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 – Part I: Temporal Logic and Abstraction –

Olaf Müller

TUM-I9911
Juni 99

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-06-I9911-50/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

A Verification Environment for I/O Automata

– Part I: Temporal Logic and Abstraction –

Olaf Müller*
Institut für Informatik, Technische Universität München, Germany.
Email: mueller@in.tum.de

Abstract

I/O automata are used to specify and reason about distributed, reactive systems. In this paper we extend standard I/O automata by a theory of abstraction. The intention is to combine theorem proving and model checking. Verifying both temporal properties and implementation relations is reduced to finite-state model checking. Even for livenness proofs merely simple first-order proof obligations remain for theorem proving. Furthermore, a methodology is developed which allows an incremental improvement of abstractions, thus increasing the potential for automation. The results are based upon the first linear-time temporal logic for I/O automata. In contrast to existing logics, formulas are evaluated over sequences of alternating states and actions which may be finite. In part II of this paper the realization of the entire theory is described using the theorem prover Isabelle and different model checking tools.

1 Introduction

I/O automata are a semantic model for reactive, distributed systems together with a tailored refinement concept. The model has originally been proposed by Lynch and Tuttle [15], subsequent developments are mainly due to Lynch and Vaandrager [16, 13, 8]. The method has already been successfully applied to the verification of several non-trivial case studies, ranging from communication protocols [24] and automated transit systems [6] to database applications [14].

In this paper we extend standard I/O automata by a theory of abstraction in order to lay the basis for an effective integration of deductive verification and model checking. In part II [20] the realization of a corresponding toolbox is described.

1.1 A Temporal Logic of Steps

The abstraction theory is based upon a linear-time temporal logic, called Temporal Logic of Steps (TLS), which allows to express properties over finite or infinite runs of I/O automata. Apart from its use for abstraction there are two benefits of TLS.

First, it can serve as a property specification language for I/O automata. Previously, only implementation relations between automata have been considered in I/O automata theory.

*Research supported by BMBF, *KorSys*

Temporal properties, however, are especially advantageous for model checking, which will be — in combination with abstraction techniques — one of the key subjects of this paper.

Second, temporal formulas can be used to specify liveness conditions for safe I/O automata. The game-theoretic treatment of liveness in standard I/O automata theory [8] defines liveness conditions just as sets of executions and gives no advice for how to specify and reason about 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. [22, 13]). Temporal logic, however, has been proven to be an adequate methodology for liveness proofs [7, 10].

A number of temporal logics for program verification have already been proposed, most notable are Lamport's Temporal Logic of Actions (TLA) [11] and the temporal logic developed by Manna and Pnueli [17]. 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. Both concepts, however, are essential characteristics of I/O automata theory. Thus, we have to provide the corresponding modifications.

1.2 A Theory of Abstraction

An important aim of this paper 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 [4, 12, 18, 9, 5, 25] 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 [4]. If h is a homomorphism (*abstraction function*), the abstraction is guaranteed to be sound, i.e. if the abstract system satisfies a property so does the original system.

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. This applies to abstraction functions as well, as they can be regarded as 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 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'

[1]This is the key idea of the usual abstractions e.g. in [4].

is always a direct mapping of the concrete execution α under h, 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. In addition, we propose a tailored abstraction methodology which enables the reuse of simple abstraction functions even for the liveness part and thus decreases the required intuition. It is based upon further techniques 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.

Abstraction rules are provided for properties expressed both as temporal formulas and as I/O automata. Therefore, we may always choose the more adequate description technique to describe system properties.

1.3 Structure of the Paper

The paper is organized as follows: In §2 we recall the basics of I/O automata theory [13]. In §3 we introduce TLS, in §4 we use it to specify live I/O automata. The abstraction theory is presented in §5, in §7 we conclude.

2 Basic Theory of I/O Automata

Safe and Fair I/O Automata. 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 $acts(S) \equiv in(S) \cup out(S) \cup int(S)$.

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. We write $s \xrightarrow{a}_A t$, or just $s \xrightarrow{a} t$ 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 $in(sig(A))$, etc.

