of Filter over $\oplus$ plays this rôle. □

The following lemma represents the inverse direction of the preceding result.

Lemma 10.4.11
Given a finite schedule which results from merging two schedules sch_A and sch_B and a trace tr, tr is finite.

$$\frac{\mathsf{Finite}\ (\mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `sch_B)}{\mathsf{Finite}(tr)}$$

Proof.
The proof is by finite structural induction. However, as induction cannot be done on the term $\mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `sch_B$, the proposition has to be strengthened before. We obtain

the following lemma

$$\begin{array}{l} \mathsf{Finite}\ y \\ \Rightarrow \forall z\ tr.\mathsf{Forall}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ tr \wedge \\ \qquad y = z \oplus (\mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `sch_B) \\ \qquad \Rightarrow \mathsf{Finite}\ tr \end{array}$$

which can be proved by finite structural induction on y. The proof is very tedious because of many case distinctions, but does not give further insight. □

The following lemma represents the basis for the take lemma application in Lemma 10.4.13. It allows to "push" certain elements through mksch which do not produce any output. Therefore, the first element which is responsible for the output of mksch can be identified.

Lemma 10.4.12
Under certain restrictions mksch distributes over $\oplus$ in the oracle argument as follows:

$$\frac{\mathsf{Finite}(bs) \quad \mathcal{A}_1\text{: compatible } A\ B \quad \mathcal{A}_2\text{: is-sig-of}(A) \wedge \text{is-sig-of}(B)}{\begin{array}{l} \forall sch_B.\mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ B)\ sch_B \wedge \\ \qquad \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ B \wedge x \notin \mathsf{act}\ A)\ as \wedge \\ \qquad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ B)\ `sch_B = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B)\ `(bs \oplus tr) \\ \qquad \Rightarrow \exists x_1\ x_2.\mathsf{mksch}\ A\ B\ `(bs \oplus tr)\ `sch_A\ `sch_B = x_1 \oplus \mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `x_2 \wedge \\ \qquad\qquad \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ B \wedge a \notin \mathsf{act}\ A)\ x_1 \wedge \\ \qquad\qquad \mathsf{Finite}\ x_1 \wedge sch_B = (x_1 \oplus x_2) \wedge \\ \qquad\qquad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ B)\ `x_1 = bs \end{array}}$$

Proof.
The proof is by **finite structural induction** on as:

Base case $as = \mathsf{nil}$**:** Trivial with $x_1 := \mathsf{nil}$ and $x_2 := sch_B$.

Inductive step $bs = b\hat{\ }ss$: Take $\mathcal{C}_1$ as induction hypothesis. Therefore, we may assume:

$$\begin{array}{ll} \mathcal{A}_3\text{:} & \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ B)\ sch_B \\ \mathcal{A}_4\text{:} & b \in \mathsf{act}(B) \wedge a \notin \mathsf{act}(A) \wedge \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ B \wedge x \notin \mathsf{act}\ A)\ ss \\ \mathcal{A}_5\text{:} & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ B)\ `sch_B = b\hat{\ }\mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B)\ `(ss \oplus tr) \end{array}$$

We have to show:

$$\begin{array}{ll} \mathcal{C}_2\text{:} & \exists x_1'\ x_2'.\mathsf{mksch}\ A\ B\ `(b\hat{\ }ss \oplus tr)\ `sch_A\ `sch_B = x_1' \oplus \mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `x_2' \wedge \\ & \qquad \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ B \wedge a \notin \mathsf{act}\ A)\ x_1' \wedge \\ & \qquad \mathsf{Finite}\ x_1' \wedge sch_B = (x_1' \oplus x_2') \wedge \\ & \qquad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ B)\ `x_1' = (b\hat{\ }ss) \end{array}$$

Lemma 6.5.2 is used to derive

$$\begin{aligned}\mathcal{A}_6\colon\ & sch_B = \mathsf{Takewhile}\,(\lambda x.\ x \notin \mathsf{ext}\ B)\ \text{`}sch_B\ \oplus\ b\hat{}\,\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\,(\lambda x.\ x \notin \mathsf{ext}\ B)\ \text{`}sch_B)\\ & \wedge \mathsf{Finite}\,(\mathsf{Takewhile}\,(\lambda x.\ x \notin \mathsf{ext}\ B)\ \text{`}sch_B \wedge b \in \mathsf{ext}(B)\end{aligned}$$

from $\mathcal{A}_5$. Analogous to the proof of Lemma 10.4.6 this can be simplified to

$$\begin{aligned}\mathcal{A}_7\colon\ & sch_B = \mathsf{Takewhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B\ \oplus\ b\hat{}\,\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)\\ & \wedge \mathsf{Finite}\,(\mathsf{Takewhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B \wedge b \in \mathsf{ext}(B)\end{aligned}$$

because of $\mathcal{A}_2$ and $\mathcal{A}_3$. Now, the equation for sch_B in $\mathcal{A}_7$ is substituted in $\mathcal{A}_5$. The result is further simplified, again according to the proof of Lemma 10.4.6, to

$$\begin{aligned}\mathcal{A}_8\colon\ & \mathsf{Filter}\,(\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}(ss\ \oplus\ tr) =\\ & \mathsf{Filter}\,(\lambda x.\ x \in \mathsf{ext}\ B)\ \text{`}(\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B))\end{aligned}$$

using $\mathcal{A}_7$ and $\mathcal{A}_2$. Similarly, by substituting the equation for sch_A in $\mathcal{A}_3$ and simplification we derive

$$\mathcal{A}_9\colon\ \mathsf{Forall}\,(\lambda a.\ a \in \mathsf{act}\ B)\ (\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B))$$

from $\mathcal{A}_3$. Therefore, the induction hypothesis can be applied with the instantiation $sch_B :=$ $\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)$, which guarantees the existence of x_1 and x_2, such that

$$\begin{aligned}\mathcal{A}_{10}\colon\ & \mathsf{mksch}\ A\ B\ \text{`}(ss\ \oplus\ tr)\ \text{`}sch_A\ \text{`}(\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B))\\ & = x_1\ \oplus\ \mathsf{mksch}\ A\ B\ \text{`}tr\ \text{`}sch_A\ \text{`}x_2\\ \mathcal{A}_{11}\colon\ & \mathsf{Forall}\,(\lambda a.\ a \in \mathsf{act}\ B \wedge a \notin \mathsf{act}\ A)\ x_1 \wedge \mathsf{Finite}\ x_1\\ \mathcal{A}_{12}\colon\ & (\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)) = (x_1\ \oplus\ x_2)\\ \mathcal{A}_{13}\colon\ & \mathsf{Filter}\,(\lambda a.\ a \in \mathsf{ext}\ B)\ \text{`}x_1 = ss\end{aligned}$$

Now we have derived sufficient information from the assumptions to turn to $\mathcal{C}_2$. The existence of x_1 and x_2 is used to instantiate x_1' and x_2' as follows:

$$x_1' := (\mathsf{Takewhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)\ \oplus\ b\hat{}\,x_1 \quad \text{and} \quad x_2' := x_2$$

Therefore, $\mathcal{C}_2$ reduces to

$$\begin{aligned}\mathcal{C}_3\colon\ & \mathsf{mksch}\ A\ B\ \text{`}(b\hat{}\,ss\ \oplus\ tr)\ \text{`}sch_A\ \text{`}sch_B\\ & = ((\mathsf{Takewhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)\ \oplus\ b\hat{}\,x_1)\ \oplus\ \mathsf{mksch}\ A\ B\ \text{`}tr\ \text{`}sch_A\ \text{`}x_2\\ \mathcal{C}_4\colon\ & \mathsf{Forall}\,(\lambda a.\ a \in \mathsf{act}\ B \wedge a \notin \mathsf{act}\ A)\ ((\mathsf{Takewhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)\ \oplus\ b\hat{}\,x_1)\\ \mathcal{C}_5\colon\ & \mathsf{Finite}\ ((\mathsf{Takewhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)\ \oplus\ b\hat{}\,x_1)\\ \mathcal{C}_6\colon\ & sch_B = (((\mathsf{Takewhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)\ \oplus\ b\hat{}\,x_1)\ \oplus\ x_2)\\ \mathcal{C}_7\colon\ & \mathsf{Filter}\,(\lambda a.\ a \in \mathsf{ext}\ B)\ \text{`}((\mathsf{Takewhile}\,(\lambda x.\ x \in \mathsf{int}\ B)\ \text{`}sch_B)\ \oplus\ b\hat{}\,x_1) = ss\end{aligned}$$

$\mathcal{C}_3$ follows from the rewrite rules for mksch and $\mathcal{A}_{10}$. As Forall distributes over $\oplus$, $\mathcal{C}_4$ follows from $\mathcal{A}_4$, $\mathcal{A}_{11}$, and the application of the sequence lemma

$$(\forall a.\ Q(a) \Rightarrow P(a)) \quad \Rightarrow \quad \mathsf{Forall}\ P\ (\mathsf{Takewhile}\ Q\ `s)$$

which in turn requires basic lemmata about the action signatures, in particular compatibility, i.e $\mathcal{A}_1$. Similarly, as Finite distributes over $\oplus$, $\mathcal{C}_5$ follows from $\mathcal{A}_7$ and $\mathcal{A}_{11}$. $\mathcal{C}_6$ follows from the equation for sch_A in $\mathcal{A}_7$, $\mathcal{A}_{12}$, and associativity of $\oplus$. Finally, $\mathcal{C}_7$ follows from $\mathcal{A}_{13}$, $b \in \mathsf{ext}(B)$ in $\mathcal{A}_7$, and some sequence reasoning, similar to that needed for $\mathcal{C}_3$ of the proof of Lemma 10.4.6. $\Box$

The result holds analogously for "pushing actions through" mksch which are only from the signature of A instead of that of B.

In the following we prove the most sophisticated lemma of the entire compositionality proof. As there is no way to satisfiy the admissibility check, we apply the take lemma.

Lemma 10.4.13
Given a schedule which results form merging two schedules sch_A and sch_B of A and B and an oracle tr, filtering it w.r.t. actions of A yields sch_A again.

$$\begin{array}{ll}
\mathcal{A}_1: & \mathsf{compatible}\ A\ B \\
\mathcal{A}_2: & \mathsf{is\text{-}sig\text{-}of}(A) \wedge \mathsf{is\text{-}sig\text{-}of}(B) \\
\mathcal{A}_3: & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ tr \\
\mathcal{A}_4: & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(A))\ sch_A;\ \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(B))\ sch_B \\
\mathcal{A}_5: & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `tr = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ A)\ `sch_A \\
\mathcal{A}_6: & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ `tr = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ `sch_B \\
\mathcal{A}_7: & \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch_A \wedge \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch_B \\
\hline
\mathcal{C}_1: & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `(\mathsf{mksch}\ A\ B\ `tr\ `sch_A\ `sch_B) = sch_A
\end{array}$$

Proof.
We apply the proof rule in Thm. 6.5.5 that supports the take lemma together with induction from all $k < n$ to n, and use the instantiation $Q := \lambda x. x \in \mathsf{act}(A) \wedge x \notin \mathsf{act}(B)$. Therefore, we have to solve two subgoals, called *basic step* and *main step* in the sequel:

Basic step: The further assumption

$$\mathcal{A}_8: \quad \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(B) \wedge x \notin \mathsf{act}(A))\ tr$$

is guaranteed by the proof rule 6.5.5. The following case distinction is made:

- **Case** $\mathsf{Finite}(tr)$ **:** The goal is to show, that both sides of $\mathcal{C}_1$ equal nil. To show that $sch_A = \mathsf{nil}$, we employ Lemma 10.4.8 and $\mathcal{A}_7$, so that it remains to show

$$\mathcal{C}_2: \quad \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}(A))\ `sch_A = \mathsf{nil}$$

which equals $\mathsf{Filter}(\lambda x.\, x \in \mathsf{act}\, A)\text{`}tr = \mathsf{nil}$ according to $\mathcal{A}_5$. This in turn is discharged using the sequence lemma

$$\mathsf{Forall}\ Q\ s \wedge \mathsf{Finite}(s) \wedge (\forall a.\ Q(a) \Rightarrow \neg P(a)) \Rightarrow \mathsf{Filter}\ P\ \text{`}s = \mathsf{nil}$$

together with $\mathcal{A}_8$.

To show that $\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}(\mathsf{mksch}\ A\ B\ \text{`}tr\ \text{`}sch_A\ \text{`}sch_B) = \mathsf{nil}$, we employ the same sequence lemma as above to generate the following proof obligations:

$\mathcal{C}_3$: $\mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(B) \wedge x \notin \mathsf{act}(A))\ (\mathsf{mksch}\ A\ B\ \text{`}tr\ \text{`}sch_A\ \text{`}sch_B)$
$\mathcal{C}_4$: $\mathsf{Finite}\ (\mathsf{mksch}\ A\ B\ \text{`}tr\ \text{`}sch_A\ \text{`}sch_B)$
$\mathcal{C}_5$: $x \in \mathsf{act}(B) \wedge x \notin \mathsf{act}(A) \Rightarrow x \notin \mathsf{act}(A)$

$\mathcal{C}_3$ follows from Lemma 10.4.9 together with $\mathcal{A}_1$ and $\mathcal{A}_8$, $\mathcal{C}_4$ follows from Lemma 10.4.10 together with $\mathcal{A}_1$ – $\mathcal{A}_8$, and $\mathcal{C}_5$ is trivial.

- **Case** $\neg\mathsf{Finite}(tr)$ **:** Similar to the previous case, it is shown that both sides of $\mathcal{C}_1$ equal $\perp$. Instead of Lemma 10.4.10 we have to use Lemma 10.4.11.

Main step: We assume the induction hypothesis

$$\begin{array}{ll}\mathcal{A}_8 & \forall m.\ m < n \Rightarrow \forall tr\ sch_A\ sch_B.\\ & \quad \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ tr \wedge\\ & \quad \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ A)\ sch_A \wedge \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ B)\ sch_B \wedge\\ & \quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}tr = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ A)\ \text{`}sch_A \wedge\\ & \quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}tr = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ \text{`}sch_B \wedge\\ & \quad \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch_A;\ \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ sch_B\\ & \quad \Rightarrow \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}(\mathsf{mksch}\ A\ B\ \text{`}tr\ \text{`}sch_A\ \text{`}sch_B)|_m = sch_A|_m\end{array}$$

where we write $|_m$ for $\mathsf{take}\ n$ and, in parts instead of $\mathcal{A}_3$, $\mathcal{A}_5$, $\mathcal{A}_6$:

$\mathcal{A}_9$: $\mathsf{Forall}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ (bs\ \oplus\ a\hat{\ }rs)$
$\mathcal{A}_{10}$: $\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}(bs\ \oplus\ a\hat{\ }rs) = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ A)\ \text{`}sch_A$
$\mathcal{A}_{11}$: $\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}(bs\ \oplus\ a\hat{\ }rs) = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ \text{`}sch_B$
$\mathcal{A}_{12}$: $\mathsf{Finite}(bs) \wedge \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(B) \wedge x \notin \mathsf{act}(A))\ bs$
$\mathcal{A}_{13}$: $a \notin \mathsf{act}(B) \vee a \in \mathsf{act}(A)$

It has to be proved:

$$\mathcal{C}_2\text{:}\quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}(\mathsf{mksch}\ A\ B\ \text{`}(bs\ \oplus\ a\hat{\ }rs)\ \text{`}sch_A\ \text{`}sch_B)|_n = sch_A|_n$$

First, Lemma 10.4.12 allows forward reasoning on the assumptions $\mathcal{A}_4$, $\mathcal{A}_{11}$, and $\mathcal{A}_{12}$. Therefore, there are x_1 and x_2 such that

$$\begin{array}{ll}
\mathcal{A}_{14}\text{:} & \mathsf{mksch}\ A\ B\ `(bs \oplus a\hat{\ }rs)\ `sch_A\ `sch_B = x_1 \oplus \mathsf{mksch}\ A\ B\ `(a\hat{\ }rs)\ `sch_A\ `x_2 \\
\mathcal{A}_{15}\text{:} & \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{act}\ B \wedge a \notin \mathsf{act}\ A)\ x_1 \\
\mathcal{A}_{16}\text{:} & \mathsf{Finite}\ x_1 \\
\mathcal{A}_{17}\text{:} & sch_B = (x_1 \oplus x_2) \\
\mathcal{A}_{18}\text{:} & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ B)\ `x_1 = bs
\end{array}$$

Using $\mathcal{A}_{14}$ we reduce $\mathcal{C}_3$ to

$$\mathcal{C}_4\text{:}\quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `(x_1 \oplus \mathsf{mksch}\ A\ B\ `(a\hat{\ }rs)\ `sch_A\ `x_2)|_n = sch_A|_n$$

Furthermore note, that $\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `x_1 = \mathsf{nil}$ because of the sequence lemma

$$\mathsf{Forall}\ Q\ s \wedge \mathsf{Finite}(s) \wedge (\forall a.\ Q(a) \Rightarrow \neg P(a)) \quad \Rightarrow \quad \mathsf{Filter}\ P\ `s = \mathsf{nil} \quad (*)$$

together with $\mathcal{A}_{12}$. Therefore, as Filter distributes over $\oplus$, $\mathcal{C}_4$ reduces to

$$\mathcal{C}_5\text{:}\quad \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `(\mathsf{mksch}\ A\ B\ `(a\hat{\ }rs)\ `sch_A\ `x_2)|_n = sch_A|_n$$

Now, we have obtained a statement similar to the initial proposition, but with a there is an element in front of the oracle, that will not be eliminated by the Filter operation like the elements in bs before. Roughly speaking, the effort till now was the tribute for working with the take lemma instead of structural induction. From now on, the proof is in some sense analogous to familiar inductive proofs.

We make a case distinction on a. Because of $\mathcal{A}_{13}$ there are only three cases:

- **Case** $a \in \mathsf{act}(A) \wedge a \notin \mathsf{act}(B)$**:** Let us first perform forward reasoning on some of the assumptions. The assumptions $\mathcal{A}_9$ – $\mathcal{A}_{11}$ are transformed into

$$\begin{array}{ll}
\mathcal{A}_{19}\text{:} & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{ext}\ (A \parallel B))\ rs \\
\mathcal{A}_{20}\text{:} & a\hat{\ }(\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ `rs) = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ A)\ `sch_A \\
\mathcal{A}_{21}\text{:} & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ `rs = \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ `x_2
\end{array}$$

  $\mathcal{A}_{19}$ follows from $\mathcal{A}_9$ by the distributivity of Forall over $\oplus$. $\mathcal{A}_{20}$ follows from $\mathcal{A}_{10}$ and $\mathcal{A}_{12}$ by an argument similar to that used to derive $\mathcal{C}_5$. $\mathcal{A}_{21}$ follows from $\mathcal{A}_{11}$ by the following two calculations:

$$\begin{array}{ll}
 & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ `(bs \oplus a\hat{\ }rs) \\
= & \quad \{\mathcal{A}_{12}\ ,\ \mathcal{A}_{13},\ \textit{distributivity of}\ \mathsf{Filter}\ \textit{over}\ \oplus,\ (*)\} \\
 & bs \oplus \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ `rs \\
= & \quad \{\mathcal{A}_{18}\} \\
 & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ B)\ `x_1 \oplus \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ `rs)
\end{array}$$

which shows the transformation of the left side, and

$$
\begin{array}{ll}
 & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ `sch_B \\
= & \quad \{\mathcal{A}_{17}\} \\
 & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ `(x_1\ \oplus\ x_2) \\
= & \quad \{\mathit{Distributivity\ of}\ \mathsf{Filter}\ \mathit{over}\ \oplus\} \\
 & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ `x_1\ \oplus\ \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ B)\ `x_2
\end{array}
$$

which shows the transformation of the right side. Now, Lemma 6.5.2 is used to derive

$$
\begin{array}{ll}
\mathcal{A}_{22}: & sch_A = \mathsf{Takewhile}\ (\lambda x.\ x \notin \mathsf{ext}\ A)\ `sch_A \\
 & \qquad \oplus\ a\hat{}\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \notin \mathsf{ext}\ A)\ `sch_A) \\
 & \wedge\ \mathsf{Finite}\ (\mathsf{Takewhile}\ (\lambda x.\ x \notin \mathsf{ext}\ A)\ `sch_A \wedge a \in \mathsf{ext}(A)
\end{array}
$$

from $\mathcal{A}_{20}$. Because of $\mathcal{A}_2$ it is possible to rewrite the resulting assumption with $\mathsf{is\text{-}sig}\ S \Rightarrow (a \notin \mathsf{externals}\ S) = (x \in \mathsf{internals}\ S \vee x \notin \mathsf{actions}\ S)$. Subsequently, the assumption is rewritten with $\mathsf{Forall}\ P\ s \Rightarrow \mathsf{Takewhile}\ (\lambda x.\ Q(x) \vee (\neg P(x)))\ `s = \mathsf{Takewhile}\ Q\ `s$ and the analogous lemma for Dropwhile, which is possible because of $\mathcal{A}_4$. Therefore, $\mathcal{A}_{22}$ turns into

$$
\begin{array}{ll}
\mathcal{A}_{23}: & sch_A = \mathsf{Takewhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A \\
 & \qquad \oplus\ a\hat{}\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A) \\
 & \wedge\ \mathsf{Finite}\ (\mathsf{Takewhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A \wedge a \in \mathsf{ext}(A)
\end{array}
$$

Let us return to the proposition to prove. $\mathcal{C}_5$ is rewritten by the rule for mksch to

$$
\begin{array}{ll}
\mathcal{C}_6: & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ ` \\
 & \ (\mathsf{Takewhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A \\
 & \quad \oplus\ a\hat{}\mathsf{mksch}\ A\ B\ `rs\ `(\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A))\ `x_2)|_n = \\
 & sch_A|_n
\end{array}
$$

This is simplified using distributivity of Filter over $\oplus$ and the lemma $(\forall a.\ Q(a) \Rightarrow P(a)) \Rightarrow \mathsf{Filter}\ P\ `(\mathsf{Takewhile}\ Q\ `s) = \mathsf{Takewhile}\ Q\ `s$. Furthermore, on the right side the equation for sch_A in $\mathcal{A}_{23}$ is substituted. Therefore, we get:

$$
\begin{array}{ll}
\mathcal{C}_7: & \mathsf{Takewhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A \\
 & \quad \oplus\ a\hat{}\mathsf{mksch}\ A\ B\ `rs\ `(\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A))\ `x_2)|_n = \\
 & \mathsf{Takewhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A\ \oplus\ a\hat{}\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A)|_n
\end{array}
$$

As the prefixes of both sides coincide, the compared sequence length can be reduced using Lemma 6.5.6. Therefore, it remains to show that for all k with $k < n$ holds:

$$
\begin{array}{ll}
\mathcal{C}_8: & \mathsf{mksch}\ A\ B\ `rs\ `(\mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A))\ `x_2)|_k = \\
 & \mathsf{TL}\ `(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ `sch_A)|_k
\end{array}
$$

This matches the proposition of the induction hypothesis. In order to apply it, some assumptions have to be further transformed. First, we get

$$\mathcal{A}_{24} \quad \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(A))\ (\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ \text{`}sch_A))$$
$$\mathcal{A}_{25} \quad \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}(B))\ x_2$$

$\mathcal{A}_{24}$ follows from the fact that Forall is closed under the TL and Dropwhile operations, and $\mathcal{A}_{25}$ follows from $\mathcal{A}_{17}$ and the distributivity of Forall over $\oplus$. Next, $\mathcal{A}_{20}$ is further transformed to

$$\begin{aligned}\mathcal{A}_{26}\colon\ & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}rs = \\ & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{ext}\ A)\ \text{`}(\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ \text{`}sch_A)))\end{aligned}$$

by substituting the equation for sch_A in $\mathcal{A}_{23}$ and using sequence lemma (*). Finally, $\mathcal{A}_7$ is transformed into

$$\mathcal{A}_{27} \quad \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ (\mathsf{TL}\ \text{`}(\mathsf{Dropwhile}\ (\lambda x.\ x \in \mathsf{int}\ A)\ \text{`}sch_A))) \wedge \mathsf{Last\text{-}act\text{-}is\text{-}ext}\ A\ x_2$$

using the facts, that Last-act-is-ext is closed under TL and Dropwhile, and distributes over $\oplus$. Therefore, the induction hypothesis can be applied using the instantiations $tr := rs$, $sch_A := \mathsf{TL}\,\text{`}(\mathsf{Dropwhile}\,(\lambda x.\, x \in \mathsf{int}\,A)\,\text{`}sch_A))$, and $sch_B := x_2$, which finishes the proof of this case.