Let us illustrate this model by the "running example" of this paper, a rough simplification of an industrial case study also performed in our framework [19].

Example 2.1
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 from the set $Alarms = \{PonR, Fuel, Eng, \ldots\}$ 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.

The corresponding I/O automaton $Cockpit_C$ can be modeled by a list, called *stack*, which is initially empty and makes the following transitions — given in the standard precondition/effect style —:

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

In this paper we assume that every I/O automaton A is equipped with a set of state variables $\mathcal{V}$, such that every state s is a mapping from state variables to their values. The set of all state mappings of $\mathcal{V}$ is called $\mathbf{St}(\mathcal{V})$. A state mapping s in represented as a list of pairs, e. g. $s = [(stack, [Fuel])]$ in the example above.

Execution Schemes. Having our temporal logic in mind we define the behaviors of I/O automata a little bit more general than usual [13]. Instead of *executions* we use *execution schemes*, which may contain a further stuttering action $\surd$ and, in addition, may consist of states and actions not necessarily associated to an I/O automaton.

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 $\alpha = s_0\, a_1 s_1 \ldots$ 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, where ∞ indicates the infinite scheme. Define the i-th *prefix* $\alpha|_i$ and the i-th *suffix* ${}_i|\alpha$ of α, for $0 \leq i \leq |\alpha|$, as $\alpha|_i \equiv s_0\, a_1 s_1 \ldots a_i s_i$ and

$$ {}_i|\alpha \quad \equiv \quad \begin{cases} s_i\, a_{i+1} s_{i+1} \ldots & \text{if } i < |\alpha| \\ s_{|\alpha|} & \text{if } \alpha \text{ is finite and } i = |\alpha| \end{cases} $$

Executions and Traces. An *execution fragment* of an I/O automaton A is an execution scheme $s_0\, a_1 s_1 \cdots$ over $(states(A), acts(A))$ such that for all i, $s_i \overset{a_{i+1}}{\to}_A s_{i+1}$. An *execution* is an execution fragment that begins with a start state of A. As $\surd \notin acts(A)$ executions of I/O automata do not contain any stuttering steps $s_i \surd s_i$. The set of all executions of A is denoted by $execs(A)$. 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 $s \overset{\gamma}{\Rightarrow}_A t$, 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*.

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 $s \overset{a}{\to}_A t$.

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 *fairexecs*(A). The set of all traces originating from fair executions of A are denoted by *fairtraces*(A).

Live I/O Automata. 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. See [8] for this *environment-freedom* condition which is ensured by setting up a game between the system and its environment: the system is environment-free if it can win the game no matter what moves the environment performs.

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).

Note that every fair I/O automaton induces a live I/O automaton, if for every set in *sfair*(A) holds: once it is enabled, it can only be disabled by the occurrence of a locally-controlled action [23].

Implementation Relations. Given two live I/O automata (C, L_C) and (A, L_A) with the same external actions, there is a *safe trace inclusion*, written as $C \preceq_S A$, iff $traces(C) \subseteq traces(A)$, and a *live trace inclusion*, written as $(C, L_C) \preceq_L (A, L_A)$, iff $traces(L_C) \subseteq traces(L_A)$.

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 $states(C) \times states(A)$, written as $C \leq_F A$, such that:

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

Forward simulations are correct, i.e. if $C \leq_F A$ then $C \preceq_S A$. All results of this paper hold for *backward simulations* as well. See [16] for their definition. Simulations induce a specific correspondence between executions, given by *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[2] $m : \{0, 1, \ldots, |\alpha|\} \to \{0, 1, \ldots, |\alpha'|\}$ such that

- $m(0) = 0$,

[2]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 ∞.

- $(s_i, t_{m(i)}) \in R$ for all $0 \leq i \leq |\alpha|$,
- $\mathit{trace}(b_{m(i-1)+1} \cdots b_{m(i)}) = \mathit{trace}(a_i)$ 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.