- **Case** $a \in \mathsf{act}(A) \wedge a \in \mathsf{act}(B)$**:** Even more elaborate than the previous case, but analogous.
- **Case** $a \notin \mathsf{act}(A) \wedge a \notin \mathsf{act}(B)$**:** From $\mathcal{A}_9$ and the fact that Forall distributes over $\oplus$ it follows that $a \in \mathsf{ext}\,(A \parallel B)$, which is equal to $a \in \mathsf{ext}(A) \vee a \in \mathsf{ext}(B)$. Therefore, a contradiction to the actual case is derived from the fact $a \in \mathsf{ext}(A) \Rightarrow a \in \mathsf{act}(A)$, which finishes the proof. □

The result holds analogously for filtering on the actions of B, yielding sch_B instead of sch_A.

Definition 10.4.14 (Trace Property Composition)
Parallel composition on trace properties is defined as follows:

$$\|_{tr} \quad : \quad (\alpha)\,\mathsf{trace\text{-}prop} \to (\alpha)\,\mathsf{trace\text{-}prop} \to (\alpha)\,\mathsf{trace\text{-}prop}$$

$$\begin{aligned}&(sig_A, traces_A)\ \|_{ex}\ (sig_B, traces_B) \equiv \\ &\{\mathsf{sig\text{-}comp}\ sig_A\ sig_B, \\ &\qquad \{tr \mid \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ \text{`}tr \in traces_A\} \\ &\ \ \cap\ \{tr \mid \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ \text{`}tr \in traces_B\} \\ &\ \ \cap\ \{tr \mid \mathsf{Forall}\ (\lambda a.\ a \in (\mathsf{externals}\ sig_A \cup \mathsf{externals}\ sig_A))\ tr\}\}\end{aligned}$$

Again we rephrase the compositionality result of Theorem 10.4.1 in a succinct way.

Corollary 10.4.15 (Compositionality of Trace Properties)
Trace properties are compositional.

$$\frac{\begin{array}{l}\mathsf{compatible}\ A\ B\\ \mathsf{is\text{-}trans\text{-}of}(A) \wedge \mathsf{is\text{-}trans\text{-}of}(B)\\ \mathsf{is\text{-}sig\text{-}of}(A) \wedge \mathsf{is\text{-}sig\text{-}of}(B)\end{array}}{\mathsf{Traces}\ (A \parallel B) = \mathsf{Traces}(A) \parallel_{tr} \mathsf{Traces}(B)}$$

10.5 Compositional Reasoning

In order to prove the main compositionality result we merely need a single further lemma.

Lemma 10.5.1
Compatibility ensures that for traces which contain only external actions of A or B filtering to external and all actions of A coincides.

$$\frac{\mathcal{A}_1\colon \mathsf{compatible}\ A\ B \quad \mathcal{A}_2\colon \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{ext}\ (A \parallel B))\ tr}{\mathcal{C}_1\colon \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A)\ `tr = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ A)\ `tr}$$

Proof.
We first apply the sequence lemma

$$\frac{\forall x.\ P\ x \Rightarrow (Q\ x = R\ x) \qquad \mathsf{Forall}\ P\ s}{\mathsf{Filter}\ Q\ `s = \mathsf{Filter}\ R\ `s}$$

Instantiating P by $\lambda a.\ a \in \mathsf{ext}\ (A \parallel B)$ the goal $\mathcal{C}_1$ reduces to

$$\begin{array}{ll}\mathcal{C}_2\colon & a \in \mathsf{ext}\ (A \parallel B) \Rightarrow a \in \mathsf{act}\ A = a \in \mathsf{ext}\ A\\ \mathcal{C}_3\colon & \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{ext}\ (A \parallel B))\ tr\end{array}$$

where $\mathcal{C}_3$ can directly be discharged by $\mathcal{A}_2$. $\mathcal{C}_2$ can be reduced by rewriting $a \in \mathsf{ext}\,(A \parallel B)$ to $a \in \mathsf{ext}(A) \vee a \in \mathsf{ext}(B)$ and automatic first order reasoning to

$$\begin{array}{ll}\mathcal{C}_4\colon & a \in \mathsf{ext}(A) \Rightarrow a \in \mathsf{act}(A)\\ \mathcal{C}_5\colon & a \in \mathsf{ext}(B) \wedge a \notin \mathsf{ext}(A) \Rightarrow a \notin \mathsf{act}(A)\end{array}$$

The first holds by definition of ext, the second is a consequence of the compatibility of A and B, which is guaranteed by$\mathcal{A}_1$. □

Theorem 10.5.2 (Main Compositionality Theorem)
The safe implementation relation is compositional.

$$\frac{\begin{array}{ll} \mathcal{A}_1\colon & \mathsf{is\text{-}trans\text{-}of}(A_1) \wedge \mathsf{is\text{-}trans\text{-}of}(A_2) \wedge \mathsf{is\text{-}trans\text{-}of}(B_1) \wedge \mathsf{is\text{-}trans\text{-}of}(B_2) \\ \mathcal{A}_2\colon & \mathsf{is\text{-}sig\text{-}of}(A_1) \wedge \mathsf{is\text{-}sig\text{-}of}(A_2) \wedge \mathsf{is\text{-}sig\text{-}of}(B_1) \wedge \mathsf{is\text{-}sig\text{-}of}(B_2) \\ \mathcal{A}_3\colon & \mathsf{compatible}\ A_1\ B_1 \wedge \mathsf{compatible}\ A_2\ B_2 \\ \mathcal{A}_4\colon & A_1 \preceq_S A_2 \\ \mathcal{A}_5\colon & B_1 \preceq_S B_2 \end{array}}{\mathcal{C}_1\colon\ (A_1 \parallel B_1) \preceq_S (A_2 \parallel B_2)}$$

Proof.
By definition of $\preceq_S$ $\mathcal{A}_4$ and $\mathcal{A}_5$ rewrite to:

$$\begin{array}{ll} \mathcal{A}_6\colon & \mathsf{traces}\ A_1 \subseteq \mathsf{traces}\ A_2 \wedge \mathsf{in}\ A_1 = \mathsf{in}\ A_2 \wedge \mathsf{out}\ A_1 = \mathsf{out}\ A_2 \\ \mathcal{A}_7\colon & \mathsf{traces}\ B_1 \subseteq \mathsf{traces}\ B_2 \wedge \mathsf{in}\ B_1 = \mathsf{in}\ B_2 \wedge \mathsf{out}\ B_1 = \mathsf{out}\ B_2 \end{array}$$

Similarly, $\mathcal{C}_1$ can be reduced to:

$$\begin{array}{ll} \mathcal{C}_2\colon & \mathsf{traces}\ (A_1 \parallel B_1) \subseteq \mathsf{traces}\ (A_2 \parallel B_2) \\ \mathcal{C}_3\colon & \mathsf{in}\ (A_1 \parallel B_1) = \mathsf{in}\ (A_2 \parallel B_2) \wedge \mathsf{out}\ (A_1 \parallel B_1) = \mathsf{out}\ (A_2 \parallel B_2) \end{array}$$

$\mathcal{C}_3$ can easily be discharged by the definition of in and $\parallel$ using $\mathcal{A}_6$ and $\mathcal{A}_7$. To reduce $\mathcal{C}_2$ we use compositionality on the trace level (Theorem 10.4.1) for which $\mathcal{A}_1$ – $\mathcal{A}_3$ are necessary. Therefore, the following has to be shown for every tr

$$\begin{array}{ll} \mathcal{C}_4\colon & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A_2)\ `tr \in \mathsf{traces}\ A_2 \wedge \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B_2)\ `tr \in \mathsf{traces}\ B_2 \\ \mathcal{C}_5\colon & \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{ext}\ (A_2 \parallel B_2))\ tr \end{array}$$

under the further assumptions

$$\begin{array}{ll} \mathcal{A}_8\colon & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A_1)\ `tr \in \mathsf{traces}\ A_1 \wedge \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B_1)\ `tr \in \mathsf{traces}\ B_1 \\ \mathcal{A}_9\colon & \mathsf{Forall}\ (\lambda a.\ a \in \mathsf{ext}\ (A_1 \parallel B_1))\ tr \end{array}$$

From $\mathcal{A}_6$ and $\mathcal{A}_7$ we get that the external actions of the components are equal:

$$\mathcal{A}_{10}\colon\ \ \mathsf{ext}(A_1) = \mathsf{ext}(A_2) \wedge \mathsf{ext}(B_1) = \mathsf{ext}(B_2)$$

Therefore, using $\mathcal{A}_9$ and $\mathsf{ext}\ (C_1 \parallel C_2) = \mathsf{ext}\ C_1 \cup \mathsf{ext}\ C_2$, $\mathcal{C}_5$ can be discharged. It remains to show $\mathcal{C}_4$. Using the trace inclusion properties of the components in $\mathcal{A}_6$ and $\mathcal{A}_7$ it can be inferred by forward reasoning from $\mathcal{A}_8$ that the filtered tr is not only in A_1 but also in A_2, and analogously for B_1 and B_2:

$$\mathcal{A}_{11}\colon\ \ \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A_1)\ `tr \in \mathsf{traces}\ A_2 \wedge \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B_1)\ `tr \in \mathsf{traces}\ B_2$$

Therefore, to show $\mathcal{C}_4$ it suffices to show

$$\begin{aligned}\mathcal{C}_6\text{:}\quad & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A_1)\ \text{`}tr = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A_2)\ \text{`}tr \wedge \\ & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B_1)\ \text{`}tr = \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ B_2)\ \text{`}tr\end{aligned}$$

We consider only the first conjunct of $\mathcal{C}_6$, the case for B_1, B_2 follows analogously. To prove $\mathcal{C}_6$ we make use of the fact that tr has only external actions of A_1 or A_2, which implies together with the compatibility of the component automata that filtering all actions from tr actually means filtering only the external ones (Lemma 10.5.1). However, the external actions are equal according to $\mathcal{A}_{10}$. Formally, the following equalities terminate the proof:

$$\begin{aligned} & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A_1)\ \text{`}tr \\ = & \quad \{\ \textit{Lemma 10.5.1},\ \mathcal{A}_3,\ \mathcal{A}_9\ \} \\ & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ A_1)\ \text{`}tr \\ = & \quad \{\mathcal{A}_{10}\} \\ & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{ext}\ A_2)\ \text{`}tr \\ = & \quad \{\ \textit{Lemma 10.5.1},\ \mathcal{A}_3,\ \mathcal{A}_9,\ \mathcal{A}_{10}\ \} \\ & \mathsf{Filter}\ (\lambda a.\ a \in \mathsf{act}\ A_2)\ \text{`}tr \end{aligned}$$

□

10.6 Non-Interference

In this section a central property of I/O automata, called *non-interference*, is derived: in an arbitrary network of I/O automata a component A may always perform a local action. The other components cannot block A, instead they either have to stutter or to enable the desired communication. As a result, when reasoning about the enabling of an action in this network, it is sufficient to reason about the enabling of the action at one component. This property is mainly a consequence of the input-enabling of I/O automata and of their compatibility. As the input-enabling condition was not yet necessary in the preceding proofs, it is of interest how it gets involved here. In addition, non-interference needs compositionality on schedule level and some lemmas that have been derived in the context of the correctness proof of refinement mappings. Therefore the proof of this property may be regarded as a remarkable quintessence of the previous verification efforts.

We start with a lemma, that reveals the influence of the input-enabling condition.

Lemma 10.6.1 (Schedules are always extendible by Inputs)

$$\frac{\mathcal{A}_1\text{: } a \in \mathsf{in}(A) \quad \mathcal{A}_2\text{: } \mathsf{input\text{-}enabled}(A) \quad \mathcal{A}_3\text{: } \mathsf{Finite}(sch) \quad \mathcal{A}_4\text{: } sch \in \mathsf{schedules}(A)}{\mathcal{C}_1\text{: } (sch\ \oplus\ [a!]) \in \mathsf{schedules}(A)}$$

Proof.
We first perform forward reasoning on the assumptions. $\mathcal{A}_4$ guarantees the existence of an execution (s, ex_1) with the following properties:

$$\mathcal{A}_5: \quad s \in \mathsf{starts\text{-}of}(A) \wedge \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, ex_1) \wedge sch = \mathsf{Filter\text{-}act}\ `ex_1$$

From $sch = \mathsf{Filter\text{-}act}\ `ex_1$ and $\mathsf{Finite}(sch)$ ($\mathcal{A}_3$) follows $\mathsf{Finite}(ex_1)$ by the lemma

$$\mathsf{Finite}\ (\mathsf{Map}\, f\ `seq) \Rightarrow \mathsf{Finite}\ seq$$

which holds for every sequence seq. As every finite execution has a last state, the existence of a state u with

$$\mathcal{A}_6: \quad \mathsf{last\text{-}state}\ (s, ex_1) = u$$

is given. $\mathcal{A}_2$ ensures input-enabledness of A, i.e. $\forall s.\, \exists t.\ s \xrightarrow{a}_A t$. Therefore, there is also a state t from the last state of (s, ex_1):

$$\mathcal{A}_7: \quad u \xrightarrow{a}_A t$$

Now we turn to the proposition $\mathcal{C}_1$ and reduce it by unfolding the schedule definition to:

$$\mathcal{C}_2: \quad \exists ex_2 \in \mathsf{executions}(A).\ (sch\ \oplus\ [a!]) = \mathsf{Filter\text{-}act}\ `(\mathsf{snd}\ ex_2)$$

Such an ex_2 can be defined as an extension of ex_1 as follows:

$$ex_2 := (s, ex_1\ \oplus\ [(a, t)!])$$

Therefore, using the definition of $\mathsf{executions}$ $\mathcal{C}_2$ reduces to

$$\begin{aligned} &\mathcal{C}_3: \quad s \in \mathsf{starts\text{-}of}(A) \\ &\mathcal{C}_4: \quad \mathsf{is\text{-}exec\text{-}frag}\ A\ (s, ex_1\ \oplus\ [(a, t)!]) \\ &\mathcal{C}_5: \quad (sch\ \oplus\ [a!]) = \mathsf{Filter\text{-}act}\ `(ex_1\ \oplus\ [(a, t)!]) \end{aligned}$$

$\mathcal{C}_3$ follows from $\mathcal{A}_5$. Using the distributivity of Map over $\oplus$, $\mathcal{C}_5$ reduces to $sch = \mathsf{Filter\text{-}act}`ex_1$ which is trivial also from $\mathcal{A}_5$. It remains to show $\mathcal{C}_4$. Here Lemma 9.2.7 is applied, which states that the $\mathsf{is\text{-}exec\text{-}frag}$ property propagates under certain conditions from sequences to their concatenation. The free variable in the rule is instantiated by u. Therefore it remains to show:

$$\mathcal{C}_6: \quad \mathsf{is\text{-}exec\text{-}frag}\ (s, ex_1) \wedge \mathsf{is\text{-}exec\text{-}frag}\ (u, [(a, t)!]) \wedge \mathsf{last\text{-}state}\ (s, ex_1) = u.$$

The first and last conjunct of $\mathcal{C}_6$ are discharged by $\mathcal{A}_5$ and $\mathcal{A}_6$, respectively. The second conjunct rewrites to $u \xrightarrow{a}_A t$ and therefore follows from $\mathcal{A}_7$. $\square$

Theorem 10.6.2 (Non-Interference of I/O Automata)

$$\begin{array}{ll}
\mathcal{A}_1\text{:} & a \in \mathsf{local}(A) \\
\mathcal{A}_2\text{:} & \mathsf{compatible}\ A\ B \\
\mathcal{A}_3\text{:} & \mathsf{input\text{-}enabled}(B) \\
\mathcal{A}_4\text{:} & \mathsf{Finite}(sch) \\
\mathcal{A}_5\text{:} & sch \in \mathsf{schedules}\ (A \parallel B) \\
\mathcal{A}_6\text{:} & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(A))\ \text{`}(sch\ \oplus\ [a!]) \in \mathsf{schedules}(A) \\
\hline
\multicolumn{2}{c}{\mathcal{C}_1\text{: } (sch\ \oplus\ [a!]) \in \mathsf{schedules}\ (A \parallel B)}
\end{array}$$

Proof.
Compositionality on schedule level (Theorem 10.3.1) allows to derive

$$\begin{array}{ll}
\mathcal{A}_7\text{:} & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}sch \in \mathsf{schedules}\ A \\
\mathcal{A}_8\text{:} & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}sch \in \mathsf{schedules}\ B \\
\mathcal{A}_9\text{:} & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ sch
\end{array}$$

from $\mathcal{A}_5$. The same theorem applies to $\mathcal{C}_1$ and reduces it to

$$\begin{array}{ll}
\mathcal{C}_2\text{:} & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ A)\ \text{`}(sch\ \oplus\ [a!]) \in \mathsf{schedules}\ A \\
\mathcal{C}_3\text{:} & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}(sch\ \oplus\ [a!]) \in \mathsf{schedules}\ B \\
\mathcal{C}_4\text{:} & \mathsf{Forall}\ (\lambda x.\ x \in \mathsf{act}\ (A \parallel B))\ sch \wedge a \in \mathsf{local}(A)
\end{array}$$

$\mathcal{C}_2$ is discharged by $\mathcal{A}_6$, whereas the first conjunct in $\mathcal{C}_4$ is just $\mathcal{A}_9$ and the second is a simple consequence of $\mathcal{A}_1$. Therefore it remains to show $\mathcal{C}_3$. This is done by case distinction on a.

Case $a \in \mathsf{int}(A)$: Because of the compatibility of A and B ($\mathcal{A}_2$) a cannot be an action of B. Therefore, intuitively speaking, B has to stutter in the current step. Formally, the following equalities yield the result:

$$\begin{array}{cl}
 & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}(sch\ \oplus\ [a!]) \\
 & \quad \{\textit{Distributivity of}\ \mathsf{Filter}\ \textit{over}\ \oplus\} \\
= & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}sch\ \oplus\ \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}[a!] \\
 & \quad \{a \notin \mathsf{act}(B)\} \\
= & \mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ \text{`}sch \\
 & \quad \{\mathcal{A}_8\} \\
\in & \in \mathsf{schedules}(B)
\end{array}$$

Case $a \in \mathsf{out}(A)$: Because of the compatibility of A and B ($\mathcal{A}_2$) a can only be an input action of B. Note that here not only the encapsulation of internal actions is needed but also the condition $\mathsf{out}(A) \cap \mathsf{out}(B) = \{\}$ of the compatibility requirement. In the compatibility proofs this was not the case. Intuitively speaking, B can now always extend the schedule

$\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}(B))\ `sch$ by this input action a because B is input-enabled. Therefore we get the desired property:

$$\mathsf{Filter}\ (\lambda x.\ x \in \mathsf{act}\ B)\ `sch \in \mathsf{schedules}(B)$$

Formally, the property is an application of the preceding lemma, using the assumptions $\mathcal{A}_3$, $\mathcal{A}_4$, and $\mathcal{A}_8$, together with distributivity of Filter over $\oplus$ and the fact that any finite sequence remains finite after filtering. $\square$

10.7 Conclusion and Related Work

The compositionality proof turned out to be a nontrivial challenge. In particular it permits to draw conclusions w.r.t. the power and applicability of HOLCF on the one hand, and the general insufficiency of semi-formal paper proofs on the other hand. Both aspects will be discussed subsequently.

- For the proof it was essential to take advantage of HOLCF's possibilities to define and reason about sophisticated recursive functions. We argue that a definition of the necessary merge functions in pure HOL would be extremely awkward, if not infeasible. Despite of these significant strengths of HOLCF we also reached its limits:
  First, admissibility proof obligations occurred, which could not be discharged automatically. This problem could be circumvented by providing and applying the take lemma infrastructure of §6.5, which does not need admissibility. We believe that this proof infrastructure will be mandatory for every serious application of sequences in HOLCF.
  Second, we faced a noncontinuous fair merge function. In our concrete context, we got along with a modification of the overall proof. However, this modification would not be applicable to a generalization of the compositionality proof to *fair* or *live* I/O automata. The reason is that fairness considers internal actions as well. Therefore, it is not possible to cut off the internal rests of an execution without making the corresponding trace potentially unfair. This is the reason why we proved compositionality only for safe I/O automata and did not extend the result to live I/O automata. In addition, it demonstrates that the de facto restriction of HOLCF to continuous functions can become a handicap in complex applications.

- The corresponding compositionality proof in the literature [LT87], which is based on an informal sequence model, comprises only two pages. This has several reasons:
  First, the relation between behaviours of a composition and its components is described implicitly, whereas in our rigorous approach this relation has to be made explicit by defining nontrivial merge functions. It is not clear at all how to prove the existence of such a relation if it is described only implicitly. Our explicit merge

function seems to be the most feasable way to perform the proof, but it nevertheless poses remarkable complications. For details recall the comparative discussions in the respective sections.
Furthermore, and even more importantly, reasoning about this informal relation is intuitively appealing, whereas a rigorous argumentation brings about unexpected complications. In our formalization the complications manifest themselves in nontrivial admissibility obligations. Other complications are to be expected for alternative sequence models, for example nontrivial limes constructions for the model $\mathsf{nat} \rightarrow \mathsf{bool}$. In fact, the major complexity of the proof is due to these complications, which are hidden under the surface of the informal proof.

We conclude that the power of HOLCF should only be used carefully at places where it is really needed. This confirms our HOL/HOLCF methodology described in §5.2.

Furthermore, we conclude that rigorous proofs are worthwhile, as they often reveal subtle details which are easily neglected in an informal argumentation, but have significant influence on the entire proof. This applies in particular to formalisms involving (possibly) infinite sequences, as sequences are intuitively simple, but require quite a mass of theory when being treated formally.

To our knowledge, there is no directly related work.

Chapter 11

Temporal Logic, Live I/O Automata, and Abstraction

In this chapter the extensions to the theory of I/O automata described in the chapters 3–4 are definitionally embedded in Isabelle. This includes the temporal logic TLS, live I/O automata, and the abstraction theory. Thereby TLS is built on top of a generic temporal logic. This allows us to study semantic embeddings of temporal logics in general and, furthermore, reveals the connection to the temporal logic of Manna and Pnueli.

11.1 Introduction

In this chapter the notions developed in §3 – §4 are definitionally embedded in Isabelle, building on top of the sequence model of §6 and the safe I/O automaton model of §8–§10. The extensions include the temporal logic TLS, live I/O automata, and the abstraction theory. Thereby, TLS is not encoded directly, but as an instance of a generic temporal logic which is evaluated over sequences of arbitrary elements. This has several advantages:

- It allows us to study the adequateness of our domain-theoretic sequence model w.r.t. temporal logics in a general setting.
- It reveals the connection to standard temporal logics over state sequences [MP95, Krö87]. In particular, we will see that validities from these logics carry over to our setting, which enables us to reuse many results from [MP95] and [Krö87]. Although the opposite direction does not hold in general, particularities of I/O executions will be captured in specific TLS formulas, which increases the extent of reuse.
- It offers an abstract view on the special treatment of finite computations in TLS.

Using TLS the I/O automaton model in Isabelle is extended to general liveness. To prove live implementation we introduce live refinement mappings and live forward simulations and establish their soundness within Isabelle.

Finally, the entire abstraction theory for functions is definitionally embedded in Isabelle on top of TLS and the live I/O automaton model. Thus, we may use the theorem prover not only to check the soundness of abstractions, but to describe and reason about complex systems in a compositional manner as well.

The chapter is organized as follows. In §11.2 a generic temporal logic is embedded in Isabelle, which is used in §11.3 to embed TLS. In §11.4 live I/O automata are formalized and in §11.5 the abstraction rules are derived within Isabelle.

11.2 A Generic Temporal Logic

In the second part of this chapter we will now embed the notions developed in the previous subsections into Isabelle. In this section we first embed a generic temporal logic over finite and infinite sequences (called TL), compare it with the logics in [Krö87] and [MP95], and evaluate the domain-theoretic sequence model w.r.t. this temporal logic.

We will use a shallow embedding, which means that we do not distinguish between syntax and semantics of temporal formulas as done in §3.2. Instead, formulas are directly regarded as predicates and temporal operators as predicate transformers. However, Isabelle's syntax facilities permit to denote these transformers by the usual syntax.

Definition 11.2.1 (Predicates)
We introduce a type for predicates and a corresponding notion of evaluation, which simply means function application.

$$
\begin{array}{lcl}
(\alpha)\,\mathsf{pred} & = & \alpha \to \mathsf{bool} \\
\models & :: & \alpha \to (\alpha)\,\mathsf{pred} \to \mathsf{bool} \\
x \models P & \equiv & P(x)
\end{array}
$$

The boolean connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $=$ are lifted to predicates in a pointwise way[1]. □

As $(\alpha)\,\mathsf{pred}$ is polymorphic, it will be used to describe both state predicates and sequence predicates. The latter represent temporal formulas and are defined below.

Definition 11.2.2 (TL – A Generic Temporal Logic)
Formulas of the temporal logic TL are described by the type

$$(\alpha)\,\mathsf{temporal} \quad = \quad ((\alpha)\,\mathsf{sequence})\,\mathsf{pred}$$

[1]Because it is always clear from the context whether a connective is lifted or not, we use the same symbols, although in Isabelle the syntax differs slightly.

As lifted boolean connectives exist already for this type, it suffices to define $\Box$, $\bigcirc$, and $\langle - \rangle$, where the latter means lifting a state predicate to a temporal formula.

$$\begin{array}{lcl} \langle - \rangle & :: & (\alpha)\,\mathsf{pred} \to (\alpha)\,\mathsf{temporal} \\ \langle P \rangle & \equiv & \lambda s.\ P\ (\mathsf{the}\ (\mathsf{HD}\ `s)) \\[4pt] \Box, \bigcirc & :: & (\alpha)\,\mathsf{temporal} \to (\alpha)\,\mathsf{temporal} \\ \Box\, P & \equiv & \lambda s.\ \forall s_2.\ s_2 \gg^+ s \Rightarrow P(s_2) \\ \bigcirc\, P & \equiv & \lambda s.\ \textbf{if}\ \mathsf{TL}\ `s = \bot\ \textbf{then}\ P(s)\ \textbf{else}\ P\ (\mathsf{TL}\ `s) \end{array}$$

where suffixes and non-empty suffixes are defined as follows:

$$\begin{array}{lcl} \gg,\ \gg^+ & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \to \mathsf{bool} \\ s_2 \gg s & \equiv & \exists s_1.\ \mathsf{Finite}\ s_1 \wedge s = s_1 \oplus s_2 \\ s_2 \gg^+ s & \equiv & s_2 \neq \mathsf{nil} \wedge s_2 \neq \bot \wedge (s_2 \gg s) \end{array}$$

Further temporal operators are defined as usual:

$$\begin{array}{lcl} \Diamond\, P & \equiv & \neg\Box\neg P \\ P \leadsto Q & \equiv & \Box\,(P \Rightarrow \Diamond\, Q) \end{array}$$

Validity of P means that it holds for all non-empty sequences[2]:

$$\models P \ \equiv\ \forall s.\ s \neq \mathsf{nil} \wedge s \neq \bot \Rightarrow s \models P$$

□

Treatment of Finite Sequences. Obviously, the empty sequence represents a pathological case in every temporal logic involving finite computations. In the definitions above this is reflected by the fact that $\mathsf{nil} \models \langle P \rangle$ equals $P(\mathit{arbitrary})$, where *arbitrary* is a fixed, but unknown value (recall the definition of the in §5.2.2). Therefore nothing reasonable can be concluded for this case. We solve the problem by circumventing the cases $s = \bot$ and $s = \mathsf{nil}$ completely. They are ruled out in the validity definition and the temporal operators are defined in such a way, that they do not introduce new statements of the form $\mathsf{nil} \models P$ or $\bot \models P$[3]. This is the reason why we define $\Box$ using only non-empty suffixes and use the TL operator for $\bigcirc$ only if the sequence consists of at least two elements. Therefore, if $s = [a]$ then $\mathsf{TL}\ `s \neq \bot$ and therefore $\bigcirc P\ s = P\ s$.

The key observation is now, that the exclusion of empty sequences suffices already to treat finite computations. This in turn conforms very well with the semantics of TLS: when embedding TLS in TL in the next section we will see that the additional stuttering step,

[2]In Isabelle different symbols are used for validity and evaluation to avoid ambiguities.

[3]Note however, that $\bot \models P$ and $\mathsf{nil} \models P$ are indeed true in our formalization, but they are not used in the validity definition.

that stutters the last state of non-infinite executions, excludes the cases $s = \mathsf{nil}$ and $s = \bot$. Therefore the stuttering action $\surd$ in TLS can, in a more abstract view, also be regarded as a remedy to rectify the insufficiency of general temporal logics involving non-infinite computations.

Comparison with standard LTL. We will now show that

$$\models_{LTL} P \qquad \text{iff} \qquad \models P$$

where $\models_{LTL}$ denotes validity in [MP95] or [Krö87] and P is restricted to the operators of TL. There are two requirements to satisfy: first, the temporal operators must have the same semantics, and second, it must make no difference whether formulas are evaluated over finite and infinite sequences (for $\models$) or over infinite sequences only (for $\models_{LTL}$).

The first requirement is not completely trivial as we did not define the operators pointwise (as in [MP95]) in order to match our sequence model (see below), but it is easy to see. The second requirement follows by an argument analogous to Lemma 3.2.7. It is trivial that $\models P$ implies $\models_{LTL} P$, as $\models_{LTL}$ regards only the subset of infinite executions considered by $\models$. For the other direction define the operator ∇' that adds infinite stuttering in a slightly modified version as

$$\nabla'\sigma \quad \equiv \quad \begin{cases} \mathit{arbitrary} & \text{if } \sigma = \mathit{nil} \vee \sigma = \bot \\ \sigma s_n s_n \ldots & \text{if } \sigma = s_0 s_1 \ldots s_n \text{ is finite or partial} \\ \sigma & \text{if } \sigma \text{ is infinite} \end{cases}$$

Now, suppose $\models_{LTL} P$, which means that P holds for all infinite sequences. We have to show that P holds for every non-empty sequence σ as well. As $\nabla'\sigma \models P$ is infinite in this case, we directly get the result by the fact that $\sigma \models P$ equals $\nabla'\sigma \models P$, which holds under the assumption $\sigma \notin \{\mathit{nil}, \bot\}$ and follows by an argument similar to that of Lemma 3.2.7.

We now can exploit this comparison to get a completeness result for TL simply by carrying it over from [Krö87]. Below we prove some theorems in Isabelle, which represent a complete set of rules w.r.t. validity in TL according to [Krö87][4].

Lemma 11.2.3 (TL Theorems)
The following theorems about TL have been proved in Isabelle.

$$\frac{\models P \Rightarrow Q \qquad \models P}{\models Q}\ (mp) \qquad\qquad \frac{\models P}{\models \bigcirc P}\ (nex) \qquad\qquad \frac{\models P \Rightarrow Q \quad \models P \Rightarrow \bigcirc P}{\models P \Rightarrow \Box Q}\ (ind)$$

$$\begin{array}{llcll} \models & \bigcirc\neg P & = & \neg\bigcirc P & (ax1) \\ \models & \bigcirc(P \Rightarrow Q) & \Rightarrow & (\bigcirc P \Rightarrow \bigcirc Q) & (ax2) \\ \models & \Box P & \Rightarrow & (P \wedge \bigcirc\Box P) & (ax3) \end{array}$$

[4]Note that this completeness result cannot be proved in Isabelle as we use a shallow embedding.

Proof.
The induction rule (ind) boils down to finite structural induction, where the step case is subtle as elements are added at the end instead of in front of the sequence. The other proofs are straightforward, the proof of ($ax3$) is presented as an example. Unfolding the definitions of $\models$ and $\wedge$, ($ax1$) reduces to the propositions

$$\begin{array}{ll}\mathcal{C}_1: & \Box P(s) \Rightarrow P(s) \\ \mathcal{C}_2: & \Box P(s) \Rightarrow \bigcirc\Box P(s)\end{array}$$

which have to be shown for every s with $s \neq \bot \wedge s \neq \mathsf{nil}$. $\mathcal{C}_1$ reduces further to

$$\mathcal{C}_3: \quad (\forall s_2.\, s_2 \neq \bot \wedge s_2 \neq \mathsf{nil} \wedge s_2 \gg s \Rightarrow P(s_2)) \Rightarrow P(s)$$

which is true because of $s \gg s$ and s is nonempty. $\mathcal{C}_2$ reduces similarly to the lemma

$$s \neq \bot \wedge s \neq \mathsf{nil} \quad \Rightarrow \quad (s_2 \gg \mathsf{TL}\,`s \Rightarrow s_2 \gg s)$$

which finishes the proof for ($ax1$). $\Box$

Domain-Theoretic Sequences and Temporal Logics. The domain-theoretic sequence model in HOLCF turned out to be surprisingly adequate for defining a temporal logic. As the definition of the temporal operators shows, every theorem about TL boils down to sequence lemmas about HD, TL, and $\oplus$. Furthermore, admissibility obligations mostly cease to apply, as $\gg^+$ needs $\oplus$ only with a finite first argument, so that finite structural induction can be applied. If admissibility obligations appear nevertheless, they can usually be discharged automatically, as the operators HD, TL, and $\oplus$ are all defined continuously[5].

This simplicity is the result of a careful choice of the way the operators are formalized. In fact, a pointwise definition like in [MP95] would be inappropriate in our sequence model, the same holds for a $\Box$ operator defined by some kind of drop operator motivated by the semantics of TLA [Lam94]. Furthermore, note that the interface methodology proposed for HOL and HOLCF in §5.2 has been carefully respected by, for example, using the HOL quantifier $\forall$ instead of a HOLCF sequence to describe all suffixes in the definition of $\Box$.

A number of different attempts have already been made to definitionally embed temporal logics in higher-order logic [Lån94, Wri92, Cho93]. Up to our knowledge, only infinite sequences have been considered, which are represented by functions on natural numbers. It is not obvious, how these approaches should be generalized to deal with finite sequences as well. Furthermore, operators that deal with stuttering are not considered there. Take, for example, the operator that eliminates stuttering by replacing all subsequences $s \cdots s$

[5]Note the difference here to the HOLCF sequence model in [SS95] which features only two flavors of sequences and is used for FOCUS [BDD$^+$93]. This model would be inappropriate in this context, as it provides a non-continuous $\oplus$ operator, which implies that proofs about $\oplus$ tend to be tremendously difficult.

by s. This would be some kind of filter operation which is easily dealt with in our setting. In a functional setting, however, we face an operator which relocates elements in an inhomogeneous way, which is extremely awkward to handle according to the results of §7.7.

11.3 The Temporal Logic of Steps

In this section we use the generic temporal logic over sequences of the previous section to embed TLS, i.e. a temporal logic over executions of I/O automata. The idea is to encode executions $\alpha = s_0 a_1 s_1 \ldots$ into a sequence of triples, where every triple (s_i, a_{i+1}, s_{i+1}) represents one step $s_i \overset{a_{i+1}}{\rightarrow}_A s_{i+1}$ of the automaton A. As finite executions are asymmetric in the sense that they contain one more state than actions, a stuttering triple $(s_n, \surd, s_n)$ is added for the last state s_n of finite executions. This ensures that every state of the execution appears as the first component of some triple in the resulting sequence. Furthermore, it matches exactly the "stutter" semantics of TLS.

In Isabelle, this encoding consists of two parts. First, executions, which are represented as state/sequence pairs, are transformed into triple sequences. Note that hereby redundancy is introduced, as most of the states are represented twice. Second, the action type α is extended to $(\alpha)\,\mathsf{option}$, which ensures that the stuttering action, represented by None, is not already an element of the action type α. Both transformations are performed by the function $\mathsf{ex\text{-}to\text{-}seq}$ defined below.

Definition 11.3.1 (Encoding Executions)
The transformation $\mathsf{ex\text{-}to\text{-}seq}$ from executions to sequences is defined as follows

$$
\begin{array}{lcl}
\mathsf{ex\text{-}to\text{-}seq} & :: & (\alpha,\sigma)\,\mathsf{execution} \to (\sigma \times (\alpha)\,\mathsf{option} \times \sigma)\,\mathsf{sequence} \\
\mathsf{ex\text{-}to\text{-}seq}\;(s, ex) & \equiv & \mathsf{ex\text{-}to\text{-}seq_c}\,`(\mathsf{mk\text{-}total}\;ex)\;s
\end{array}
$$

using a continuous operation $\mathsf{ex\text{-}to\text{-}seq_c}$ which runs down the sequence component of the execution ex, transforms the actions, and adds the stutter step.

$$
\begin{array}{lcl}
\mathsf{ex\text{-}to\text{-}seq_c} & :: & (\alpha \times \sigma)\,\mathsf{sequence} \to_c \sigma \to (\sigma \times (\alpha)\,\mathsf{option} \times \sigma)\,\mathsf{sequence} \\
\mathsf{ex\text{-}to\text{-}seq_c}\,`\bot\; s & = & \bot \\
\mathsf{ex\text{-}to\text{-}seq_c}\,`\mathsf{nil}\; s & = & [(s, \mathsf{None}, s)!] \\
\mathsf{ex\text{-}to\text{-}seq_c}\,`((a,t)\hat{\ }ex)\; s & = & (s, \mathsf{Some}(a), t)\hat{\ }\mathsf{ex\text{-}to\text{-}seq_c}\,`ex\; t
\end{array}
$$

In the case of ex being partial, it is made total first, which means that the final $\bot$ is substituted by nil.

$$
\begin{array}{lcl}
\mathsf{mk\text{-}total} & :: & (\alpha)\,\mathsf{sequence} \to (\alpha)\,\mathsf{sequence} \\
\mathsf{mk\text{-}total}(s) & \equiv & \textbf{if}\ \mathsf{Partial}(s)\ \textbf{then}\ \varepsilon\, t.\, \mathsf{Finite}(t) \wedge s = t \oplus \bot \\
 & & \qquad\qquad\qquad \textbf{else}\ s
\end{array}
$$

□

Note that $\mathsf{ex\text{-}to\text{-}seq}$ cannot be defined as a continuous function, as adding a further element to a partial sequence is not even monotone. Therefore, we divided the definition into the continuous $\mathsf{ex\text{-}to\text{-}seq_c}$ and the discontinuous $\mathsf{mk\text{-}total}$. This reduces the occurring discontinuity to a generic function with the further advantage that it may be reused for other discontinuous definitions as well, e.g. for defining fair merge.

Corollary 11.3.2 (Characterization of $\mathsf{ex\text{-}to\text{-}seq}$)
Providing an equational characterization of $\mathsf{mk\text{-}total}$, it is easy to derive the desired rewrite rules for $\mathsf{ex\text{-}to\text{-}seq}$ from those for $\mathsf{ex\text{-}to\text{-}seq_c}$.

$$\begin{array}{lcl}
\mathsf{mk\text{-}total}(\bot) & = & \mathsf{nil} \\
\mathsf{mk\text{-}total}(\mathsf{nil}) & = & \mathsf{nil} \\
\mathsf{mk\text{-}total}(a\hat{\ }s) & = & a\hat{\ }(\mathsf{mk\text{-}total}\ s) \\[4pt]
\mathsf{ex\text{-}to\text{-}seq}\ (s, \bot) & = & [(s, \mathsf{None}, s)!] \\
\mathsf{ex\text{-}to\text{-}seq}\ (s, \mathsf{nil}) & = & [(s, \mathsf{None}, s)!] \\
\mathsf{ex\text{-}to\text{-}seq}\ (s, (a,t)\hat{\ }ex) & = & (s, \mathsf{Some}(a), t)\hat{\ }\mathsf{ex\text{-}to\text{-}seq}\ (t, ex)
\end{array}$$

Definition 11.3.3 (TLS – Temporal Logic of Steps)
Formulas of TLS are represented by TL formulas, whose sequence elements are transition triples extended by an optional stuttering action None. Predicates over these triples are called step predicates.

$$\begin{array}{lcl}
(\alpha, \sigma)\,\mathsf{tls\text{-}temporal} & = & (\sigma \times (\alpha)\,\mathsf{option} \times \sigma)\,\mathsf{temporal} \\
(\alpha, \sigma)\,\mathsf{step\text{-}pred} & = & (\sigma \times (\alpha)\,\mathsf{option} \times \sigma)\,\mathsf{pred}
\end{array}$$

Evaluating formulas over executions boils down to evaluating formulas over sequences using $\mathsf{ex\text{-}to\text{-}seq}$. Validity and A-validity are defined accordingly[6].

$$\begin{array}{lcl}
\models_{\mathsf{ex}} & :: & (\alpha, \sigma)\,\mathsf{execution} \to (\alpha, \sigma)\,\mathsf{tls\text{-}temporal} \to \mathsf{bool} \\
exec \models_{\mathsf{ex}} P & \equiv & (\mathsf{ex\text{-}to\text{-}seq}\ exec) \models P \\[4pt]
\models_{\mathsf{ex}}\ P & \equiv & \forall exec.\ exec \models_{\mathsf{ex}} P \\
A \models_{\mathsf{A}} P & \equiv & \forall exec \in \mathsf{executions}(A).\ exec \models_{\mathsf{ex}} P
\end{array}$$

□

Note, that for $\models_{\mathsf{ex}}$ the boolean connectives have the same pointwise meaning as for $\models$, e.g. $exec \models_{\mathsf{ex}} \neg P = exec \not\models_{\mathsf{ex}} P$. This, however, does not hold for $\models_{\mathsf{A}}$.

When talking about I/O automata it is often more convenient to use predicates on states and actions rather than step predicates, which always take the stuttering action into account. Below we present the necessary link between these predicates.

[6]Once more the symbol $\models_{\mathsf{ex}}$ is overloaded in a way not supported by Isabelle.

Definition 11.3.4 (Lifting Predicates)
The functions $\mathsf{ext_s}$ and $\mathsf{ext_a}$ lift state and action predicates to step predicates. Possibly occurring stuttering actions force the resulting step predicate to evaluate to False.

$$\begin{array}{lcl} \mathsf{ext_s} & :: & (\sigma)\,\mathsf{pred} \rightarrow (\alpha,\sigma)\,\mathsf{step\text{-}pred} \\ \mathsf{ext_s}(P) & \equiv & \lambda(s,a,t).\ P(s) \end{array}$$

$$\begin{array}{lcl} \mathsf{ext_a} & :: & (\alpha)\,\mathsf{pred} \rightarrow (\alpha,\sigma)\,\mathsf{step\text{-}pred} \\ \mathsf{ext_a}(P) & \equiv & \lambda(s,a',t).\ \mathbf{case}\ a'\ \mathbf{of} \\ & & \qquad\qquad \begin{array}{lll} & \mathsf{None} & \Rightarrow\ \mathsf{False} \\ \mid & \mathsf{Some}\ a & \Rightarrow\ P(a) \end{array} \end{array}$$

□

We use the following abbreviating syntax, which looks slightly less appealing in Isabelle.

$$\langle P\rangle_{\mathsf{s}} = \langle \mathsf{ext_s}(P)\rangle \qquad \langle P\rangle_{\mathsf{a}} = \langle \mathsf{ext_a}(P)\rangle$$

Lemma 11.3.5 (Relation between Validities)
Validity on sequences is stronger than on executions.

$$\models P \ \Rightarrow\ \models_{\mathsf{ex}} P \quad (\mathit{Val\text{-}Rel})$$

Therefore, the theorems $(ax1) - (ax3)$ from Lemma 11.2.3 carry over from $\models$ to $\models_{\mathsf{ex}}$. The same holds for the theorems (mp), (nex), and (ind).

Proof.
We merely prove $(\mathit{Val\text{-}Rel})$ which postulates that $\mathsf{ex\text{-}to\text{-}seq}(exec) \models_{\mathsf{ex}} P$ holds for every $exec$, provided that $s \models_{\mathsf{ex}} P$ holds for every s with $s \neq \bot \wedge s \neq \mathsf{nil}$. This is true as $\mathsf{ex\text{-}to\text{-}seq}$ produces only non-empty sequences. □

Note that the other direction of $(\mathit{Val\text{-}Rel})$ is not true, as not every transition sequence is an image under $\mathsf{ex\text{-}to\text{-}seq}$: there may be None elements occurring not only after the final state of non-infinite executions, or non-identical successor states.

Therefore, the completeness considerations for TL do not carry over to TLS. However, the specific form of sequences generated by $\mathsf{ex\text{-}to\text{-}seq}$ can be captured by a temporal formula for each automaton step. Having derived these step formulas, it is often sufficient to use further on only rules for $\models$ instead of $\models_{\mathsf{ex}}$. Therefore, they build some kind of interface between TL and TLS.