3 TLS: A Temporal Logic of Steps

The basic ingredients of our temporal logic will be *step predicates.* In order to handle actions explicitly, they depend not only on a variable v in a given state and its value in the successor state (refered to by v'), but also on a further variable act which is exclusively reserved for actions. Furthermore, we introduce a stuttering action $\surd$, which is disjoint from all relevant action signatures. See [19] for the formal definition of step predicates, which are exactly analogous to the actions from TLA [11] except from considerung an additional action component.

Here, we will introduce step predicates with the running example introduced before. Consider the automaton $\mathit{Cockpit}_C$. Let $\mathcal{V}$ be $\{\mathit{stack}\}$ and $\mathcal{A}$ be $\{\{\mathit{Alarm}(a)\} \cup \{\mathit{Ack}(a)\} \mid a \in \mathit{Alarms}\}$. Then a possible step predicate p over $(\mathcal{V}, \mathcal{A})$ could be $\mathit{stack} = [] \wedge \mathit{act} = \mathit{Alarm}(\mathit{Fuel}) \wedge \mathit{stack}' = [\mathit{Fuel}]$ which states that the alarm Fuel is added by the action Alarm to the empty alarm stack. Thus, p holds for the step (s, a, t), written as $(s, a, t) \models p$, if $s = [(\mathit{stack}, [])]$, $a = \mathit{Alarm}(\mathit{Fuel})$, and $t = [(\mathit{stack}, [\mathit{Fuel}])]$. Specific step predicates which depend on a single state or action only, are called *state* and *action* predicates, respectively. We write $s \models p$ and $a \models p$, respectively. Validity is denoted by $\models p$ for all kinds of step predicates.

We will now define the temporal formulas of TLS, which in contrast to step predicates refer to entire execution schemes.

Definition 3.1 (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 $(\mathbf{St}(\mathcal{V}), \mathcal{A})$. We define inductively what it means that a temporal formula P over $(\mathcal{V}, \mathcal{A})$ holds for α, written $\alpha \models P$. 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 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 ${}_1\lvert\alpha \models P$ $\Box$

Here the use of the stuttering action $\surd$ becomes clear. In order to handle finite computations, a single stuttering step $s_n \surd s_n$ is added at the end of all finite executions $s_0 a_1 s_1 \cdots s_n$. This is done in the definition of step predicates for execution schemes of length zero. Note that only one stuttering step is added, such that the resulting execution will still be finite. Intuitively, we thereby ensure that finite executions may possibly be continued to infinity.

In fact, we will show below that the evaluation of temporal formulas does not change when extending executions by infinite stuttering. Furthermore note that $\bigcirc P$ does not differ from P if $|\alpha| \leq 1$.

Definition 3.2 (Derived Temporal Formulas)
Given temporal formulas P and Q the following are temporal formulas as well.

$$\begin{array}{lllll}
\textbf{Boolean Operators}: & P \Rightarrow Q & \equiv & \neg(P \wedge \neg Q) \\
 & P \vee Q & \equiv & (\neg P) \Rightarrow Q \\
\textbf{Eventually}: & \Diamond P & \equiv & \neg \Box \neg P \\
\textbf{Leads To}: & P \leadsto Q & \equiv & \Box(P \Rightarrow (\Diamond Q))
\end{array}$$

□

For infinite execution schemes the meaning of TLS formulas is as one might expect from standard temporal logic. Note however 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 $s_n \surd s_n$. In particular the meaning of $\Box\Diamond P$ and $\Diamond\Box P$ coincides for finite executions: both express that P is true for the last state s_n.

Definition 3.3 (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 $(\mathbf{St}(\mathcal{V}), A)$. 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. 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. □

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

Definition 3.4 (Adding Stuttering Steps)
Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ be an execution scheme. Then

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

□

Theorem 3.5 (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 $\leadsto$.

4 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, a 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 traditional fairness sets of §2.

Definition 4.1 (Fair and Live I/O Automata via TLS)
A live I/O automaton (A, L) is described by specifying L in terms of a temporal formula F:

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

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

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

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 $\mathit{SF}_A(\Lambda)$:

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

$\Box$

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 [8].