Lemma 11.3.6 (Interface of Step Formulas)
Pre- and postconditions for I/O automata steps imply a directly corresponding temporal formula for every action a.

$$\frac{\forall s\ t.\ P(s) \wedge s \xrightarrow{a}_A t \ \Rightarrow\ Q(t)}{A \models_{\mathsf{A}} \langle P\rangle_{\mathsf{s}} \wedge \langle \lambda x.\ x = a\rangle_{\mathsf{a}} \Rightarrow \bigcirc\langle Q\rangle_{\mathsf{s}}}$$

Proof.
Suppose $(s_1, ex) \in \mathsf{executions}(A)$. We have to show that $\mathsf{ex\text{-}to\text{-}seq}\,(s_1, ex) \models \langle P \rangle_{\mathsf{s}} \wedge \langle \lambda x.\, x = a \rangle_{\mathsf{a}} \Rightarrow \bigcirc \langle Q \rangle_{\mathsf{s}}$. If $ex = \bot$ or $ex = \mathsf{nil}$, the stuttering step added by $\mathsf{ex\text{-}to\text{-}seq}$ leads to $\mathsf{ext_a}(\lambda x.\ x = a)(\mathsf{None}) = \mathsf{False}$, which falsifies the assumption and therefore proves the entire proposition. Otherwise, $ex = (a_1, t_1)\hat{\ } ex_1$. Therefore, we may assume $P(s_1)$, and, as stuttering actions are only added for empty executions, $a_1 = a$. With $(s_1, (a_1, t_1)\hat{\ } ex_1) \in \mathsf{executions}(A)$ we get $s_1 \xrightarrow{a}_A t_1$ and therefore with the assumption of the entire proposition $Q(t_1)$. It remains to show $\mathsf{ex\text{-}to\text{-}seq}(s_1, (a, t_1)\hat{\ } ex_1) \models \bigcirc \langle Q \rangle_{\mathsf{s}}$. As $\mathsf{ex\text{-}to\text{-}seq}$ produces always identical succeeding states we get $\mathsf{ex\text{-}to\text{-}seq}(s_1, (a, t_1)\hat{\ } ex_1) = (s_1, a, t_1)\hat{\ }(t_1, \star, \star)\hat{\ }\star$, which shows the desired result immediately. □

As the proof shows, these formulas incorporate the fact, that $\mathsf{ex\text{-}to\text{-}seq}$ adds stuttering steps only at the end of finite executions and produces always identical succeeding states. Therefore, the application of these formulas yields temporal formulas, which can then be treated by standard LTL reasoning.

11.4 Live I/O Automata

Below, the I/O automata model in Isabelle (see §8) is extended to general liveness. In particular, fairness specified by TLS formulas replaces the former fairness sets.

Definition 11.4.1 (Live I/O Automata, L-Validity, Implementation)
Live I/O automata are represented by a pair of a safe[7] I/O automaton and a TLS formula.

$$(\alpha, \sigma)\,\mathsf{live\text{-}ioa} \quad = \quad (\alpha, \sigma)\,\mathsf{ioa} \times (\alpha, \sigma)\,\mathsf{tls\text{-}temporal}$$

A TLS formula P is L-valid for a live I/O automaton (A, L) if it is A-valid for A under the further assumption L:

$$(A, L) \models_{\mathsf{L}} P \quad \equiv \quad A \models_{\mathsf{A}} (L \Rightarrow P)$$

Live executions, traces, and implementations generalize the corresponding fair notions in a canonical way.

$$\begin{array}{lcl}
\mathsf{live\text{-}executions}\,(A, L) & \equiv & \{exec.\ exec \in \mathsf{executions}(A) \wedge exec \models_{\mathsf{ex}} L\} \\
\mathsf{live\text{-}traces}\,(A, L) & \equiv & \{\mathsf{mk\text{-}trace}\ A\ `(\mathsf{snd}\ ex) \mid_{ex} ex \in \mathsf{live\text{-}executions}\,(A, L)\} \\
C \preceq_L A & \equiv & \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A) \wedge \\
 & & \mathsf{live\text{-}traces}(C) \subseteq \mathsf{live\text{-}traces}(A)
\end{array}$$

□

[7] We assume implicitly that every automaton of type (α, σ) ioa features empty fairness sets in this context.

Definition 11.4.2 (Fair I/O Automata)
Weak and strong fairness is defined by the TLS formulas WF and SF.

$$\begin{array}{lcl} \mathsf{WF}, \mathsf{SF} & :: & (\alpha,\sigma)\,\mathsf{ioa} \to (\alpha)\,\mathsf{set} \to (\alpha,\sigma)\,\mathsf{tls\text{-}temporal} \\ \mathsf{WF}\ A\ acts & \equiv & \Diamond\Box\langle \mathsf{enabled}\ A\ acts\rangle_{\mathsf{s}} \Rightarrow \Box\Diamond\langle \lambda a.\ a \in acts\rangle_{\mathsf{a}} \\ \mathsf{SF}\ A\ acts & \equiv & \Box\Diamond\langle \mathsf{enabled}\ A\ acts\rangle_{\mathsf{s}} \Rightarrow \Box\Diamond\langle \lambda a.\ a \in acts\rangle_{\mathsf{a}} \end{array}$$

□

The use of the lifting function $\mathsf{ext_a}$ in the fairness formulas prevents them from being true for finite executions only because a stuttering action has been added at the last state. Therefore the semantics of fairness coincides for both finite and infinite executions with the usual one using fairness sets.

Definition 11.4.3 (Live Refinement Notions)
Refinement mappings which transfer liveness from every execution to its corresponding one are called *live refinement mappings*:

$$\begin{array}{lcl} \mathsf{is\text{-}live\text{-}refmap} & :: & (\sigma_1 \to \sigma_2) \to (\alpha,\sigma_1)\,\mathsf{live\text{-}ioa} \to (\alpha,\sigma_2)\,\mathsf{live\text{-}ioa} \to \mathsf{bool} \\ \mathsf{is\text{-}live\text{-}refmap}\,f\ (C,L)\ (A,M) & \equiv & \mathsf{is\text{-}ref\text{-}map}\,f\ C\ A \wedge \\ & & \forall exec \in \mathsf{execution}(C).\ exec \models_{\mathsf{ex}} L \\ & & \quad \Rightarrow (\mathsf{corresp}^{\mathsf{ref}}\ A\ f\ exec) \models_{\mathsf{ex}} M \end{array}$$

Forward simulations with the analogous behaviour are called *live forward simulations*:

$$\begin{array}{lcl} \mathsf{is\text{-}live\text{-}simulation} & :: & (\sigma_1 \times \sigma_2)\,\mathsf{set} \to (\alpha,\sigma_1)\,\mathsf{live\text{-}ioa} \to (\alpha,\sigma_2)\,\mathsf{live\text{-}ioa} \to \mathsf{bool} \\ \mathsf{is\text{-}live\text{-}simulation}\ R\ (C,L)\ (A,M) & \equiv & \mathsf{is\text{-}simulation}\ R\ C\ A \wedge \\ & & \forall exec \in \mathsf{execution}(C).\ exec \models_{\mathsf{ex}} L \\ & & \quad \Rightarrow (\mathsf{corresp}^{\mathsf{sim}}\ A\ R\ exec) \models_{\mathsf{ex}} M \end{array}$$

□

Theorem 11.4.4 (Soundness of Live Refinement Notions)
Live refinement mappings and live forward simulations induce live implementation, if the external actions coincide.

$$\frac{\mathsf{is\text{-}live\text{-}refmap}\,f\ (C,L)\ (A,M) \quad \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A)}{(C,L) \preceq_L (A,M)}$$

$$\frac{\mathsf{is\text{-}live\text{-}simulation}\ R\ (C,L)\ (A,M) \quad \mathsf{in}(C) = \mathsf{in}(A) \wedge \mathsf{out}(C) = \mathsf{out}(A)}{(C,L) \preceq_L (A,M)}$$

Proof.
Simple generalizations of the soundness proofs for fair refinement mappings (Theorem 9.2.10) and fair simulations (Theorem 9.3.7). □

11.5 Abstraction Rules

The entire abstraction theory of §4, apart from the extension to relations, has been definitionally embedded in Isabelle on top of TLS (§11.2 – §11.3) and the live I/O automaton model (§11.4). In the sequel we sketch this development.

First, the function c_h for any state function h is encoded as a mapping functional, then the different notions of weakenings/strengthenings and abstraction functions for the safe and live case are introduced.

Definition 11.5.1 (Functional Correspondence)

$$\begin{array}{lcl} \mathsf{corresp}^{\mathsf{abs}} & :: & (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\mathsf{execution} \to (\alpha, \sigma_2)\,\mathsf{execution} \\ \mathsf{corresp}^{\mathsf{abs}}\ h\ (s, ex) & \equiv & (h\ s, \mathsf{Map}\ (\lambda(a,t).\ (a, h\ t))\ `ex) \end{array}$$

Definition 11.5.2 (Weakening and Strengthening)

$$\begin{array}{lcl} \mathsf{aut\text{-}weak} & :: & (\alpha, \sigma_2)\,\mathsf{ioa} \to (\alpha, \sigma_1)\,\mathsf{ioa} \to (\sigma_1 \to \sigma_2) \to \mathsf{bool} \\ \mathsf{aut\text{-}weak}\ A\ C\ h & \equiv & \forall exec \in \mathsf{executions}(C).\ \mathsf{corresp}^{\mathsf{abs}}\ h\ exec \in \mathsf{executions}(A) \end{array}$$

$$\begin{array}{lcl} \mathsf{temp\text{-}strength}, \mathsf{temp\text{-}weak} & :: & (\alpha, \sigma_2)\,\mathsf{tls\text{-}temporal} \to (\alpha, \sigma_1)\,\mathsf{tls\text{-}temporal} \to (\sigma_1 \to \sigma_2) \to \mathsf{bool} \\ \mathsf{temp\text{-}strength}\ Q\ P\ h & \equiv & \forall exec.\ (\mathsf{corresp}^{\mathsf{abs}}\ h\ exec) \models_{\mathsf{ex}} Q) \Rightarrow exec \models_{\mathsf{ex}} P \\ \mathsf{temp\text{-}weak}\ Q\ P\ h & \equiv & \mathsf{temp\text{-}strength}\ (\neg Q)\ (\neg P)\ h \end{array}$$

$$\begin{array}{lcl} \mathsf{step\text{-}strength}, \mathsf{step\text{-}weak} & :: & (\alpha, \sigma_2)\,\mathsf{step\text{-}pred} \to (\alpha, \sigma_1)\,\mathsf{step\text{-}pred} \to (\sigma_1 \to \sigma_2) \to \mathsf{bool} \\ \mathsf{step\text{-}strength}\ Q\ P\ h & \equiv & \forall s\ a\ t.\ Q\ (h\ s, a, h\ t) \Rightarrow P\ (s, a, t) \\ \mathsf{step\text{-}weak}\ Q\ P\ h & \equiv & \mathsf{step\text{-}strength}\ (\neg Q)\ (\neg P)\ h \end{array}$$

Definition 11.5.3 (Abstraction Functions)

$$\begin{array}{lcl} \mathsf{is\text{-}abstraction} & :: & (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\mathsf{ioa} \to (\alpha, \sigma_2)\,\mathsf{ioa} \to \mathsf{bool} \\ \mathsf{is\text{-}abstraction}\ h\ C\ A & \equiv & (\forall s_0 \in \mathsf{starts\text{-}of}\ C.\ (h\ s_0) \in \mathsf{starts\text{-}of}\ A) \land \\ & & (\forall s\ t\ a. \mathsf{reachable}\ C\ s \land s \xrightarrow{a}_C t \\ & & \qquad \Rightarrow (h\ s) \xrightarrow{a}_A (h\ t)) \end{array}$$

$$\begin{array}{lcl} \mathsf{is\text{-}live\text{-}abstraction} & :: & (\sigma_1 \to \sigma_2) \to (\alpha, \sigma_1)\,\mathsf{live\text{-}ioa} \to (\alpha, \sigma_2)\,\mathsf{live\text{-}ioa} \to \mathsf{bool} \\ \mathsf{is\text{-}live\text{-}abstraction}\ h\ (C, M)\ (A, L) & \equiv & \mathsf{is\text{-}abstraction}\ h\ C\ A \land \mathsf{temp\text{-}weak}\ M\ L\ h \end{array}$$

Theorem 11.5.4 (Soundness of Abstractions)
Abstractions permit to reduce automaton weakenings to step weakenings ($Sound$-1), safe abstractions are sound w.r.t. safe implementation ($Sound$-2), and live abstractions are

sound w.r.t live implementation (*Sound*-3).

$$\frac{\textsf{is-abstraction}\ h\ C\ A}{\textsf{aut-weak}\ A\ C\ h}\ (Sound\text{-}1)$$

$$\frac{\textsf{is-abstraction}\ h\ C\ A \qquad \textsf{in}(C) = \textsf{in}(A) \wedge \textsf{out}(C) = \textsf{out}(A)}{C \preceq_S A}\ (Sound\text{-}2)$$

$$\frac{\textsf{is-live-abstraction}\ h\ (C, L)\ (A, M) \qquad \textsf{in}(C) = \textsf{in}(A) \wedge \textsf{out}(C) = \textsf{out}(A)}{(C, L) \preceq_L (A, M)}\ (Sound\text{-}3)$$

Proof.
See the proofs of Theorems 4.2.6, 4.2.15, and 4.2.8. □

Theorem 11.5.5 (Abstraction Rules for Live I/O Automata)
As examples we present the rules (Abs-P_2) and (Abs-A_2) for live I/O automata.

$$\frac{\begin{array}{c}\textsf{is-live-abstraction}\ h\ (C, L)\ (A, M)\\ \textsf{temp-strength}\ Q\ P\ h\\ (A, M) \models_L Q\end{array}}{(C, L) \models_L P} \qquad \frac{\begin{array}{c}\textsf{in}(C) = \textsf{in}(A) \wedge \textsf{out}(C) = \textsf{out}(A)\\ \textsf{in}(Q) = \textsf{in}(P) \wedge \textsf{out}(Q) = \textsf{out}(P)\\ \textsf{is-live-abstraction}\ h_1\ (C, L_C)\ (A, L_A)\\ \textsf{is-live-abstraction}\ h_2\ (Q, L_Q)\ (P, L_P)\\ (A, L_A) \preceq_L (Q, L_Q)\end{array}}{(C, L_C) \preceq_L (P, L_P)}$$

Proof.
See the proofs of Theorems 4.2.11 and 4.2.17. □

The additional abstraction rules (Abs-P_1), (Abs-A_1), (Abs-P_3) and (Abs-A_3) are provided as well. It should be clear how they are encoded and proved.

The following theorems represent the dual concept to Lemma 4.2.4 which allows to reduce global weakenings/strengthenings to local ones. As we use a shallow embedding, the result cannot be presented as a single theorem in terms of the syntactical formula structure. Rather we provide a bunch of structural rules which have been used to build a tactic, which automatically performs the desired reductions.

Theorem 11.5.6
The following rules hold as well, when interchanging $\mathsf{temp\text{-}strength}$ with $\mathsf{temp\text{-}weak}$.

$$\frac{\mathsf{temp\text{-}strength}\ P_1\ Q_1\ h \quad \mathsf{temp\text{-}strength}\ P_2\ Q_2\ h}{\mathsf{temp\text{-}strength}\ (P_1 \star P_2)\ (Q_1 \star Q_2)\ h}\ \star \in \{\wedge, \vee\} \qquad \frac{\mathsf{temp\text{-}weak}\ P\ Q\ h}{\mathsf{temp\text{-}strength}\ (\neg P)\ (\neg Q)\ h}$$

$$\frac{\mathsf{temp\text{-}weak}\ P_1\ Q_1\ h \quad \mathsf{temp\text{-}strength}\ P_2\ Q_2\ h}{\mathsf{temp\text{-}strength}\ (P_1 \star P_2)\ (Q_1 \star Q_2)\ h}\ \star \in \{\Rightarrow, \leadsto\} \qquad \frac{\mathsf{step\text{-}strength}\ P\ Q\ h}{\mathsf{temp\text{-}strength}\ \langle P\rangle\ \langle Q\rangle\ h}$$

$$\frac{\mathsf{temp\text{-}strength}\ P\ Q\ h}{\mathsf{temp\text{-}strength}\ (\star P)\ (\star Q)\ h}\ \star \in \{\Box, \Diamond, \bigcirc\}$$

Proof.
For $\wedge$, $\vee$, $\neg$, and $\Rightarrow$ the rules follow directly from the definition of these operators on predicates. Therefore, it is sufficient to proof the rules for Init, $\Box$, and $\bigcirc$, as the remaining operators are defined by means of these.

"Init": Using the definitions of $\mathsf{temp\text{-}strength}$, $\mathsf{step\text{-}strength}$, $\Box$, and $\models$ we have to show

$$\mathcal{C}_1: \quad Q\ (\mathsf{the}\ (\mathsf{HD}\ ‘(\mathsf{ex\text{-}to\text{-}seq}\ (\mathsf{corresp}^{\mathsf{abs}}\ h\ exec))))$$

for every execution $exec$ using the assumptions

$$\begin{aligned} &\mathcal{A}_1: \quad P\ (\mathsf{the}\ (\mathsf{HD}\ ‘(\mathsf{ex\text{-}to\text{-}seq}\ exec))) \\ &\mathcal{A}_2: \quad \forall s, a, t.\ P\ (h\ s, a, h\ t) \Rightarrow Q\ (s, a, t) \end{aligned}$$

Writing (s, ex) for $exec$ we make a case distinction on ex. If $ex = \bot \vee ex = \mathsf{nil}$, $\mathcal{A}_1$ rewrites to $P\,(h\,s, \mathsf{None}, h\,s)$, as $\mathsf{ex\text{-}to\text{-}seq}$ adds a stutter step in this case. Similarly, $\mathcal{C}_1$ rewrites to $Q\ (s, \mathsf{None}, s)$, which is true because of $\mathcal{A}_2$. Otherwise, if $ex = (a, t)\hat{}\,ex'$, $\mathcal{A}_1$ rewrites to $P\ (h\ s, \mathsf{Def}\ a, h\ t)$, and $\mathcal{C}_1$ to $Q\ (s, \mathsf{Def}\ a, t)$, which holds because of $\mathcal{A}_2$ as well.

"$\Box$": Using the definitions of $\mathsf{temp\text{-}strength}$, $\mathsf{step\text{-}strength}$, Init, and $\models$ we have to show

$$\mathcal{C}_1: \quad Q(s_3)$$

for every sequence of transitions s_3 under the assumptions

$$\begin{aligned} &\mathcal{A}_1: \quad \forall exec.\ P\ (\mathsf{ex\text{-}to\text{-}seq}\ (\mathsf{corresp}^{\mathsf{abs}}\ h\ exec)) \Rightarrow Q\ (\mathsf{ex\text{-}to\text{-}seq}\ exec) \\ &\mathcal{A}_2: \quad \forall s_2.\ s_2 \gg^+ \mathsf{ex\text{-}to\text{-}seq}\ (\mathsf{corresp}^{\mathsf{abs}}\ h\ exec) \Rightarrow P(s_2) \\ &\mathcal{A}_3: \quad s_3 \gg^+ \mathsf{ex\text{-}to\text{-}seq}(exec) \end{aligned}$$

The following auxiliary constant plays the rôle of $\mathsf{corresp}^{\mathsf{abs}}$ on sequences.

$$\begin{aligned} \mathsf{corresp}^{\mathsf{abs}}_{\mathsf{s}} \quad &:: \quad (\sigma_1 \to \sigma_2) \to (\sigma_1 \times (\alpha)\,\mathsf{option} \times \sigma_1)\,\mathsf{sequence} \\ &\qquad \to (\sigma_2 \times (\alpha)\,\mathsf{option} \times \sigma_2)\,\mathsf{sequence} \\ \mathsf{corresp}^{\mathsf{abs}}_{\mathsf{s}}\ h\ s \quad &\equiv \quad \mathsf{Map}\ (\lambda(s, a, t).\ (f\ s, a, f\ t))\ ‘s \end{aligned}$$

It has been introduced, as it is simpler to derive lemmas involving sequences instead of executions. In particular, the following lemmas are needed during the proof.

$$\begin{array}{lcll} \textsf{ex-to-seq}\ (\textsf{corresp}^{\textsf{abs}}\ h\ exec) & = & \textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ (\textsf{ex-to-seq}\ exec) & (1) \\ s \gg^{+} \textsf{ex-to-seq}(exec) & \Rightarrow & \exists exec'.\ s = \textsf{ex-to-seq}(exec') & (2) \\ s \gg^{+} t & \Rightarrow & (\textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ s) \gg^{+} (\textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ t) & (3) \end{array}$$

The first one allows to delay the construction of the corresponding execution after the transformation ex-to-seq to sequences, the second characterizes stability of this transformation under "shifting", and the last one essentially means commutation of Map and $\oplus$.

From $\mathcal{A}_3$ we get with Lemma (2) the existence of an $exec'$ with $s_3 = \textsf{ex-to-seq}(exec')$, which turns $\mathcal{A}_3$ into $\textsf{ex-to-seq}(exec') \gg^{+} \textsf{ex-to-seq}(exec)$. This in turn reduces by Lemma (3) to

$$\mathcal{A}_4\text{:}\quad \textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ (\textsf{ex-to-seq}\ exec') \gg^{+} \textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ (\textsf{ex-to-seq}\ exec)$$

We use Lemma (1) to rewrite $\mathcal{A}_2$ and apply it then to $\mathcal{A}_4$ with the instantiation $s_2 := \textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ (\textsf{ex-to-seq}\ exec')$. Therefore, we get

$$\mathcal{A}_5\text{:}\quad P\ (\textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ (\textsf{ex-to-seq}\ exec'))$$

Once more, we use Lemma (1), this time to rewrite $\mathcal{A}_1$, which we apply then to $\mathcal{A}_5$. Therefore, we get $Q\ (\textsf{ex-to-seq}\ exec')$, which equals $\mathcal{C}_1$, as $s_3 = \textsf{ex-to-seq}(exec')$.

"$\bigcirc$": Analogous to the proof for $\square$. Instead of Lemmas 2 and 3 we use the following.

$$\begin{array}{lcll} ex \neq \bot \wedge ex \neq \textsf{nil} & \Rightarrow & \exists exec'.\ \textsf{TL}\ `(\textsf{ex-to-seq}\ (s, ex)) = \textsf{ex-to-seq}(exec') & (2') \\ \textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ (\textsf{TL}\ `s) & = & \textsf{TL}\ `(\textsf{corresp}^{\textsf{abs}}_{\textsf{s}}\ h\ s) & (3') \end{array}$$

The proof is by case distinction on $\exists x\ xs.\ \textsf{TL}\ `(\textsf{ex-to-seq}\ exec) = x\hat{\ }xs$. $\square$

Note that the key idea of the proof is the functional correspondence between executions, which means that the corresponding execution is defined by a pointwise mapping — see e.g. Lemma (3′).

11.6 Conclusion and Related Work

We presented a definitional embedding of the linear-time temporal logic TLS and live I/O automata in Isabelle. Furthermore, we laid the foundation for combining Isabelle with model checkers by providing a verified abstraction theory.

The detour over a generic temporal logic turned out to be worthwhile, as therefore the proof infrastructure developed for temporal logics in [MP95, Krö87] can be reused. Furthermore, the stuttering action $\surd$ could be interpreted as a means to avoid the empty sequence in the corresponding generic logic.

TLS has been embedded in HOLCF in a shallow way, where the algebraic sequence model turned out to be surprisingly adequate for these intentions. In contrast to sequences modeled as functions on the natural numbers [Lån94, Wri92, Cho93] it allows to incorporate finite sequences and to deal with operators that remove stuttering very easily.

An important result from a methodological point of view is the following: the resulting Isabelle support for liveness proofs obeys our tailored separation of HOL and HOLCF as well. HOLCF is hidden from the user, who operates only within the simpler HOL or uses standard rules of temporal logic.

Related Work. TLA has been embedded in higher-order logic several times already (e.g. [Lån94, Wri92, Mer95]). The formalizations in [Lån94, Wri92] have already been discussed, the distinguishing feature to [Mer95] is the fact that the temporal logic is axiomatized instead of definitionally embedded. Furthermore, temporal rules like induction are more complicated than in our setting, because primed variables have to be used instead of the usual next-time operator. For the same reason TLA has not been encoded via a generic logic, although TLA does not feature predicates over single states, but over some kind of tuple, similar to TLS. Thus it would in principle be possible to embed TLA into a generic temporal logic similar to TL. However, such a generic logic would lack the next-time operator which would abandon every possibility to refer to the next state. Therefore the logic would degenerate to a modal logic, whose expressiveness is too far away from the desired TLA.

Part IV: Applications and Evaluation

Chapter 12

Case Studies

In this chapter three case studies are presented that have been carried out within the Isabelle framework for I/O automata. The aim is to evaluate the practicability and scalability of our tool environment. The studies deal with an industrial cockpit control system, an extended version of the Alternating Bit Protocol, and a simple memory manager. We demonstrate the combination of Isabelle with model checking via abstractions as well as the interactive proof of implementation relations via forward simulations. Furthermore, we elaborate methodological aspects concerning (liveness) abstractions and propose a solution for the problem of model checking unbounded buffers which are caused by the interleaving semantics of I/O automata.

12.1 Introduction

The aim of this chapter is to show the usability, practicability, and scalability of the I/O automata framework that has been developed in §8 – §11. Therefore, we perform three case studies, each of them covering different aspects of the tool set.

First, we deal with a cockpit control system from the area of avionics, which has been taken from a current industrial development [MG95]. The display manager of this control system has been verified by a combination of Isabelle and SteP in a student project [Ham98]. Safety-critical properties are expressed as TLS formulas and cover safety as well as liveness aspects. In particular, the rule for improving liveness abstractions can be used to advantage.

Second, we deal with the safety part of the Alternating Bit Protocol. Once more we use abstraction in order to reduce the size of unbounded channels, but this time the property to be verified is expressed as an I/O automaton. Thus, one of the abstraction rules for down-scaling implementation relations is used. Furthermore, we have to address the problem of an unbounded sender buffer, which is caused by the input-enabled interleaving semantics of I/O automata. The solution of this problem demonstrates impressively the usefulness of compositionality when dealing with hierarchies of implementation layers.

Finally, we deal with a very simple example of a memory manager. The aim is to verify an implementation relation, which cannot be proved by a refinement mapping, but needs a forward simulation, i.e. the additional power of history variables.

Most of the case studies deal with abstraction, as this allows us to evaluate not only the practicability of our tool environment but the effectiveness of our abstraction theory as well. Note that the first case study has an industrial background, whereas the remaining studies represent rather trivial standard examples which are employed to illustrate certain specific aspects of the approach.

Notation. In this chapter we use a postfix dot notation to represent selector functions for state components. Similarly, dots are employed to select components of an automaton network. Thus, if s is a state of some system incorporating *Buf* as a component, then $s.Buf.queue$ denotes the queue of *Buf*. However, the representation by tuples and the dot notation are only used in the presentation of proofs, whereas specifications are described in the usual precondition/effect style. Because of this higher description level we do not use the sans serif font for constants, although all case studies in this chapter have been performed within Isabelle.

12.2 Cockpit Control System: Alarm Management

In this section we deal with an industrial case study that describes the display manager of a cockpit alarm system. The study presents a small fragment of an actual helicopter development performed by the company ESG [MG95]. It originated in the KorSys project [BDE+97] where several companies provided industrial case studies in order to evaluate the scalability of formal methods. The verification described in the following has been carried out in a two month student project [Ham98] and represents a full size version of the Examples 4.2.10 – 4.2.14. The idea is to combine Isabelle with the STeP model checker via the abstraction of TLS properties (proof obligation $(C, F_C) \models P$). In particular, the improvement of liveness abstractions is considered.

The aim of the cockpit control system is to control and monitor the operation of the non-avionics components like bord electronics, fuel and engine management. Safety-critical incidents and failures concerning the physical devices are notified by a series of different alarm messages. The pilot has to observe those messages and react immediately and

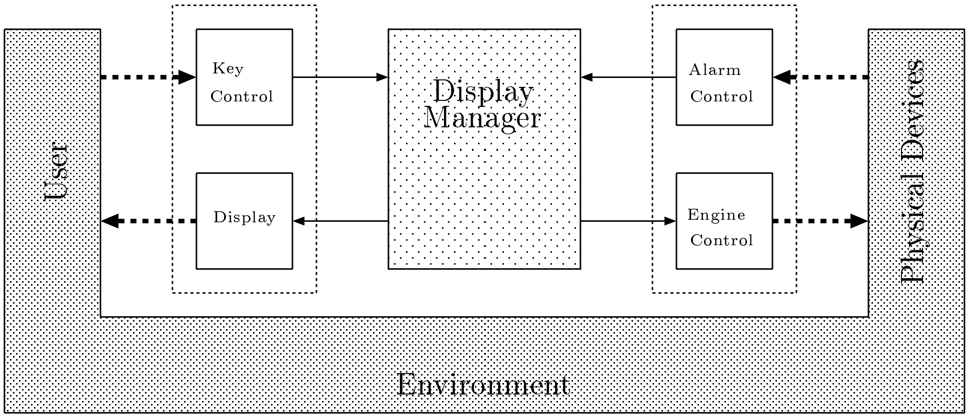

Figure 12.1: Architecture of the Cockpit Control System

properly. The architecture of the system is depicted in Fig. 12.1. We consider only the especially critical display manager of the control system. The user interface components (key manager, display) as well as the interface to the physical devices (engine, failure sensors) are not taken into account. The display manager has to store incoming alarm messages, display them immediately via a small alarm notice, and enable the pilot to choose between different pages of further information depending on the actual flight mode. There are 16 alarms with different priority, which have to be stored in the order they arrive. Furthermore, acknowledged alarms are not deleted immediately, but only after the pilot leaves a specific information mode. The complexity of the alarms made it impossible to handle the system completely by model checking. An attempt using Spin in [PS97] could treat not more than 4 alarms. However, we will be able to prove some safety-critical properties about the alarm PonR (Point of no Return) using abstraction techniques without reducing the overall number of alarms.

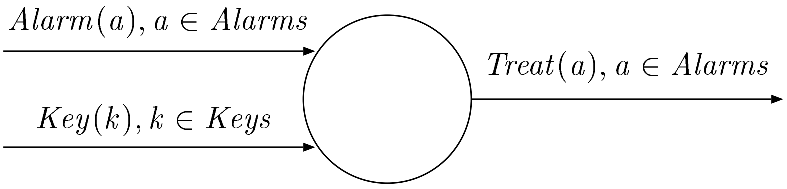

Figure 12.2: Dataflow of the Cockpit Alarm System

12.2.1 The Concrete System

In the following we describe the display manager of the cockpit alarm system in I/O automaton notation. First, we introduce the different flight modes, the keys representing the possible interactions by the pilot, and the different alarms (cf. Fig. 12.6). The state

$$
\begin{array}{l}
\textbf{output } \mathit{Treat}(a),\, a \in \mathit{Alarms}\\
\quad \textbf{pre}\text{: } \mathit{state} \neq \mathit{cPFD} \wedge \mathit{state} \neq \mathit{cNAV} \wedge\\
\qquad\quad a = (\textbf{if } (\mathit{alarmlist} \neq [\,])\ \textbf{then } \mathit{hd}(\mathit{alarmlist})\\
\qquad\qquad\qquad \textbf{else if } (\mathit{warninglist} \neq [\,])\ \textbf{then } \mathit{hd}(\mathit{warninglist})\\
\qquad\qquad\qquad\qquad \textbf{else } \mathit{None})\\
\quad \textbf{post}\text{: } \mathit{info} := \mathit{None},\\
\qquad\quad \mathit{state} := \mathit{cAC\text{-}DO};\\
\qquad\quad \mathit{alarmlist} := \textbf{if } (\mathit{alarmlist} = [\,])\ \textbf{then } [\,]\ \textbf{else } \mathit{tl}(\mathit{alarmlist}),\\
\qquad\quad \mathit{warninglist} := \textbf{if } (\mathit{alarmlist} = [\,] \wedge \mathit{warninglist} \neq [\,])\ \textbf{then } \mathit{tl}(\mathit{warninglist})\\
\qquad\qquad\qquad\qquad\qquad \textbf{else } \mathit{warninglist},\\
\qquad\quad \mathit{acklist} := \textbf{if } (\mathit{alarmlist} \neq [\,])\ \textbf{then } \mathit{acklist}@[\mathit{hd}(\mathit{alarmlist})]\\
\qquad\qquad\qquad\quad \textbf{else if } (\mathit{warninglist} \neq [\,])\ \textbf{then } \mathit{acklist}@[\mathit{hd}(\mathit{warninglist})]\\
\qquad\qquad\qquad\qquad \textbf{else if } (\mathit{acklist} \neq [\,])\\
\qquad\qquad\qquad\qquad\quad \textbf{then } \mathit{tl}(\mathit{acklist})@[\mathit{hd}(\mathit{acklist})]\\
\qquad\qquad\qquad\qquad\quad \textbf{else } [\,],\\
\qquad\quad \mathit{display} := \textbf{if } (\mathit{alarmlist} \neq [\,])\ \textbf{then } \mathit{hd}(\mathit{alarmlist})\\
\qquad\qquad\qquad\quad \textbf{else if } (\mathit{warninglist} \neq [\,])\ \textbf{then } \mathit{hd}(\mathit{warninglist})\\
\qquad\qquad\qquad\qquad \textbf{else if } (\mathit{acklist} \neq [\,])\ \textbf{then } \mathit{hd}(\mathit{acklist})\\
\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{None}
\end{array}
$$

Figure 12.3: Action *Treat* of $\mathit{Cockpit}_C$

cPFD characterizes the normal flight mode, *cNAV* the navigation mode, and *cAC* the alarm management mode. The remaining states are substates of *cAC* and determine whether the pilot deals with engine or fuel alarms or operates specific lists featuring check informations or instructions.

The alarms are divided into fuel alarms, engine alarms, and warnings by the functions *isFuelAlarm*, *isEngineAlarm*, and *isWarning*, where warnings have lower priority.

Key presses cause state changes, which are specified by means of the function *h-state* given as follows:

$$
\begin{array}{lcl}
\mathit{h\text{-}state}(\mathit{PFD}) & = & \mathit{cPFD}\\
\mathit{h\text{-}state}(\mathit{NAV}) & = & \mathit{cNAV}\\
\mathit{h\text{-}state}(\mathit{AC}) & = & \mathit{cAC}\\
\mathit{h\text{-}state}(F1) & = & \mathit{cAC\text{-}ENG}\\
\mathit{h\text{-}state}(F3) & = & \mathit{cAC\text{-}FUE}\\
\mathit{h\text{-}state}(F8) & = & \mathit{cAC\text{-}CHK}\\
\mathit{h\text{-}state}(k) & = & \mathit{cAC\text{-}oth}, \text{for all other } k
\end{array}
$$

The states of the automaton $\mathit{Cockpit}_C$ modeling the display manager are given as follows:

$$
\begin{array}{l}
\textbf{input } \mathit{Alarm}(a),\, a \in \mathit{Alarms} \\
\quad \textbf{post}\text{: } \mathit{info} := \textbf{if } (a = \mathit{None}) \textbf{ then } \mathit{info} \textbf{ else } a, \\
\qquad\quad \mathit{state} := \textbf{if } (a = \mathit{None} \vee \mathit{isWarning}(a)) \textbf{ then } \mathit{state} \\
\qquad\qquad\qquad \textbf{else if } \mathit{isFuelAlarm}(a) \textbf{ then } \mathit{cAC\text{-}FUE} \\
\qquad\qquad\qquad\quad \textbf{else if } \mathit{isEngineAlarm}(a) \textbf{ then } \mathit{cAC\text{-}ENG} \\
\qquad\qquad\qquad\qquad \textbf{else } \mathit{cAC\text{-}oth}, \\
\qquad\quad \mathit{alarmlist} := \textbf{if } (a = \mathit{None} \vee \mathit{isWarning}(a)) \textbf{ then } \mathit{alarmlist} \\
\qquad\qquad\qquad\quad \textbf{else } a : [x \in \mathit{alarmlist}.x \neq a], \\
\qquad\quad \mathit{warninglist} := \textbf{if } (a = \mathit{None}) \textbf{ then } \mathit{warninglist} \\
\qquad\qquad\qquad\quad \textbf{else if } \mathit{isWarning}(a) \textbf{ then } a : [x \in \mathit{warninglist}.x \neq a] \\
\qquad\qquad\qquad\qquad \textbf{else } \mathit{warninglist}, \\
\qquad\quad \mathit{acklist} := \textbf{if } (a = \mathit{None}) \textbf{ then } \mathit{acklist} \textbf{ else } [x \in \mathit{acklist}.x \neq a], \\
\qquad\quad \mathit{display} := \textbf{if } (a = \mathit{None} \vee \mathit{isWarning}(a)) \textbf{ then } \mathit{display} \textbf{ else } \mathit{None}
\end{array}
$$

Figure 12.4: Action *Alarm* of $\mathit{Cockpit}_C$

Field	Type	Initially	Explanation
state:	*States*	*cPFD*	flight mode
info:	*Alarms*	*None*	content of the immediate alarm notice
display:	*Alarms*	*None*	content of detailed information pages
alarmlist:	(*Alarms*) *list*	[ ]	list of not acknowledged fuel and engine alarms
warninglist:	(*Alarms*) *list*	[ ]	list of not acknowledged warnings
acklist:	(*Alarms*) *list*	[ ]	list of acknowledged alarms and warnings

Note that the number of states according to the employed datatypes in this automaton is $8 \times 16 \times 16 \times 16! \times 16! \times 16! \approx 10^{43}$. Although not all of these states are reachable, this is the reason why model checking is infeasible.

The dataflow of the system is depicted in Fig. 12.2. The input action *Alarm* has to store incoming alarms in one of the alarm lists according to their priority. Possibly older occurrences of new alarms in the stack are removed. Furthermore, the associated alarm notice is displayed in the cockpit. The input action *Key* allows the pilot to change between different flight and information modi. Finally, the output action *Treat* allows to handle and acknowledge stored alarms. It is only possible to treat the most recent alarm with highest priority. Treated alarms are displayed via the detailed information pages and then shifted to *acklist*. In detail, the transitions are given in Fig. 12.5–12.3. The precondition/effect style describing state sets by predicates allows for a very succinct presentation of the automaton. Therefore, it may often be favorable to graphical descriptions.

$$
\begin{array}{l}
\textbf{input } Key(k), k \in Keys \\
\quad \textbf{post}: info := \textbf{if } (k \neq F9 \wedge state = cAC\text{-}DO) \textbf{ then } None \textbf{ else } info, \\
\qquad\quad state := \textbf{if } (k = F9 \vee ((state = cPFD \vee state = cNAV) \wedge \\
\qquad\qquad\qquad k \neq PFD \wedge k \neq NAV \wedge k \neq AC)) \\
\qquad\qquad\qquad \textbf{then } state \\
\qquad\qquad\qquad \textbf{else } h\text{-}state(k), \\
\qquad\quad alarmlist := alarmlist, \\
\qquad\quad warninglist := warninglist, \\
\qquad\quad acklist := \textbf{if } (k \neq F9 \wedge state = cAC\text{-}DO) \textbf{ then } [\,] \textbf{ else } acklist, \\
\qquad\quad display := \textbf{if } (k \neq F9 \wedge state = cAC\text{-}DO) \textbf{ then } None \textbf{ else } display
\end{array}
$$

Figure 12.5: Action *Key* of $Cockpit_C$

$$
\begin{array}{lcl}
States & = & \{cPFD, cNAV, cAC, cAC\text{-}ENG, cAC\text{-}FUE, cAC\text{-}CHK, cAC\text{-}DO, cAC\text{-}oth\} \\
Keys & = & \{PFD, NAV, AC, F1, F2, F3, F4, F5, F6, F7, F8, F9\} \\
Alarms & = & \{None, PonR, FuelNG1, FuelNG2 \ldots\}, \text{ (altogether 16 elements)}
\end{array}
$$

Figure 12.6: State Information for $Cockpit_C$

12.2.2 The Abstract System

In this study we are only interested in properties about the crucial alarm *PonR* (Point of no Return). Therefore, the key idea of the abstraction is to neglect the display and the lists *warninglist* and *acklist* completely and reduce the list *alarmlist* to a single boolean variable *PonR-in* stating whether *PonR* is stored or not. The components *state* and *info* are retained. Thus, the state space of the abstract automaton $Cockpit_A$ looks as follows:

Field	Type	Initially	Explanation
state:	*States*	*cPFD*	flight mode
info:	*Alarms*	*None*	content of the immediate alarm notice
PonR-in:	*bool*	false	flag stating whether *PonR* is stored or not

Obviously, the desired abstraction function is defined as

$$h(s) \equiv (s.state, s.info, PonR \in s.alarmlist)$$

The concrete transitions of $Cockpit_A$ are given in Fig. 12.7.

output $Treat(a), a \in Alarms$
  pre: **if** $(a = None)$ **then** $PonR\text{-}in = false$
  post: $info := None,$
    $PonR\text{-}in :=$ **if** $(a = PonR)$ **then** $false$ **else** $true,$
    $state := cAC\text{-}DO$

---

input $Alarm(a), a \in Alarms$
  post: $info :=$ **if** $(a = None)$**then** $info$**else** $a,$
    $PonR\text{-}in :=$ **if** $(a = PonR)$ **then** $True$ **else** $PonR\text{-}in,$
    $state :=$ **if** $(a = None \vee isWarning(a))$ **then** $state$
      else if $isFuelAlarm(a)$ **then** $cAC\text{-}FUE$
        else if $isEngineAlarm(a)$ **then** $cAC\text{-}ENG$
          else $cAC\text{-}oth$

---

input $Key(k), k \in Keys$
  post: $info :=$ **if** $(k \neq F9 \wedge state = cAC\text{-}DO)$ **then** $None$ **else** $info,$
    $PonR\text{-}in := PonR\text{-}in,$
    $state :=$ **if** $(k = F9 \vee ((state = cPFD \vee state = cNAV) \wedge$
      $k \neq PFD \wedge k \neq NAV \wedge k \neq AC))$
      then $state$
      else $h\text{-}state(k)$

Figure 12.7: Actions of $Cockpit_A$

12.2.3 Verification of Safety-Critical Properties

Our aim is to verify the following five properties about the alarm $PonR$, where we use the abbreviation act for $a = act$ in TLS formulas.

$$\begin{aligned}
P_1 &\equiv \Box(Alarm(PonR) \rightarrow \bigcirc(info = PonR \wedge PonR \in alarmlist))\\
P_2 &\equiv \Box((PonR \notin alarmlist \wedge \neg Alarm(PonR)) \rightarrow \bigcirc(PonR \notin alarmlist))\\
P_3 &\equiv \Box(Treat(PonR) \rightarrow \bigcirc(PonR \notin alarmlist))\\
P_4 &\equiv \Box((PonR \in alarmlist \wedge \neg Treat(PonR)) \rightarrow \bigcirc(PonR \notin alarmlist))
\end{aligned}$$

$$\begin{aligned}
P_5 \equiv\ & \Box(\Diamond\Box\neg(\exists a_1.Alarm(a_1) \wedge a_1 \neq None) \wedge \Box\Diamond(\exists a_1.Treat(a_1)))\\
& \rightarrow \Diamond PonR \notin alarmlist)
\end{aligned}$$

The formulas P_1 and P_2 express that $PonR$ is immediately stored upon and only upon the $Alarm(PonR)$ action. Furthermore, P_1 states that the alarm notice $info$ is set correctly. P_3 and P_4 express that $PonR$ is removed from the alarm list upon and only upon the $Treat(PonR)$ action. Finally, P_5 expresses that $PonR$ is eventually removed from the alarm list, provided that from some point on no further (nontrivial) alarm arrives and infinitely often some arbitrary *Treat* action is performed.

We prove all these properties using the same abstraction function h. The safety properties P_1 – P_4 are proved using rule $(Abs\text{-}P_1)$ of §4.2.2, the liveness property P_5 using rule $(Abs\text{-}P_3)$. Thus, we first have to show $Cockpit_C \leq_{AF} Cockpit_A$ via h. This represents the main interactive part of the abstraction proof, but consists of only a handful invariants. Then we define P_1^+ – P_5^+, which represent the corresponding properties to P_1 – P_5 in the abstract system.

$$
\begin{array}{lcl}
P_1^+ & \equiv & \Box(Alarm(PonR) \rightarrow \bigcirc(info = PonR \wedge PonR\text{-}in)) \\
P_2^+ & \equiv & \Box((\neg PonR\text{-}in \wedge \neg Alarm(PonR)) \rightarrow \bigcirc\neg PonR\text{-}in) \\
P_3^+ & \equiv & \Box(Treat(PonR) \rightarrow \bigcirc\neg PonR\text{-}in) \\
P_4^+ & \equiv & \Box((PonR\text{-}in \wedge \neg Treat(PonR)) \rightarrow \bigcirc\neg PonR\text{-}in) \\
P_5^+ & \equiv & \Box(\Diamond\Box\neg(\exists a_1. Alarm(a_1) \wedge a_1 \neq None) \wedge \Box\Diamond(\exists a_1. Treat(a_1))) \rightarrow \Diamond\neg PonR\text{-}in)
\end{array}
$$

As the formulas P_i^+ merely interchange the atomic formula $PonR \in alarmlist$ in P_i by $PonR\text{-}in$, Isabelle proves automatically that the P_i^+ are temporal strengthenings of the P_i. Thus, for $i = 1, .., 4$ we easily get the desired $Cockpit_C \models P_i$ by checking $Cockpit_A \models P_i^+$ in STeP.

Concerning the liveness property P_5, things are not that simple, as $Cockpit_A \models P_5^+$ does not hold. This is due to specific loops introduced in $Cockpit_A$ (see Example 4.2.14 for a deeper discussion). Thus, we may either modify the abstraction function h or strengthen the abstract system with an additional assumption H_1. Our experience in this study suggests to rather maintain the abstraction function from the safety part, as this saves us from redoing the abstraction proof $Cockpit_C \leq_{AF} Cockpit_A$. Furthermore, it seems to require less intuition to add specific further assumptions than to invent a new abstraction function. Therefore, we easily prove the additional assumption

$$H_1 \equiv \Box\Diamond(\exists a_1.\ Treat(a_1) \wedge a \neq None) \Rightarrow \Box\Diamond(\exists a_1.\ Alarm(a_1) \wedge a = None)$$

in the concrete system. Taking its corresponding abstract property H_1^+ as an assumption in the abstract system, it is easy to verify the desired property with the STeP model checker.

12.3 Alternating Bit Protocol

In this section we will use the Alternating Bit Protocol to illustrate the abstraction of implementation relations (proof obligation $(C, F_C) \preceq_S (A, F_A)$). Concretely, we will employ the abstraction rule $(Abs\text{-}A_1)$ of §4.2.3. Furthermore, we will demonstrate the usefulness of different implementation layers and compositionality in the development process. We restrict us to safety, as liveness has already been extensively studied by means of the previous case study.

The Alternating Bit Protocol (ABP) [BSW69] is designed to ensure that messages are delivered in order, from a sender to a receiver, in the presence of channels that can lose and duplicate messages. We employ a particular implementation of the Alternating Bit Protocol, namely one using unbounded channels. This is in contrast to pure model checking approaches where the channels are always of a fixed capacity (usually 1). The key idea of this exercise is the fact that channels may lose and duplicate, but not reorder messages. Thus is is possible to "compactify" channels without altering their behaviour by collapsing all adjacent identical messages. This is what the abstraction function will do.