The fairness notions induced by WF_A and SF_A conincide with the fairness notion of traditional I/O automata theory.

Lemma 4.2 (Correspondence of Fairness Notions)
Let A be a safe I/O automaton and L a liveness condition which is induced by the TLS formula $F = \bigwedge_{i=1}^{n} \mathit{WF}_A(\Lambda_i) \wedge \bigwedge_{j=1}^{m} \mathit{SF}_A(\Lambda'_j)$. Furthermore, define B to equal A for the safety part, i.e. $X(B) = X(A)$ for every component $X \in \{\mathit{sig}, \mathit{states}, \mathit{start}, \mathit{steps}\}$ of A, and assume that B is fair with $\mathit{wfair}(B) = \bigcup_{i=1}^{n} \Lambda_i$ and $\mathit{sfair}(B) = \bigcup_{j=1}^{m} \Lambda'_j$. Then the fair executions of B induced by the fairness sets $\mathit{wfair}(B)$ and $\mathit{sfair}(B)$ equal those induced by the fairness formula F of A:

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

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. $\Box$

4.1 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 forward simulations to *live forward simulations*, which means that they in addition transfer liveness from the implementation to the specification automaton.

Definition 4.3 (Live Forward 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 we say that $(C, F_C) \leq_F^L (A, F_A)$ holds iff there is relation S such that if $C \leq_F A$ 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 $\alpha' \models F_A$. $\Box$

Theorem 4.4 (Soundness of Live Forward 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, if $(C, F_C) \leq_X^L (A, F_A)$ for some $X \in \{F, R, iB\}$ then $(C, F_C) \preceq_L (A, F_A)$.

Proof.
Essentially Lemma 6.18 in [8] except for the fact that liveness conditions are specified indirectly in terms of temporal formulas, together with Lemma 4.2. $\Box$

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 $\alpha' \models F_A$. 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 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 4.5 (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 ${}_i|\alpha \models \Box a \notin \Lambda$. 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 [8]) 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 4.6 (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 (§2) 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 4.7
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 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

$$\begin{array}{l|ccc}
\alpha' \in \mathit{execs}(A) & \Diamond\Box \mathit{Enabled}_A(\Lambda) & \longrightarrow & \Box\Diamond\ a \in \Lambda \\
 & \downarrow & & \uparrow \\
 & \text{(Lemma 4.6)} & & \text{(Lemma 4.5)} \\
 & \downarrow & & \uparrow \\
\alpha \in \mathit{execs}(C) & \Diamond\Box \mathit{Enabled}_C(\Lambda) & \longrightarrow & \Box\Diamond\ a \in \Lambda
\end{array}$$

Figure 1: Schematic example of a simulation proof involving fairness

$\alpha' \models \Diamond\Box \mathit{Enabled}_A(\Lambda)$. Here, Lemma 4.6 allows us to switch from α' to α, thus we get $\alpha \models \Diamond\Box \mathit{Enabled}_C(\Lambda)$, provided that the state predicate $\mathit{Enabled}_A(\Lambda)$ implies $\mathit{Enabled}_C(\Lambda)$ for all $(s,t) \in R$. From the assumption $\alpha \models \mathit{WF}_C(\Lambda)$ we get $\alpha \models \Box\Diamond a \in \Lambda$. Here, Lemma 4.5 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 4.6 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 next.

5 Abstraction Theory

In this section we first show how properties about executions can be reduced to properties about steps, if an appropiate abstraction function exists (in §5.1). Then, abstraction rules are derived for properties both expressed as temporal formulas and as automata (in §5.2 and §5.4, respectively). In §5.3 a methodology is proposed that is directed towards more automation of abstraction.

5.1 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.