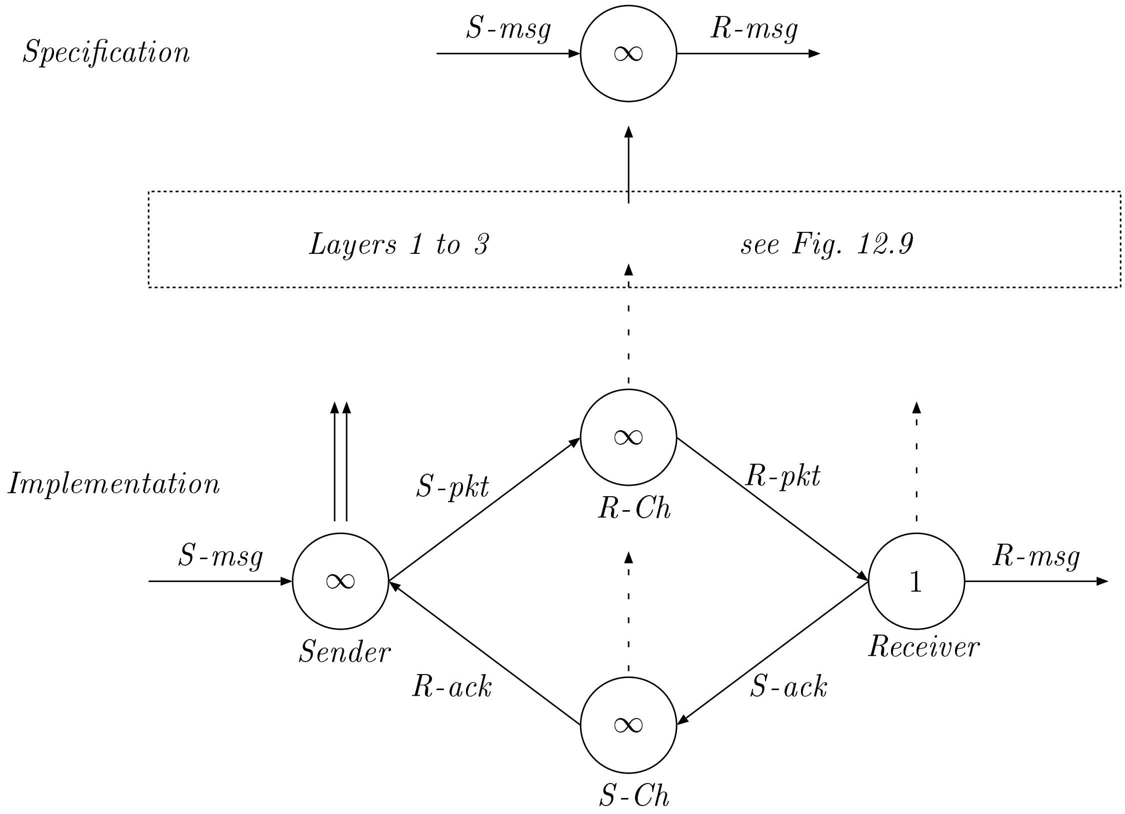

Figure 12.8: Dataflow of the ABP: Specification and Implementation. Circles represent I/O automata, numbers give "cardinality" of datastructures, single arrows denote weak refinement mappings (and data flow), double arrows denote refinement mappings, and dotted arrows indicate identity.

12.3.1 System Description

First, we describe the I/O automata that represent the desired behaviour for the communication protocol and its realization (cf. Fig. 12.8).

Specification. The FIFO-communication of the ABP can be specified by a simple queue, an I/O automaton *Spec*. The state of *Spec* is a message queue q, initially empty, modeled with the type $(\mu)\mathit{list}$, where the parameter μ represents the message type. The actions are $S\text{-}\mathit{msg}$, putting a message at the end of of the queue, and $R\text{-}\mathit{msg}$, taking a message from the head of the queue.

input $S\text{-}\mathit{msg}(m)$	**output** $R\text{-}\mathit{msg}(m)$
post: $q := q@[m]$	**pre**: $q = m : \mathit{rst}$
	post: $q := \mathit{rst}$

Implementation. The implementation is a parallel composition of 4 processes, a *Sender*, a *Receiver*, and proprietary channels $S\text{-}\mathit{Ch}$ and $R\text{-}\mathit{Ch}$ for both. The “dataflow” in the system is depicted in Fig. 12.8. Messages are transmitted from the *Sender* to the *Receiver* together with a single header bit as packets of type $\mathit{bool} \times \mu$.

The *Sender* receives messages by the action $S\text{-}\mathit{msg}$ and stores them in the sender queue. The head of this queue is repeatedly sent to $S\text{-}\mathit{Ch}$, until the corresponding header bit returns as acknowlegdement via $R\text{-}\mathit{Ch}$. Then it is removed from the queue and the next message is sent with the inverted header bit. The *Receiver* stores at most one message and passes it on via $R\text{-}\mathit{msg}$. Notice that we use unbounded buffers for the channels $S\text{-}\mathit{Ch}$ and $R\text{-}\mathit{Ch}$ and the *Sender* (see Fig. 12.8, we use ∞ to denote unbounded queues).

Sender and Receiver. The states of the *Sender* and the *Receiver* are given as follows:

	Field	Type	Initially
Sender:	q:	$(\mu)\mathit{list}$	$[\,]$
	header:	bool	true

	Field	Type	Initially
Receiver:	$\mathit{message}$:	$(\mu)\mathit{option}$	None
	header:	bool	false

The *Sender* makes the following transitions:

input $S\text{-}\mathit{msg}(m)$	**output** $S\text{-}\mathit{pkt}(b, m)$	**input** $R\text{-}\mathit{ack}(b)$
post: $q := q@[m]$	**pre**: $q = m : \mathit{rst} \wedge$	**post**:
	$\mathit{header} = b$	**if** $b = \mathit{header}$
		then $q := \mathit{tl}(q); \mathit{header} := \neg\mathit{header}$

The *Receiver* makes the following transitions:

output $R\text{-}\mathit{msg}(m)$
pre: $\mathit{message} = \mathit{Some}(m)$
post: $\mathit{message} := \mathit{None}$

input $R\text{-}pkt(b, m)$
  post:
    if $b \neq header \wedge message = None$
    then $message := Some(m); header := \neg header$

---

output $S\text{-}ack(b)$
  pre: $b = header$

Note that *R-pkt* does not change the state unless $message = None$. This ensures that the *Receiver* has passed the last message on via *R-msg* before accepting a new one.

The Channels. The channels, *R-Ch* and *S-Ch*, have very similar functionality. Therefore they can be designed as instances of a generic channel *Ch*, which has a single state component $q : (\alpha)list$, initially empty, that makes the following transitions:

input $S(a)$ **post**: $q := q@[a]$ **or** $q := q$	**output** $R(a)$ **pre**: $q \neq [\,] \wedge a = hd(q)$ **post**: $q := tl(q)$ **or** $q := q$

The optional possibility to cause no effect for both actions S and R models for the input action S the possibility of losing messages, and for the output action R the possibility of duplicating messages. The concrete channels *S-Ch* and *R-Ch* are obtained from the generic channel *Ch* by renaming S and R appropriately (see the dataflow diagram in Fig. 12.8).

12.3.2 Abstraction of the Sender Queue

What we are aiming at is a finite-state description of the Alternating Bit Protocol that represents an abstraction of the implementation described above. To achieve this, we have to address three problems:

1. The message alphabet has to be finite.
2. The sender queue has to be bounded.
3. The channel queues have to be bounded.

The first requirement deals with data abstraction as opposed to state abstraction, which is not covered by our abstraction theory, as a notion of interface refinement does not exist for I/O automata (cf. §4.5). Other formalisms, however, address this kind of abstraction. Wolper, for example, develops a notion of *data independence*, which allows to reduce an infinite data domain to a small finite one [Wol86]. In [ACW90] and [Sab88] this method is applied to the Alternating Bit Protocol. There, only three different message values are needed to verify the protocol's functional correctness. Although we did not investigate how to transfer the theory of Wolper to our setting in general, we may transfer this particular result for the Alternating Bit Protocol. The necessary data independence is easily checked: the transitions of all occurring I/O automata are independent of the value of messages being transmitted. Thus, we will analogously restrict the model checking algorithm used later on to deal with only three different message values.

A formal treatment of data-abstraction in Isabelle/HOL would need a modification of the way we model data. Currently the diversity of data is modeled by polymorphic types[1]. But since types are a meta-level notion and cannot be talked about (e.g. quantified) in HOL, even formalizing data independence seems to be impossible. Using object-level sets instead of polymorphism would cure this problem but is likely to complicate the theory.

In this section we deal with the second requirement: the boundedness of the sender queue. Due to the inherent input enabledness of I/O automata we are not allowed to simply restrict the number of input actions of the sender in order to bound its buffer. Thus, we face a problem which applies more generally: aiming at model checking, how should we deal with unbounded buffers, which are part of the implementation because of the interleaving semantics and the input enabledness of I/O automata?

We propose the following general solution. As we cannot restrict the environment directly, the needed assumption about the environment is embodied into a further automaton *Env*, which plays the rôle of this particular part of the environment. A limited number of input actions can then be achieved by a handshake protocol with *Env*. Compositionality allows us to exclude the *Env* component from model checking. Thus, the usually needed unbounded buffer is circumvented.

In the context of our protocol this means that the *Sender* is replaced by the parallel composition $Env \parallel Sender1$, where *Env* represents an unbounded queue and *Sender1* a one-element buffer (cf. Fig. 12.9 and Fig. 12.8). *Env* passes a message on to *Sender1* (via *S-pass*) only if it is requested to do so by the explicit action *Next* issued by *Sender1*. Therefore *Sender1* is bounded to at most one element.

Formally, *Env* is modeled by the pair

Field	Type	Initially
q:	$(\mu)list$	$[\,]$
$send\text{-}next$:	$bool$	$true$

and performs the following transitions.

input *Next* **post**: $send\text{-}next := true$	**input** $S\text{-}msg(m)$ **post**: $q := q @ [m]$	**output** $S\text{-}pass(m)$ **pre**: $q = m : rst \wedge$ $send\text{-}next = true$ **post**: $q := rst$

[1]It is not true that a polymorphic IOA is automatically data independent: HOL-formulae may contain the polymorphic equality "=" which destroys data independence.

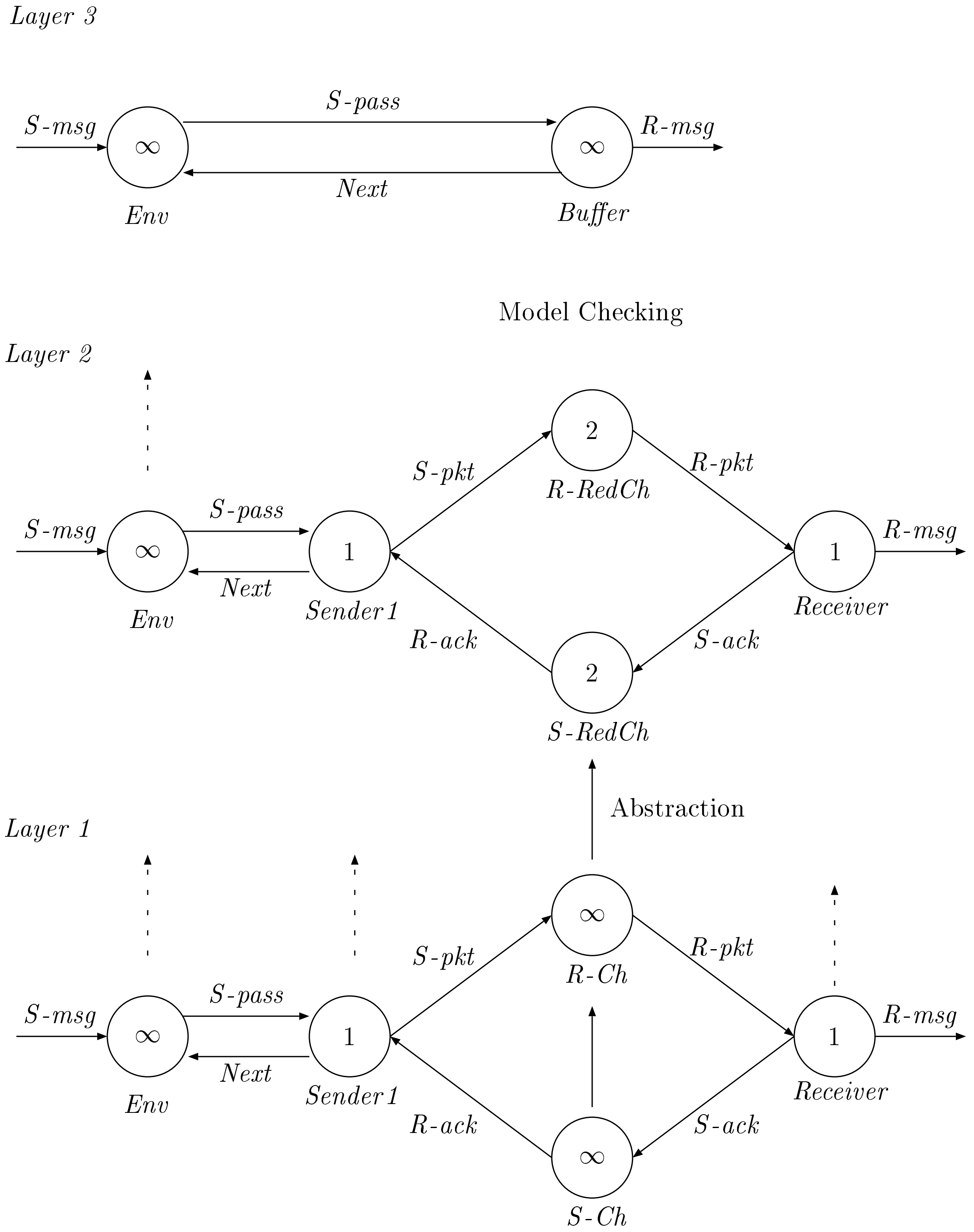

Figure 12.9: Dataflow of the ABP: Abstraction and Model Checking

The process *Sender1* is modeled by the pair

Field	Type	Initially
$message$:	$(\mu)\,option$	$None$
$header$:	$bool$	$true$

and makes the following transitions:

output $Next$ **pre**: $message = None$	**input** $S\text{-}pass(m)$ **post**: $message := Some(m)$
output $S\text{-}pkt(b, m)$ **pre**: $message = Some(m) \wedge$ $b = header$	**input** $R\text{-}ack(b)$ **post**: **if** $b = header$ **then** $message := None;\ header := \neg header$

In order to show the desired implementation relation between the implementation and layer 1 of Fig. 12.9 we only have to establish $Sender \preceq_S (Env \parallel Sender1)$ because of the compositionality of our automaton model. This is easily proved in Isabelle using a refinement mapping. Note, however, that a weak refinement mapping would not suffice as *Next* and *S-pass* are internal messages, and weak refinement mappings do not permit internal actions in the abstract system.

As the specification does not consider the component *Env*, we introduce a further layer by dividing the specification into two automata: *Env* and an additional *Buffer* (layer 3 in Fig. 12.9). The idea is to replace one unbounded buffer by two unbounded buffers which interact by a handshake-protocol. Formally, the component *Buffer* consists of a queue q, initially empty, which makes the following transitions:

output $Next$ **pre**: $true$	**input** $S\text{-}pass(m)$ **post**: $q := q@[m]$	**output** $R\text{-}msg(m)$ **pre**: $q = m : rst$ **post**: $q := rst$

It is easily established by a weak refinement mapping, that layer 3 implements the specification.

12.3.3 Abstraction of the Channel Queues

Thus, it remains to show, that layer 1 implements layer 3. We will do this by a combination of model checking and abstraction using rule $(Abs\text{-}A_1)$ of §4.2.3. Abstraction is employed to reduce the size of the channels, which are still unbounded.

Once more, we can take advantage of the compositionality of I/O automata by proving this abstraction only for the channels. The other components of layer 1 remain unchanged. The idea of the channel abstraction is based on the observation that at most two different messages are held in each channel. This is easily explained: each message is repeatedly sent to *S-Ch*, until the corresponding acknowledgment arrives. Once we switch to the next message, *S-Ch* can only contain copies of the previous message. Hence, *S-Ch*'s queue is always of the form old^*new^*. The same is true for *R-Ch*. Thus, if all adjacent identical messages are merged, the channels have size at most 2.

We accomplish this merge of adjacent identical messages in a generic way for the channel *Ch*, as the process is the same for both concrete channels *S-Ch* and *R-Ch*. A compactified channel *RedCh* is obtained from *Ch* if new messages are only added provided they differ from the last one added. Thus *RedCh* is identical to *Ch* except for the action S:

$$\begin{array}{l}
\textbf{input } S(a) \\
\quad \textbf{post}\text{: } \textbf{if } a \neq hd(reverse(q)) \vee q = [\,] \\
\quad\quad\quad \textbf{then } q := q@[a] \textbf{ or } q := q
\end{array}$$

The associated abstraction function h that merges adjacent identical messages is given as follows:

$$\begin{array}{lcl}
h([\,]) & = & [\,] \\
h(x : xs) & = & \textbf{case } xs \textbf{ of} \\
 & & \quad [\,] \;\Rightarrow\; [x] \\
 & & \quad y : ys \;\Rightarrow\; \textbf{if } x = y \textbf{ then } h(xs) \textbf{ else } x : h(xs)
\end{array}$$

Thus, to ensure the correctness of the abstraction, we have to show the proof obligation *is-abstraction h Ch RedCh*. The proof in Isabelle is rather straightforward, no invariant is needed.

In order to transfer this result to the instantiated channels, we rename *RedCh* and obtain the collapsed versions of *R-Ch* and *S-Ch*, called *R-RedCh* and *S-RedCh*. Using a theorem that guarantees that abstractions are closed under renaming, we get the desired abstraction results also for the concrete channels *S-Ch*, *R-Ch* and their collapsed versions *S-RedCh* and *R-RedCh*.

12.3.4 Model Checking

Finally, it remains to show for the model checker that layer 2 without the *Env* component, i.e. the system with compactified channels and bounded sender queue, implements the buffer of layer 3.

For this task we cannot use any of the standard model checkers available. In particular, we cannot take advantage of the tailored translation to the μ-calculus model checker μcke provided in §4.4. The reason is that the abstraction of the channels yields a system that is not explicitly finite, as the queues of the channels are still modeled by (unbounded) lists. Finiteness is only implied by the context, i.e. the behaviour of the protocol. It is merely our intuition about the protocol which guarantees that at any one time there are at most two different messages on each channel. Thus, the model checker which should confirm this intuition has to be able to handle unbounded datatypes as well.

Thus, we wrote a simple model checker in ML which performs full state space exploration. Concretely, given a function f between two (implicitly) finite state systems, the model checker tests for every reachable transition that f is indeed a weak refinement mapping. In our case, the function f is given by

$$\begin{array}{rcl} f(s) & \equiv & l(s.Receiver.message) \mathbin{@} \mathit{if}\ s.Receiver.header = s.Sender.header \\ & & \qquad\qquad\qquad\qquad\quad \mathit{then}\ l(s.Sender.message) \\ & & \qquad\qquad\qquad\qquad\quad \mathit{else}\ tl(l(s.Sender.message)) \end{array}$$

where $l : (\alpha)option \to (\alpha)list$ is defined by $l(Some(x)) = [x]$ and $l(None) = [\,]$.

The successful termination of the model checker tells us that the compactified channels represent indeed a finite system in the context of the ABP. Furthermore, it establishes the desired implementation relation between layer 2 and layer 3, which finishes our case study.

Methodology and Related Work. Note that our choice to abstract the channels only to implicitly finite state systems and not to explicitly finite datatypes has a significant advantage from a methodological point of view: the decision saves us from explicitly proving our intuition about the finiteness in Isabelle. This is the reason why our abstraction function is so simple, and no invariants are needed for the proof of its correctness. Thus, by pushing the proof of the invariants to the model checker we delegated as much as possible to the automatic proof tool. Compare with this the experiences made by Havelund and Shankar [HS96] in verifying a communication protocol via abstraction. Their correctness proof of the abstraction in PVS involved 45 of the entire 57 invariants which were needed for the proof without model checker.

Also related to this case study is the result by Abdulla and Jonsson [AJ93] that certain properties of finite state systems communicating via unbounded lossy channels are decidable, which they apply to the Alternating Bit Protocol. However, in our work the channels can both lose and duplicate messages, hence their result does not apply directly.

12.4 Dynamic Memory Management

In this section we present a very simple example of a memory component, that has been refined in our Isabelle framework. The example, however, is powerful enough to illustrate the necessity of history variables. In our setting this means that this time a refinement mapping does not suffice, but a forward simulation has to be applied. Thus, the importance of the correctness proof for forward simulations in §9.3 is demonstrated.

The dataflow of both the specification and the implementation of the memory component is depicted in Fig. 12.10. The task of the component is to manage the memory of a computing system in a dynamic way: new memory locations are requested by the environment (action *New*), then allocated by the system (action *Loc*), and finally set free again by the environment (action *Free*). The implementation of the system is simply done by providing the memory locations one-by-one in an increasing manner, where garbage collection does not take place, i.e. the action *Free* has no effect on the system.

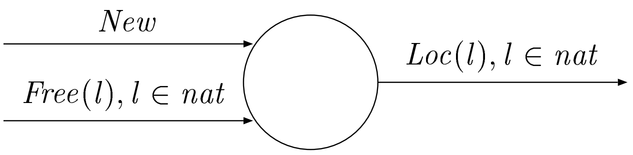

Figure 12.10: Dataflow of the Memory Component

Specification. Concretely, the state space of the specification A is given by a set *used* of natural numbers, which characterizes the locations that are actually allocated, and a flag *new* which indicates whether a new request has arrived.

Field	Type	Initially
$used$:	$(nat)set$	$\emptyset$
new:	$bool$	$false$

The transitions of the specification are given as follows:

input New
  post: $new := true$

output $Loc(l), l \in nat$
  pre: $new \wedge l \notin used$
  post: $used := used \cup \{l\}$;
    $new := false$

input $Free(l), l \in nat$
  post: $used := used \setminus \{l\}$

Implementation. The implementation C employs instead of a set of natural numbers just one natural number, which indicates the point up to which locations have already been allocated. Furthermore, a flag *new* is used, which has the same meaning as before.

Field	Type	Initially
inc:	nat	0
new:	$bool$	$false$

The transitions of the implementation are given as follows:

input *New* **post**: $new := true$	**output** $Loc(l), l \in nat$ **pre**: $new \wedge l = inc$ **post**: $inc := inc + 1;\ new := false$	**input** $Free(l), l \in nat$ **post**: *No effect*

Refinement. Let us first prove that there is not any refinement mapping that could establish the desired refinement. Suppose f is such a refinement mapping. Then, it holds in particular, that for all locations $l \in nat$, if $s \overset{Free(l)}{\rightarrow}_C t$ then $f(s) \overset{Free(l)}{\Rightarrow}_A f(t)$. As the action *Free* has no effect in C, and A has no internal actions, this implication reduces to $t = s \Rightarrow f(s) \overset{Free(l)}{\rightarrow}_A f(t)$. As the action $Free(l)$ in A removes l from the set of allocated locations, this in turn reduces to $f(s).used = f(s).used \setminus \{l\}$. This, however, can only be fulfilled for all l if $f(s).used = \emptyset$ for all s. Obviously, this trivial function f would not be a refinement mapping for *Loc*. On the contrary, *Loc* requires that $f((s.inc{+}1, s.new)).used = f(s).used \cup \{s.inc\}$, which is shown analogously to the *Free* case above. The problem is that the implementation does not perform garbage collection, i.e. it forgets about locations which are no longer in use. This results in a lack of information, which is revealed by the *Free* action, that handles the removal of no longer needed locations. What we need is either a history variable, that keeps track of all returned locations in the implementation, or a relation that leaves it open whether a location is still in use or not. We take the latter approach and define the following relation:

$$\begin{array}{lcl} R & :: & ((nat \times bool) \times ((nat)set \times bool))set \rightarrow bool \\ R & \equiv & \{(s,t).\ (\forall l \in t.used.\ l < s.inc) \wedge s.new = t.new\} \end{array}$$

Our I/O automata framework proves nearly automatically that R is a forward simulation between the two systems. Thus, the correctness result for forward simulations proved in §9.3 yields the desired implementation relation within Isabelle.

12.5 Conclusion

The case studies of this chapter showed the power and usability of our I/O automata framework in Isabelle. In particular, we demonstrated the support for forward simulations and the combination of Isabelle with model checking using the two different abstraction variants. The last case study demonstrated the necessity of forward simulations. The first two case studies permit a number of conclusions, which are presented in the following.

Cockpit Alarm System. Because of its industrial background this study allows first conclusions w.r.t. the scalability of our approach. In particular, it demonstrates impressively that abstraction techniques allow for significantly better verification results than those obtainable by pure model checking or pure theorem proving approaches. Whereas model checking could handle at most 4 alarms in [PS97], we could prove safety-critical properties for the full-size system with 16 alarms. The properties merely refered to the single alarm *PonR*, but this specific alarm could be replaced by any other alarm[2]. In addition, the needed effort was distinctly lower than for a pure refinement proof in Isabelle. As a single drawback we should mention the intuition which is needed to find effective abstractions. Further work should investigate heuristics that suggest promising abstraction functions.

Furthermore, the study showed the usability and practicability of our tool environment. The entire study was carried out in less than two months by a student, who knew neither Isabelle, STeP, I/O automata, nor the cockpit alarm system in advance. The main reasons for this ease of use may be the simple model of I/O automata, its direct representation in Isabelle/HOL, and the tailored proof tactics that support refinement and abstraction.

Finally, the study suggests the following methodology concerning liveness abstractions. It seems to be favorable not to invent new abstraction functions for the liveness part, but to stay with the function from the safety part and to add (or remove) assumptions from one of the models using our extended abstraction rule. This extends the possibilities of reuse and requires less intuition.

Alternating Bit Protocol. A characteristic feature of this case study is the embedding of abstraction techniques into the context of layered implementations and automata networks. It is shown how the use of compositionality can be employed to reduce the desired abstraction to deal only with selected, small components of the entire system. This reduces the interactive part of the proof considerably.

The second characteristic feature of this study has the the same advantage. The abstraction function is chosen in such a way, that the intuition of the protocol's behaviour needs not to be proved explicitly. Rather it can be delegated to the automatic proof tool. Thus, the time-consuming interactive proof of invariants can significantly be reduced. This stands in contrast to existing attempts w.r.t. abstractions in PVS [HS96]. The drawback is that we get only an implicitly finite state system in the concrete example.

Furthermore, we identified an inherent mismatch between the interleaving semantics and input-enabledness of I/O automata on the one hand and the intention of model checking on the other hand. Interleaving and input-enabledness forces one to model unbounded buffers which are not amenable to finite state model checking. We proposed a general solution: the buffers are shifted to an additional component which models the relevant behaviour of the environment from the system's point of view.

Finally, the study showed the importance of data abstraction as opposed to state abstraction, which is not supported in our abstraction theory, as the corresponding refinement concept does not exist in I/O automata theory (cf. §4.5). Fortunately, we could transfer results by Wolper [Wol86] to our concrete setting. We also indicated the problems that are to expect if we would try to apply this approach to our Isabelle framework in general.

[2]The employed abstraction function would merely fail for properties that refer to several alarms that interact with each other.

Chapter 13

Conclusion

In this chapter we review the results achieved in this thesis and give an outline of possible future work.

13.1 Summary

In this thesis we extended untimed I/O automata by a temporal logic and by abstraction rules and formalized the resulting theory in Isabelle. Thus, we yield a completely verified environment for the analysis of embedded systems, including a-priori and a-posteriori verification and a combination of theorem proving with model checking.

Table 13.1 gives a quantitative impression of this verification framework.

	Number of lemmas	Number of spec lines	Number of proof steps
Sequences	167	224	498
Automata (Signatures, I/O Automata, Traces, Refinements)	90	634	314
Meta-Theory (Simulations, Compositionality, Noninterference)	127	408	1150
Extensions (Temporal Logic, Live I/O Automata, Abstraction)	81	386	358
Total	**465**	**1652**	**2322**

Table 13.1: Proof Statistics of the entire I/O Automata Framework

A remarkable observation is that the number of proof steps needed in the average case for establishing a lemma is much lower for the sequence package than for the meta-theory of

I/O automata. This demonstrates the adequacy of domain theory for modeling sequences on the one hand, and the inherent difficulty of proving results like compositionality in a rigorous way on the other hand. Recall the discussion in §8.6 for a more detailed evaluation of the I/O automata tool set.

Apart from this concrete verification environment this thesis provides contributions which are not only relevant for I/O automata and their tool support in Isabelle, but could even give further stimulations for the formal methods community in a broader sense. These contributions cover the following fields:

Temporal Logic. The original motivation for TLS has been the lack of a property specification language which could also be used to facilitate implementation proofs between live I/O automata. However, it also provides deeper insights into related logics for program development. Characteristic for TLS is its restricted use of stuttering which permits a reasonable employment of the next-time operator. This allowed us to establish a formal relation between TLS and the temporal logics by Manna/Pnueli and Kröger. On the other hand, TLS is closely related to TLA. In particular, the use of TLS for specifying liveness conditions for I/O automata allows for a direct comparison between TLA and I/O automata: whereas TLA features a more complicated basic model, I/O automata possess a more sophisticated refinement and abstraction concept. The reason is that TLA incorporates arbitrary stuttering already in the basic model, whereas I/O automata require refinement mappings and abstraction functions to add stuttering explicitly by internal actions.

Liveness. Proofs of liveness properties are significantly more difficult than safety proofs, because they deal with the long-term behaviour of a system which cannot be easily reduced to a property about single steps. Concerning I/O automata in particular, a further difficulty represents the transfer of liveness from concrete to abstract automata. Previously, this has been done by complicated reasoning about index mappings (Definition 2.4.5). We improve significantly upon both problems. First, TLS allows us to provide theorems which accomplish the aforementioned transfer for the usual case of fairness. Thus, fairness proofs can be performed by merely using tautologies of temporal logic. Second, we show that abstractions even allow to reduce these remaining tautologies of temporal logic to local reasoning about steps. This is possible as the specific abstraction functions permit to transfer arbitrary liveness between implementation and specification. As a result, the inherently difficult liveness proofs can be delegated to fully automatic model checking.

Abstraction. Our abstraction theory is of general interest, because of the following reasons. First, we unify several ideas of abstraction which have previously been treated separately for a-priori and a-posteriori verification. Second, our completeness result shows the formal relation of abstraction to refinement. Third, we propose a methodology that reduces the intuition usually required to find suitable (live) abstraction functions/relations. This represents a first step towards a reasonable automation of abstraction proofs.

Theory of Sequences. Our formalization of possibly infinite sequences represents a general-purpose theory which appears to be useful in a lot of applications. The theory turned out to be very successful already for modeling runs of I/O automata and for defining a temporal logic. Furthermore, we believe that our investigations concerning a combination of induction and coinduction can easily be generalized to arbitrary lazy datatypes.

HOL/LCF Methodology. Our HOL/LCF methodology appears to be mandatory for every serious application of HOLCF. While technically being based on a trivial and well-known construction, it is of significant importance from a methodological point of view. Concerning our sequence theory, it enables the reuse of HOL libraries, increases the degree of automation, and saves us from reasoning about undefinedness in many cases. With respect to our I/O automata verification environment, it even allows us to hide the more complicated logic HOLCF from the user without being forced to abandon its power for meta-theoretic investigations. We are convinced that this applies more generally: the combination of different logics and the structuring of verification tasks into different logical layers will be one of the future challenges in theorem proving.

13.2 Further Work

There is a considerable amount of further work remaining. First, the employment of abstractions could be enhanced w.r.t. two aspects:

- As mentioned in §4.5 already, our abstraction rules correspond to state refinement. It is desirable to complement these by a notion of data abstraction, which would correspond to some kind of action refinement. Therefore, future research should investigate how action refinement available for formalisms like FOCUS [BDD$^+$93] carries over to I/O automata.

- A disadvantage of abstraction is the intuition required to find suitable abstraction functions. Therefore, it is desirable to automate this process. The following two suggestions could be a promising starting point.

  1. Further case studies should be performed in order to create a catalogue of abstraction patterns. These patterns should be tailored for specific datatypes (lists, stacks, queues, sets, etc.) and provide a list of standard properties each of them equipped with the required abstraction function and useful strengthenings/weakenings for liveness.

2. The methodology for generating abstractions incrementally proposed in §4.2 should be supported by an appropiate tool set. A promising candidate for that may be the environment currently being built in the QUEST project [Que98], where theorem proving, model checking, and testing is combined with the design tool AutoFocus [HSSS96].

Second, the verification environment for I/O automata described in §8 – §11 would gain from further tool-specific enhancements and additions. In detail, the following extensions are desirable:

- Currently, I/O automata have to be specified in Isabelle in the basic set comprehension format described in §8.2. More user-friendly would be an I/O automaton language closer to the informal precondition/effect style introduced in §2.5, like the one defined in [GLV97]. This language should be supported by an associated compiler and a syntax checker.
- So far, the STeP model checker is not actually integrated with Isabelle and the tactic that invokes μcke as an oracle permits only primitive datatypes such as booleans and pairs. A neater integration has to automate the model checking translations proposed in §4.4. This includes an identification of a well-defined sublanguage of those I/O automata that can be transformed to a finite-state description.
- The proof infrastructure tailored for I/O automata verification in Isabelle should be further enhanced. In particular, this includes specialized proof rules and tactics for the TLS. Furthermore, methodological guidelines should be provided, which contribute to a neat correspondence between paper proofs and machine proofs. This concerns, for example, the organization of rewriting using a tailored hierarchy of simplification sets.
- A further useful addition could be verification diagrams as proposed by Browne, Manna, and Sipma [BMS95]. These diagrams allow to specify the proof sketch of refinement and abstraction proofs graphically. This is in particular useful for the verification of liveness properties.

Third, the concepts underlying the I/O automata formalization can gainfully be transfered to other settings, in particular to other verification formalisms for distributed systems. More concretely, this applies to the following areas:

- As already mentioned above, the sequence formalization could be reused in a variety of different contexts. An interesting example represents the requirement phase in Focus [BDD+93]. In this phase predicates on sequences are used to model initial requirements on a system's behaviour. Here, an employment of our sequence theory would be promising because of the simple means to express recursive predicates, as explained in §6.3. Furthermore, the proof automation due to the restriction to lifted HOL elements would be of advantage.

- The I/O automata formalization can be adapted to other automaton models describing reactive, distributed systems. This applies, for example, to the models developed in [Bro97, Kle98, Rum97, HSSS96], which (due to their synchronous input/output behaviour) are specifically tailored to support the design phase of FOCUS. In particular, an adaptation to the automaton model underlying AUTOFOCUS [HSSS96] would offer the possibility to combine verification with further tools like graphical editors, simulators and code generators. For such intentions it would be of vital importance to adopt our HOL/HOLCF methodology, which separates meta-theory and theory w.r.t. the employed logic.

Finally, it would be challenging and interesting to extend the automaton formalization to timing-based systems, such as timed automata [LV96] or hybrid I/O automata [LSVW96]. As even the basic support for linear and real arithmetic in Isabelle is rather preliminary so far, this would, however, represent a major undertaking.

Part V: Appendix

Appendix A

Selected Additional Definitions

In the following we present some definitions that have not been mentioned explicitly in this thesis. Let α be a type scheme of type class pcpo.

Definition A.0.1 (Last)
The function slast returns the last element of a sequence, if it exists, otherwise $\perp$.

$$\begin{array}{rlcl} & \mathsf{slast} & :: & (\alpha)\,\mathsf{seq} \to_c \alpha \\ & \mathsf{slast}\,`\perp & = & \perp \\ & \mathsf{slast}\,`\mathsf{nil} & = & \perp \\ x \neq \perp \Rightarrow & \mathsf{slast}\,`(x \,\#\, xs) & = & \textbf{If } \mathsf{is\text{-}nil}\,`xs \textbf{ then } x \\ & & & \qquad\qquad \textbf{else } \mathsf{slast}\,`xs \end{array}$$

The function Last specializes slast to arguments of type class term.

$$\begin{array}{lcl} \mathsf{Last} & :: & (\alpha_{\mathsf{term}})\,\mathsf{sequence} \to_c (\alpha_{\mathsf{term}})\,\mathsf{lift} \\ \mathsf{Last} & \equiv & \mathsf{slast} \end{array}$$

$\Box$

Definition A.0.2 (Takewhile)
The function $\mathsf{stakewhile}$ returns the longest prefix of a sequence where all elements obey a given predicate.

$$\mathsf{stakewhile} \;::\; (\alpha \to_c \mathsf{tr}) \to_c (\alpha)\,\mathsf{seq} \to_c (\alpha)\,\mathsf{seq}$$

$$\begin{array}{rlcl}
 & \mathsf{stakewhile}\,`P\,`\bot & = & \bot \\
 & \mathsf{stakewhile}\,`P\,`\mathsf{nil} & = & \mathsf{nil} \\
x \neq \bot \Rightarrow & \mathsf{stakewhile}\,`P\,`(x \# xs) & = & \textbf{If } P\,`x \textbf{ then } x \# \mathsf{stakewhile}\,`P\,`xs \\
 & & & \qquad \textbf{else } \mathsf{nil}
\end{array}$$

□

Definition A.0.3 (Dropwhile)
The function sdropwhile represents the dual to stakewhile.

$$\begin{array}{rlcl}
 & \mathsf{sdropwhile} & :: & (\alpha \rightarrow_c \mathsf{tr}) \rightarrow_c (\alpha)\,\mathsf{seq} \rightarrow_c (\alpha)\,\mathsf{seq} \\
 & \mathsf{sdropwhile}\,`P\,`\bot & = & \bot \\
 & \mathsf{sdropwhile}\,`P\,`\mathsf{nil} & = & \mathsf{nil} \\
x \neq \bot \Rightarrow & \mathsf{sdropwhile}\,`P\,`(x \# xs) & = & \textbf{If } P\,`x \textbf{ then } \mathsf{sdropwhile}\,`P\,`xs \\
 & & & \qquad \textbf{else } x \# xs
\end{array}$$

□

Let α and β be type schemes of type class term.

Definition A.0.4 (Zip)
The function Zip merges two sequences to a sequence of pairs.

$$\begin{array}{rlcl}
 & \mathsf{Zip} & :: & (\alpha)\,\mathsf{sequence} \rightarrow_c (\beta)\,\mathsf{sequence} \rightarrow_c (\alpha \times \beta)\,\mathsf{sequence} \\
 & \mathsf{Zip}\,`\bot\,`y & = & \bot \\
x \neq \mathsf{nil} \Rightarrow & \mathsf{Zip}\,`x\,`\bot & = & \bot \\
 & \mathsf{Zip}\,`\mathsf{nil}\,`y & = & \mathsf{nil} \\
 & \mathsf{Zip}\,`(x\hat{\ }xs)\,`\mathsf{nil} & = & \bot \\
 & \mathsf{Zip}\,`(x\hat{\ }xs)\,`(y\hat{\ }ys) & = & (x, y)\hat{\ }\mathsf{Zip}\,`xs\,`ys
\end{array}$$

□

Definition A.0.5 (Flatten)
The function Flatten specializes sflatten to arguments of type class term.

$$\begin{array}{lcl}
\mathsf{Flatten} & : & ((\alpha)\,\mathsf{sequence})\,\mathsf{seq} \rightarrow_c (\alpha)\,\mathsf{sequence} \\
\mathsf{Flatten} & \equiv & \mathsf{sflatten}
\end{array}$$

□

Appendix B

Selected Additional Theorems

In the following we present a selection of those lemmas about sequences and I/O automata that have been used during the development of the verification framework, but have not been explicitly mentioned in this thesis.

B.1 Sequences

Lemma B.1.1 (Constructors and Selectors)
Lemmas proved about the constructors and destructors include the following:

$$\begin{array}{l}
\mathsf{HD}\ `(a\hat{\ }s) = \mathsf{Def}(a) \\
\mathsf{TL}\ `(a\hat{\ }s) = s \\
s = \bot \lor s = \mathsf{nil} \lor (\exists a\ t.\ s = a\hat{\ }t) \\
a\hat{\ }s \neq \bot \land a\hat{\ }s \neq \mathsf{nil} \\
\neg(a\hat{\ }s) \sqsubseteq \bot \land \neg(a\hat{\ }s) \sqsubseteq \mathsf{nil} \\
(a\hat{\ }s = b\hat{\ }t) = (a = b \land s = t) \\
(a\hat{\ }s \sqsubseteq b\hat{\ }t) = (a = b \land s \sqsubseteq t)
\end{array}$$

$$\frac{\begin{array}{ccc} s = \bot & s = \mathsf{nil} & s = a\hat{\ }t \\ \vdots & \vdots & \vdots \\ P & P & P \end{array}}{P}$$

□

Lemma B.1.2 (Finiteness)
Lemmas proved about Finite include the following:

$$\begin{array}{l}
\mathsf{Finite}(s \oplus t) = (\mathsf{Finite}(s) \land \mathsf{Finite}(t)) \\
\mathsf{Finite}(\mathsf{Map}\, f\ `s) = \mathsf{Finite}(s) \\
\mathsf{Finite}(s) \Rightarrow \mathsf{Finite}(\mathsf{Filter}\ P\ `s)
\end{array}$$

$$\mathsf{Finite}(s) \Rightarrow \forall t.\ \mathsf{Finite}(t) \wedge s \sqsubseteq t \Rightarrow s = t$$
$$\mathsf{adm}(\mathsf{Finite}(s))$$

□

Lemma B.1.3 (Recursive Functions)
Lemmas proved about $\oplus$ and Last include the following:

$$\mathsf{Finite}(s) \Rightarrow ((s \oplus t) = (s \oplus u)) = (t = u)$$
$$(s \oplus t) \oplus u = s \oplus t \oplus u$$
$$s \oplus \mathsf{nil} = s$$
$$(s \oplus t = \mathsf{nil}) = (s = \mathsf{nil} \wedge t = \mathsf{nil})$$
$$\mathsf{Finite}(s) \Rightarrow \mathsf{Last}\,`s = \bot \Rightarrow s = \mathsf{nil}$$

Lemmas proved about Filter include the following:

$$\mathsf{Filter}\ P\ `(\mathsf{Filter}\ Q\ `xs) = \mathsf{Filter}\ (\lambda x.\ P(x) \wedge Q(x))\ `xs$$
$$\mathsf{Filter}\ P\ `(x \oplus y) = \mathsf{Filter}\ P\ `x \oplus \mathsf{Filter}\ P\ `y$$

Lemmas proved about Map include the following:

$$\mathsf{Map}\,f\ `(\mathsf{Map}\ g\ `x) = \mathsf{Map}\ (f \circ g)\ `x$$
$$\mathsf{Map}\,f\ `(x \oplus y) = \mathsf{Map}\,f\ `x \oplus \mathsf{Map}\,f\ `y$$
$$\mathsf{Filter}\ P\ `(\mathsf{Map}\,f\ `x) = \mathsf{Map}\,f\ `(\mathsf{Filter}\ (P \circ f)\ `x)$$
$$\mathsf{Forall}\ P\ (\mathsf{Map}\,f\ `x) = \mathsf{Forall}\ (P \circ f)\ x$$
$$(\mathsf{Map}\,f\ `s = \mathsf{nil}) = (s = \mathsf{nil})$$

Lemmas proved about Forall include the following:

$$\mathsf{Finite}(x) \Rightarrow \mathsf{Forall}\ P\ (x \oplus y) = (\mathsf{Forall}\ P\ x \wedge \mathsf{Forall}\ P\ y)$$
$$\mathsf{Forall}\ P\ x \wedge \mathsf{Forall}\ P\ y \Rightarrow \mathsf{Forall}\ P\ (x \oplus y)$$
$$\mathsf{Forall}\ P\ x \Rightarrow y \sqsubseteq x \Rightarrow \mathsf{Forall}\ P\ y$$
$$\mathsf{Forall}\ P\ s \wedge (\forall a.\ P(a) \Rightarrow Q(a)) \Rightarrow \mathsf{Forall}\ Q\ s$$
$$\mathsf{Finite}(z) \wedge \mathsf{Forall}\ P\ x \wedge x = z \oplus y \Rightarrow \mathsf{Forall}\ P\ y$$
$$\mathsf{Forall}\ P\ s \Rightarrow \mathsf{Forall}\ P\ (\mathsf{TL}\,`s)$$
$$\mathsf{Forall}\ P\ s \Rightarrow \mathsf{Forall}\ P\ (\mathsf{Dropwhile}\ Q\ `s)$$

Lemmas proved about Forall in combination with Filter include the following:

$$\mathsf{Forall}\ P\ (\mathsf{Filter}\ P\ `x)$$
$$\mathsf{Forall}\ P\ x \Rightarrow \mathsf{Filter}\ P\ `x = x$$

$$(\mathsf{Finite}(x) \land \mathsf{Forall}\ (\lambda a.\ \neg P(a))\ x) = (\mathsf{Filter}\ P\ `x = \mathsf{nil})$$
$$(\neg\mathsf{Finite}(x) \land \mathsf{Forall}\ (\lambda a.\ \neg P(a))\ x) = (\mathsf{Filter}\ P\ `x = \bot)$$
$$(\mathsf{Finite}(x) \land \mathsf{Forall}\ Q\ x \land (\forall a.\ Q(a) \Rightarrow \neg P(a)) \Rightarrow \mathsf{Filter}\ P\ `x = \mathsf{nil}$$
$$(\neg\mathsf{Finite}(x) \land \mathsf{Forall}\ Q\ x \land (\forall a.\ Q(a) \Rightarrow \neg P(a)) \Rightarrow \mathsf{Filter}\ P\ `x = \bot$$
$$\frac{\mathsf{Forall}\ Q\ s \quad \mathsf{Finite}(s) \quad \forall a.\ Q(a) \Rightarrow \neg P(a)}{\mathsf{Filter}\ P\ `s = \mathsf{nil}}$$
$$\frac{\mathsf{Forall}\ Q\ s \quad \neg\mathsf{Finite}(s) \quad \forall a.\ Q(a) \Rightarrow \neg P(a)}{\mathsf{Filter}\ P\ `s = \bot}$$