Definition 5.1 (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 $\mathit{states}(C)$ and $\mathit{acts}(C)$ and $\alpha' = t_0 b_1 t_1 \ldots$ be an execution scheme over $\mathit{states}(A)$ and $\mathit{acts}(A)$. Furthermore let h be a function from $\mathit{states}(C)$ to $\mathit{states}(A)$. Then α and α' are said to *functionally correspond* under h, iff $t_i = h(s_i)$ and $b_i = a_i$ for all i. $\Box$

Note that for every execution scheme α over $\mathit{states}(C)$ and $\mathit{acts}(C)$ and a given function h there is exactly one corresponding execution scheme α' over $\mathit{states}(A)$ and $\mathit{acts}(A)$ 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. These implications will be reduced to the step level afterwards.

Definition 5.2 (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 iff 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 of P w.r.t. h, iff for all execution schemes of 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. □

Now, we define weakening and strengthening on the step level as well.

Definition 5.3 (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 step predicate over $\mathcal{T} \times \mathcal{A} \times \mathcal{T}$ and P be a step predicate over $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$. Then, Q is a *step*

- *weakening* of P iff $\forall s, t \in \mathcal{S}, a \in \mathcal{A}.\ (s, a, t) \models P \Rightarrow (h(s), a, h(t)) \models Q$.
- *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 called *local* weakenings/strengthenings. □

In the following we show how global weakenings and strengthenings can be reduced 1) from temporal formulas to step predicates and 2) from automata to transitions. We start with temporal formulas.

Theorem 5.4 (Localizing Temporal Weakenings and Strengthenings)
Let P be a temporal formula built from step 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. □

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 step predicates occurring in them. We proceed with reducing automaton weakenings. Their counterpart on step level are *abstraction functions*:

Definition 5.5 (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 $C \leq_A A$, if there is an abstraction function from C to A. $\Box$

Note that the second condition of an abstraction function implies that $Q(s, a, t) = s \xrightarrow{a}_A t$ is a step weakening of $P(s, a, t) = s \xrightarrow{a}_C t$.

Theorem 5.6 (Localizing Automaton Weakenings)
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, if $C \leq_A A$ via h, then A is an automaton weakening of C w.r.t. h.

Proof.
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 $h(s_0) \in start(A)$ and $h(s_i) \xrightarrow{a}_A h(t_i)$ for all $i \geq 0$. This gives the desired weakening property immediately. $\Box$

Abstraction functions can easily be extended to liveness.

Definition 5.7 (Live Abstraction Functions)
Let (C, F_C) and (A, F_A) be live I/O automata with the same set of actions. Then,

$$(C, F_C) \leq_A^L (A, F_A) \quad \text{iff} \quad \begin{array}{l} C \leq_A A \text{ via } h \text{ and} \\ F_A \text{ is a global weakening of } F_C \text{ w.r.t. } h \text{ for some } h. \end{array} \quad \Box$$

Theorem 5.8 (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, if $(C, F_C) \leq_A^L (A, F_A)$, then $(C, F_C) \preceq_L (A, F_A)$.

Proof.
Let $\alpha \in exec(C)$ and $\alpha \models F_C$. Then we have to show that $c_h(\alpha) \in exec(A)$, $c_h(\alpha) \models F_A$, and $trace(c_h(\alpha)) = trace(\alpha)$. The first is ensured by Lemma 5.6, 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. $\Box$

5.2 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, Theorem 5.4 has to be used to reduce those weakenings/strengthenings to local ones.

Theorem 5.9 (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 $C \leq_A A$ via h, and let P^+ be a temporal strengthening of P w.r.t. h. Then:

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

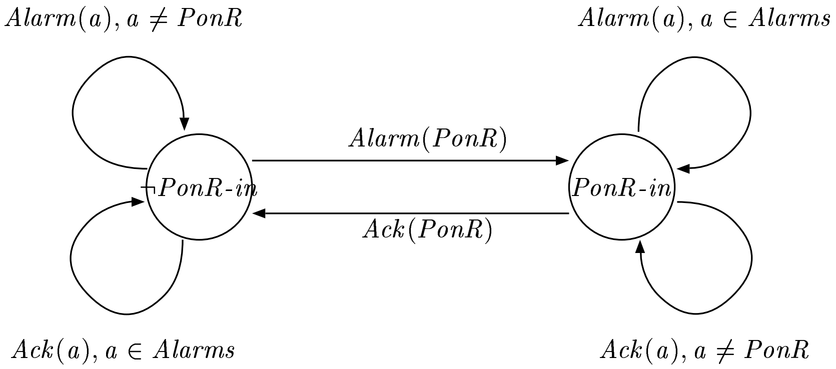

Figure 2: Transitions of $Cockpit_A$

Proof.
Let $\alpha \in exec(C)$. Then it has to be shown that P holds for α. As $C \leq_A A$ via h, we get by Lemma 5.6 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 $\alpha \models P$. $\square$

Example 5.10
The cockpit alarm system features 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 properties we want to prove about $Cockpit_C$ are the following:

$P_1 \equiv \Box(act = Alarm(PonR) \Rightarrow \bigcirc PonR \in stack)$
"Whenever PonR arrives, it is immediately stored in the stack"

$P_2 \equiv \Box(act \neq Alarm(PonR)) \Rightarrow \Box\neg(PonR \in stack)$
"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 $PonR\text{-}in$, initially *false*, which indicates if $PonR$ is stored (cf. Fig. 2 and 3).

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

---

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

The abstraction function h is defined as $h([(stack, l)]) \equiv [(PonR\text{-}in, PonR \in l)]$. As $Cockpit_A$ has been designed already with h in mind, the property $Cockpit_C \leq_A Cockpit_A$ is easily established[3]. Therefore it remains to show that the corresponding abstract properties P_i^+ for $Cockpit_A$, defined as

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

are temporal strengthenings of the concrete P_i. Consider P_1 and P_1^+. Then the obligation is reduced by the localizing theorem for abstractions (Theorem 5.4) to the state property

$$s \models PonR \in stack \quad \Rightarrow \quad h(s) \models PonR\text{-}in$$

which is trivial by the definition of h. 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 (as done in [19]).

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

Theorem 5.11 (Abstraction for Live I/O Automata)
Let (C, F_C), (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 $(C, F_C) \leq_A^L (A, F_A)$ 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 $(C, F_C) \leq_A^L (A, F_A)$ we get by Lemma 5.6 that A is an automaton weakening of C. Therefore, $c_h(\alpha) \in exec(A)$. By definition of $\leq_A^L$ 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 (cf. Fig. 3) 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 5.12
Let us try to prove the following property about $Cockpit_C$.

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

"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. Thus, a fairness assumption for Ack is needed, even if all actions apart from Ack are excluded, as

[3]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 $\neg PonR\text{-}in$.

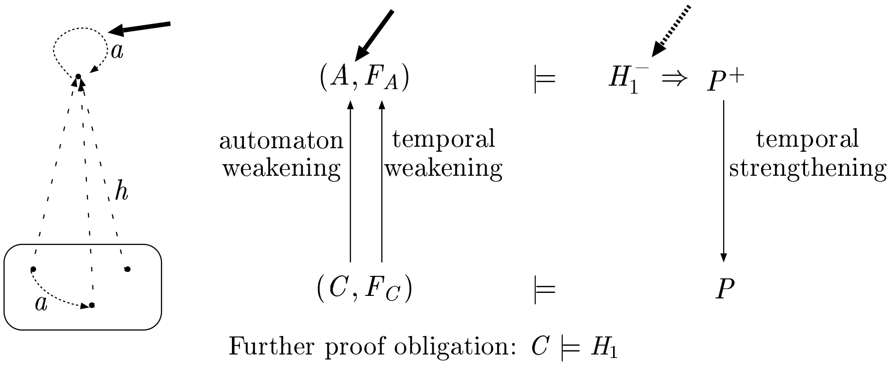

Figure 3: Motivation for H_1 in Abstraction Rule (Abs-P_3)

done by the assumption $\Diamond\Box(\forall a'.\ a \neq Alarm(a'))$. This assumption, however, permits infinite executions of the form $s_0\, a_1\, s_1 \cdots s_n \surd s_n \surd s_n \cdots$ which evidently are not fair. Therefore, we define

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

and try to prove $(Cockpit_C, F_C) \models P_3$ with the same abstraction h as before First, we define the corresponding abstract properties for P_3 and F_C:

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

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. 2).