Lemmas proved about Takewhile and Dropwhile include the following:

$$\mathsf{Forall}\ P\ (\mathsf{Takewhile}\ P\ `x)$$
$$(\forall a.\ Q(a) \Rightarrow P(a)) \Rightarrow \mathsf{Forall}\ P\ (\mathsf{Takewhile}\ Q\ `x)$$
$$\mathsf{Finite}\ (\mathsf{Takewhile}\ Q\ `x) \land (\forall a.\ Q(a) \Rightarrow \neg P(a)) \Rightarrow \mathsf{Filter}\ P\ `(\mathsf{Takewhile}\ P\ `x) = \mathsf{nil}$$
$$(\forall a.\ Q(a) \Rightarrow P(a)) \Rightarrow \mathsf{Filter}\ P\ `(\mathsf{Takewhile}\ Q\ `x) = \mathsf{Takewhile}\ Q\ `x$$
$$\mathsf{Takewhile}\ P\ `(\mathsf{Takewhile}\ P\ `x) = \mathsf{Takewhile}\ P\ `x$$
$$\mathsf{Forall}\ P\ x \Rightarrow \mathsf{Takewhile}\ P\ `(x \oplus y) = x \oplus (\mathsf{Takewhile}\ P\ `y)$$
$$\mathsf{Forall}\ P\ s \Rightarrow \mathsf{Takewhile}\ (\lambda a.\ Q(a) \lor \neg P(a))\ `s = \mathsf{Takewhile}\ Q\ `s$$
$$\mathsf{Forall}\ P\ s \Rightarrow \mathsf{Dropwhile}\ (\lambda a.\ Q(a) \lor \neg P(a))\ `s = \mathsf{Dropwhile}\ Q\ `s$$
$$\mathsf{Finite}(x) \land \mathsf{Forall}\ P\ x \Rightarrow \mathsf{Dropwhile}\ P\ (x \oplus y) = \mathsf{Dropwhile}\ P\ `y$$

□

B.2 I/O Automata

Lemma B.2.1 (Action Signatures)
Lemmas proved about action signatures include the following:

$$a \in \mathsf{externals}(sig) \Rightarrow a \in \mathsf{actions}(sig)$$
$$a \in \mathsf{internals}(sig) \Rightarrow a \in \mathsf{actions}(sig)$$
$$a \notin \mathsf{internals}(sig) \land a \notin \mathsf{externals}(sig) \Rightarrow a \notin \mathsf{actions}(sig)$$
$$(a \in \mathsf{actions}(sig) \land a \in \mathsf{externals}(sig)) = (a \in \mathsf{externals}(sig))$$
$$\mathsf{is\text{-}sig}(sig) \land a \in \mathsf{internals}(sig) \Rightarrow a \notin \mathsf{externals}\ sig$$
$$\mathsf{is\text{-}sig}(sig) \land a \in \mathsf{externals}(sig) \Rightarrow a \notin \mathsf{internals}\ sig$$
$$\mathsf{is\text{-}sig}(sig) \Rightarrow (a \notin \mathsf{externals}(sig)) = (a \in \mathsf{internals}(sig) \lor a \notin \mathsf{actions}(sig))$$

□

Lemma B.2.2 (Automata)
Lemmas proved about signatures of automata include the following:

$$\mathsf{is\text{-}safe\text{-}IOA}(A) \land s \xrightarrow{a}_A t \Rightarrow a \in \mathsf{act}\ A$$

$$\begin{aligned}
&\mathsf{ext}(A \parallel B) = \mathsf{ext}\ A \cup \mathsf{ext}\ B\\
&\mathsf{act}(A \parallel B) = \mathsf{act}\ A \cup \mathsf{act}\ B\\
&\mathsf{out}(A \parallel B) = \mathsf{out}\ A \cup \mathsf{out}\ B\\
&\mathsf{int}(A \parallel B) = \mathsf{int}\ A \cup \mathsf{int}\ B\\
&\mathsf{in}(A \parallel B) = (\mathsf{in}\ A \cap \mathsf{in}\ B) \setminus (\mathsf{out}\ A \cap \mathsf{out}\ B)
\end{aligned}$$

Lemmas proved about compatibility include the following:

$$\begin{aligned}
&\mathsf{compatible}\ A\ B = \mathsf{compatible}\ B\ A\\
&\mathsf{compatible}\ A\ B \wedge a \in \mathsf{ext}\ A \Rightarrow a \notin \mathsf{int}\ B\\
&\mathsf{compatible}\ A\ B \wedge a \in \mathsf{act}\ A \Rightarrow a \notin \mathsf{int}\ B\\
&\mathsf{compatible}\ A\ B \wedge a \in \mathsf{ext}\ A \wedge a \notin \mathsf{ext}\ B \Rightarrow a \notin \mathsf{act}\ B\\
&\mathsf{compatible}\ A\ B \wedge a \in \mathsf{out}\ A \wedge a \in \mathsf{act}\ B \Rightarrow a \in \mathsf{in}\ B\\
&\mathsf{compatible}\ A\ B \wedge a \in \mathsf{in}\ A \wedge a \in \mathsf{act}\ B \Rightarrow a \in \mathsf{in}\ A \vee a \in \mathsf{out}\ B
\end{aligned}$$

Lemmas proved about reachability include the following:

$$\begin{aligned}
&\mathsf{reachable}\ (\mathsf{restrict}\ A\ acts)\ s = \mathsf{reachable}\ A\ s\\
&\mathsf{reachable}\ (\mathsf{rename}\ A\ f)\ s \Rightarrow \mathsf{reachable}\ A\ s
\end{aligned}$$

Lemmas proved about the transition relation of the parallel composition are:

$$\begin{aligned}
&s \xrightarrow{a}_{A\|B} t\ \wedge a \in \mathsf{act}\ A \Rightarrow\ \mathsf{fst}\ s \xrightarrow{a}_{A} \mathsf{fst}\ t\\
&s \xrightarrow{a}_{A\|B} t\ \wedge a \notin \mathsf{act}\ A \Rightarrow \mathsf{fst}\ s = \mathsf{fst}\ t\\
&\mathsf{fst}\ s \xrightarrow{a}_{A} \mathsf{fst}\ t\ \wedge \mathsf{snd}(s) = \mathsf{snd}(t) \wedge a \in \mathsf{act}\ A \Rightarrow\ s \xrightarrow{a}_{A\|B} t\\
&\mathsf{snd}\ s \xrightarrow{a}_{A} \mathsf{snd}\ t\ \wedge \mathsf{fst}(s) = \mathsf{fst}(t) \wedge a \in \mathsf{act}\ B \Rightarrow\ s \xrightarrow{a}_{A\|B} t\\
&\mathsf{fst}\ s \xrightarrow{a}_{A} \mathsf{fst}\ t\ \wedge\ \mathsf{snd}\ s \xrightarrow{a}_{A} \mathsf{snd}\ t\ \wedge a \in \mathsf{act}\ A \wedge a \in \mathsf{act}\ B \Rightarrow\ s \xrightarrow{a}_{A\|B} t\\
&\mathsf{fst}(s) = \mathsf{fst}(t) \wedge \mathsf{snd}(s) = \mathsf{snd}(t) \wedge a \notin \mathsf{act}\ A \wedge a \notin \mathsf{act}B \Rightarrow\ s \xrightarrow{a}_{A\|B} t
\end{aligned}$$

□

Bibliography

[ACW90] S. Aggarwal, C. Courcoubetis, and Pierre Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, 1990.

[AF97] S. Agerholm and J. Frost. An Isabelle-based theorem prover for VDM-SL. In Elsa Gunter, editor, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOL'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1997.

[Age94a] Sten Agerholm. Formalising a model of the λ-calculus in HOL-ST. Technical Report 354, University of Cambridge Computer Laboratory, 1994.

[Age94b] Sten Agerholm. *A HOL Basis for Reasoning about Functional Programs*. PhD thesis, University of Aarhus, Denmark, 1994.

[AH97a] Myla Archer and Constance Heitmeyer. Human-style theorem proving using PVS. In Elsa Gunter, editor, *Proc. 10th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOL'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 33–48. Springer-Verlag, 1997.

[AH97b] Myla Archer and Constance Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Proc. Int. Workshop on Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, 1997.

[AJ93] Parosh Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 160–170. IEEE Press, 1993.

[AJ94] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994.

[AL88] Martin Abadi and Leslie Lamport. The existence of refinement mappings. In *Proc. 3rd IEEE Symp. Logic in Computer Science*, pages 165–177. IEEE Computer Society Press, 1988.

[AL93] Martin Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 1(15):73–132, 1993.

[BBC+96] Nikolaj Bjørner, Anca Browne, Edward Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification: 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, 1996.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus — Revised Version. Technical Report TUM-I9202-2, Technische Universität München, Fakultät für Informatik, 1993.

[BDE+97] M. Broy, W. Damm, M. Eckrich, W. Mala, and G. Venzl. Korrekte Software für sicherheitskritische Systeme: Das Projekt Korsys im Überblick. In *Statusseminar Softwaretechnologie*, pages 160–173, Bonn, Germany, 1997. Bundesministerium für Bildung und Forschung (BMBF).

[Bie97a] A. Biere. μcke – Efficient μ-calculus model checking. In O. Grumberg, editor, *Proc. 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 468–471. Springer-Verlag, 1997.

[Bie97b] Armin Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen.* PhD thesis, Institut für Informatik, Universität Karlsruhe, Germany, 1997.

[BMS95] A. Browne, Z. Manna, and H. Sipma. Generalized temporal verification diagrams. In *Proc. 15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 484–498. Springer-Verlag, 1995.

[BP98] Giampaolo Bella and Larry Paulson. Mechanising BAN kerberos by the inductive method. In *Proc. 10th Int. Conf. Computer-Aided Verification (CAV'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[BPV94] D.J.B. Bosscher, I. Polak, and F.W. Vaandrager. Verification of an audio control protocol. In W.P. de Roever H. Langmaack and J. Vytopil, editors, *Proc. 3rd Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'94)*, volume 863 of *Lecture Notes in Computer Science*, pages 170–192. Springer, 1994.

[Bro85] Manfred Broy. Extensional behaviour of concurrent, nondeterministic, communicating programs. In *Control Flow and Data Flow, Concepts of Distributed Programming*, volume 14 of *NATO ASI Series F: Computer and System Sciences*, pages 229–276. Springer-Verlag, 1985.

[Bro93a] Stephen Brooks. Full abstraction for a shared variable parallel language. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 98–109, 1993.

[Bro93b] M. Broy. Interaction Refinement – The Easy Way. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and System Sciences*. Springer, 1993.

[Bro97] M. Broy. The specification of system components by state transition diagrams. Technical Report TUM-I9729, Technische Universität München, Fakultät für Informatik, 1997.

[BSW69] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM*, 12(5):260–261, 1969.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGH94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *Proc. 6th Int. Conf. on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427, Standford, California, USA, June 1994. Springer-Verlag.

[CGL92] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proc. 19th ACM Symp. Principles of Programming Languages*, pages 343–354. ACM Press, 1992.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer-Verlag, 1994.

[Cho93] C.-T. Chou. Predicates, temporal logic, and simulations. In J.J. Joyce and J.H. Seger, editors, *Proc. 6th Int. Workshop on Higher Order Logic Theorem Provers and Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 310–323. Springer-Verlag, 1993.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[CP96] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Proc. 2nd Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Ded92] Frank Dederichs. *Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen*. PhD thesis, Technische Universität München, 1992.

[DF95] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Proc. 7th Int. Conf. Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 54–69. Springer-Verlag, 1995.

[DFH+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, and C. Parent et al. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, May 1993.

[DG97] Marco Devillers and David Griffioen. A formalization of finite and infinite sequences in PVS. Technical Report CSI-R9702, Computing Science Institute, University of Nijmegen, 1997.

[DGG97] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):22–43, 1997.

[DGM97] Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In Elsa Gunter, editor, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOL'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, 1997.

[DL97] Ekaterina Dolginova and Nancy Lynch. Safety verification for automated platoon maneuvers: A case study. In *Proc. Int. Workshop on Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 154–170. Springer-Verlag, 1997.

[dRe97] Willem-Paul de Roever (editor). *International Symposium on Compositionality – The Significant Difference*. Springer-Verlag, September 1997. Malente, Germany, Lecture Notes in Computer Science, to appear.

[Eme90] E.A. Emerson. Temporal and modal logics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072. Elsevier Science Publishers B.V., 1990.

[ESA96] *Joint Press Release Ariane 501.* 1996. ESA-CNES, http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

[Fef96] Solomom Feferman. Computation on abstract data types. The extensional approach, with an application to streams. *Annals of Pure and Applied Logic*, 81:75–113, 1996.

[Fio96] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* Cambridge University Press, 1996.

[GD98] David Griffioen and Marco Devillers. *Personal Communication.* 1998.

[GG91] S. Garland and J. Guttag. A guide to LP, the Larch Prover. Technical Report TR-82, DEC Systems Research Center, 1991.

[GL93] Susanne Graf and Claire Loiseaux. A tool for symbolic program verification and abstraction. In C. Courcoubetis, editor, *3th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 71–84. Springer-Verlag, 1993.

[GLV97] Stephen Garland, Nancy Lynch, and Mandana Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, Laboratory for Computer Science, MIT, Cambridge, MA., September 1997.

[GM93] M.C.J. Gordon and T.F. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

[Gor94] M.J.C. Gordon. Merging HOL with set theory: preliminary experiments. Technical Report 353, University of Cambridge Computer Laboratory, 1994.

[Gor95] Andrew Gordon. A tutorial on co-induction and functional programming. In *Proc. Glasgow Workshop on Functional Programming*, pages 78–95. Springer Workshops in Computing, 1995.

[GPSS80] Dov Gabbay, Amir Pnueli, S. Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM, 1980.

[Gri95] David Griffioen. Proof-checking an audio control protocol with LP. Technical Report CS-R9570, CWI Amsterdam, 1995.

[GSSL93] R. Gawlick, R. Segala, J.F. Sogaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, MIT, Cambridge, MA., December 1993.

[GV98] D. Griffioen and F. Vaandrager. Normed simulations. In *Proc. 10th Int. Conf. Computer-Aided Verification (CAV'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110:305–326, 1994.

[Ham98] Tobias Hamberger. Verifikation einer Hubschrauberüberwachungskomponente mit Isabelle und STeP. 1998. Student software development project, Technische Universität München, Germany.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[HJ97a] U. Hensel and B. Jacobs. Coalgebraic theories of sequences in PVS. Technical Report CSI-R9708, Computing Science Institute, University of Nijmegen, 1997.

[HJ97b] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. Technical Report CSI-R9703, Computing Science Institute, University of Nijmegen, 1997.

[HN96] D. Harel and A. Naamad. The Statemate Semantics of Statecharts. *ACM Transactions On Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[HP94] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. 7th Int. Conf. on Formal Description Techniques (FORTE'94)*, pages 177–194, Berne, Switzerland, 1994.

[HR87] Z. Har'El and R. P. Kurshan. The COSPAN User's Guide. Technical Report 11211-871009-21TM, AT&T Bell Laboratories, 1987.

[HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M. Gaudel and J. Woodcock, editors, *Formal Methods Europe*, Lecture Notes in Computer Science, pages 662–681. Springer-Verlag, 1996.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus – a tool for distributed systems specification. In Joachim Parrow Bengt Jonsson, editor, *Proc. FTRTFT'96 – Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, pages 467–470. Springer-Verlag, 1996.

[HSV94] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings Workshop Esprit BRA "Types for Proofs and Programs"*, Nijmegen, The Netherlands, May 1993, number 806 in Lecture Notes in Computer Science. Springer-Verlag, 1994. Full version available as Report CS-R9420, CWI, Amsterdam, March 1994.

[Hun93] Hardi Hungar. Combining model checking and theorem proving to verify parallel processes. In C. Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 154–165. Springer-Verlag, 1993.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[Kel95] Peter Kelb. *Abstraktionstechniken für automatische Verifikationstechniken.* PhD thesis, Universität Oldenburg, Germany, 1995. In German.

[KL93] R.P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Proc. 5th Int. Conf. Computer-Aided Verificatio (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 166–182. Springer-Verlag, 1993.

[Kle98] Cornel Klein. *Anforderungsspezifikation mit Transitionssystemen und Szenarien.* PhD thesis, Technische Universität München, 1998.

[KMOS95] R.P. Kurshan, M. Merritt, A. Orda, and S.R. Sachs. Modelling asynchrony with a synchronous model. In P. Wolper, editor, *Proc. 7th Int. Conf. Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 339–352. Springer-Verlag, 1995.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(1):333–354, 1983.

[Krö87] Fred Kröger. *Temporal Logic of Programs.* Springer-Verlag, 1987.

[Kur87] R.P. Kurshan. Reducibility in analysis of coordination. In Kurzhanski Varaiya, editor, *Discrete Event Systems: Models and Applications*, volume 103 of *Lecture Notes in Control and Information Science*, pages 19–39. Springer-Verlag, 1987.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes – The Automata-Theoretic Approach.* Princeton University Press, 1994.

[Lam83] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lån94] Thomas Långbacka. A HOL formalisation of the Temporal Logic of Actions. In T.F. Melham and J. Camilleri, editors, *Proc. 7th Int. Workshop on Higher Order Logic Theorem Provers and Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 332–347. Springer-Verlag, 1994.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[LMWF94] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.

[LPM93] Francois Leclerc and Christine Paulin-Mohring. Programming with streams in Coq, a case study: the sieve of eratosthenes. In H. Barendregt and T. Nipkow, editors, *Proc. Types for Proofs and Programs (TYPES'93)*, volume 806 of *Lecture Notes in Computer Science*, 1993.

[LSGL94] V. Luchangco, E. Söylemez, S. Garland, and N.A. Lynch. Verifying timing properties of concurrent algorithms. In *Proc. 7th Int. Conf. Formal Description Techniques (FORTE'94)*, pages 259–273. Chapman & Hall, 1994.

[LSVW96] N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer, 1996. Also appeared as Technical Report CS-R9578 at CWI, Computer Science Department, Amsterdam, 1995.

[LT87] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., 1987. Short version appeared at the 6th Annual ACM Symposium on Principles of Distributed Computing, pages 137–151, 1987.

[LT89] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LV95] N.A. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[LV96] N.A. Lynch and F. Vaandrager. Forward and backward simulations – part II: Timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[MBB+97] Z. Manna, N. Bjoerner, A. Browne, E. Chang, M. Colon, B. Finkbeiner, A. Kapur, H. Sipma, and T. Uribe. STeP – The Stanford Temporal Prover, User's Manual. Technical report, Computer Science Department, Stanford University, California 94305, 1997. Educational release, version 1.3.

[Mer95] Stephan Merz. Mechanizing TLA in Isabelle. In *Workshop Verification in New Orientations*, 1995. Technical Report, University Maribor.

[Mer97] Stephan Merz. Rules for abstraction. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science—ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 32–45, Kathmandu, Nepal, December 1997. Springer-Verlag.

[MG95] W. Mala and E. Grandi. Industrie Anwendung Avionik – Systemkomponente Hubschrauberüberwachung. 1995. Internal Report, ESG Elektroniksystem- und Logistik GmbH, München, Germany.

[MN95] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O-automata. In *Proc. 1st Workshop Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1995.

[MN97] Olaf Müller and Tobias Nipkow. Traces of I/O-automata in Isabelle/HOLCF. In *Proc. 7th Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 580–595. Springer-Verlag, 1997.

[MNOS98] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *J. Functional Programming*, 1998. submitted.

[MP95] Z. Manna and A Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, NY, 1995.

[MS95] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, Florida, USA, 1995. IEEE Computer Science.

[MS97] Olaf Müller and Konrad Slind. Treating partiality in a setting of total functions. *The Computer Journal*, 40(10):1–12, 1997.

[Mül98] Olaf Müller. I/O automata and beyond - temporal logic and abstraction in Isabelle. In Jim Grundy and Malcolm Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOL'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1998.

[NS95] Tobias Nipkow and Konrad Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 101–119. Springer-Verlag, 1995.

[NvO98] Tobias Nipkow and David von Oheimb. Java_{light} is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, New York, 1998.

[Ohe95] D. v. Oheimb. Datentypspezifikationen in Higher-Order LCF. Master's thesis, Computer Science Department, Technical University Munich, 1995.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*. Springer, 1996.

[ORSH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Par76] D. Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 1(3):173–181, 1976.

[Pau87] Lawrence C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Pau97] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Logic and Computation*, 7:175 – 204, 1997.

[Pit94] Andrew Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[PPG+96] Tsvetomir P. Petrov, Anna Pogosyants, Stephen J. Garland, Victor Luchangco, and Nancy A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In *Proc. 9th Int. Conf. on Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV'96)*, pages 29–44. Chapman & Hall, 1996.

[PS97] Jan Philipps and Oscar Slotosch. Case study avionics: Verification in Spin. 1997. Internal Report, Technische Universität München, Institut für Informatik, Germany.

[Que98] *The Quest Project.* 1998. http://www4.informatik.tu-muenchen.de/proj/quest.html.

[Reg94] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technische Universität München, 1994.

[Reg95] Franz Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer-Verlag, 1995.

[Rom96] J.M.T. Romijn. Tackling the RPC-Memory specification problem with I/O automata. In *Formal Systems Specification – The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*, pages 437–476. Springer-Verlag, 1996.

[Rum97] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* PhD thesis, Technische Universität München, 1997.

[RV96] J. Romijn and F. Vaandrager. A note on fairness in I/O automata. *Information Processing Letters*, 59(5):245–250, 1996.

[Sab88] Krishan Sabnani. An algorithmic technique for protocol verification. *IEEE Transactions on Communications*, 36(8):924–930, 1988.

[SAGG+93] Joergen F. Soegaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Proc. 5th Int. Conf. Computer-Aided Verificatio (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

[SG90] D. Scott and C. Gunter. Semantic Domains and Denotational Semantics. In *Handbook of Theoretical Computer Science*, chapter 12, pages 633 – 674. Elsevier Science Publisher, 1990.

[Sif83] Joseph Sifakis. Property preserving homomorphsims of transition systems. In E.M. Clarke and D. Kozen, editors, *4th Workshop on Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 458–473. Springer-Verlag, 1983.

[SL95] Roberto Segala and Nancy Lynch. Probalbilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[SLL93] J.F. Sogaard-Andersen, N.A. Lynch, and B.W. Lampson. Correctness of communication protocols – A case study. Technical Report MIT/LCS/TR-589, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1993.

[SLW95] B. Steffen, K.G. Larsen, and C. Weise. A constraint oriented proof methodology based on modal transition systems. In *Proc. 1st Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 17–40. Springer-Verlag, 1995.

[SS95] B. Schätz and K. Spies. *Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik*. 1995. Technischer Bericht, TU München, Institut für Informatik, SFB-Bericht Nr. 342/16/95 A.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 131–192. Elsevier Science Publishers B.V., 1990.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pages 322–331. IEEE Computer Society Press, 1986.

[Wen97] Markus Wenzel. Type classes and overloading in higher-order logic. In E.L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer-Verlag, 1997.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. Principles of Programming Languages*, pages 184–193. ACM Press, 1986.

[Wra89] G.C. Wraith. A note on categorical datatypes. In D.H. Pitt, A. Poigne, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer-Verlag, 1989.

[Wri92] J. von Wright. Mechanising the Temporal Logic of Actions in HOL. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Proc. 1991 Int. Workshop on the HOL Theorem Proving System and its Applications*, pages 155–159. IEEE Computer Society Press, 1992.

Index