- F_A is not a temporal weakening of F_C.
  To explain this we use the structural abstraction rules of Theorem 5.4. Then it remains to show that $Enabled_A\,(\bigcup_{a \in Alarms} Ack(a))$ implies the formula $Enabled_C\,(\bigcup_{a \in Alarms} Ack(a))$ for all states being mapped under h. This is, however, not true, 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 maps $[\,]$ to $false$.

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. 3 and Fig. 4 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 abstraction H_1^- can be used as further assumption for P^+.

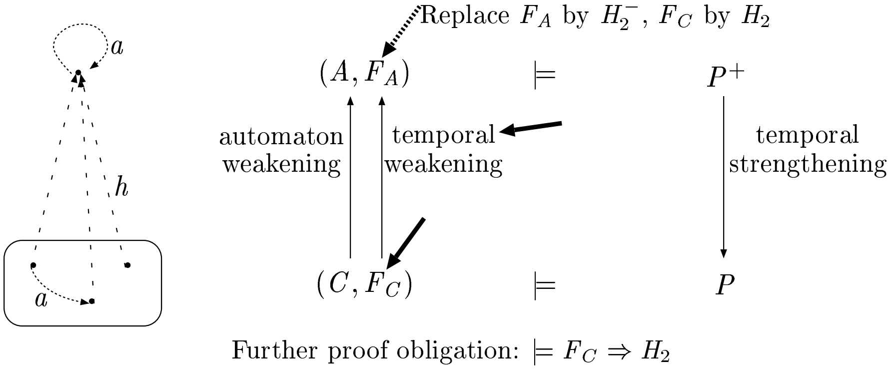

Figure 4: Motivation for H_2 in Abstraction Rule $(Abs\text{-}P_3)$

- 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 5.13 (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 $(C, H_2) \leq_A^L (A, H_2^-)$ 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.
Essentially the same as for Theorem 5.11. $\square$

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.

Example 5.14
With the rule $(Abs\text{-}P_3)$ we are now able to prove $(Cockpit_C, F_C) \models P_3$ with the abstraction function h. First, we define $H_1 \equiv \Box\Diamond(\exists a.\ act = Ack(a)) \Rightarrow \Box\Diamond(\exists a.\ act = Alarm(a))$ which falsifies the assumption $\Diamond\Box(\forall a.\ act \neq Alarm(a))$ 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. H_1 is an instance of the more general *well-founded formulas* as defined in [18]. The general idea is to define an order on automata transitions and then to 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. Therefore, we may take H_1^- to strengthen the abstract liveness assumption. Second, we define

$$\begin{array}{lcl} H_2 & \equiv & \Diamond\Box(PonR \in stack) \Rightarrow \Box\Diamond act \in \bigcup_{a \in Alarms} Ack(a) \\ H_2^- & \equiv & \Diamond\Box PonR\text{-}in \Rightarrow \Box\Diamond act \in \bigcup_{a \in Alarms} Ack(a) \end{array}$$

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 $PonR\text{-}in$ 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, rule ($Abs\text{-}P_3$) allows us now to reduce $(Cockpit_C, F_C) \models P_3$ to the desired $(Cockpit_A, H_2^-) \models H_1^- \Rightarrow P_3^+$, which is indeed satisfied and can be proved by the STeP model checker (see [19]). 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 the following abstract automaton $Cockpit'_A$, which is augmented w.r.t. $Cockpit_A$ by a boolean variable $Other\text{-}in$, initially false, indicating whether an alarm $a \neq PonR$ is stored:

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

The corresponding abstraction function h_2 is defined as:

$$h_2([(stack, l)]) \quad \equiv \quad [(PonR\text{-}in, PonR \in l), (Other\text{-}in, \exists a \neq PonR.\ a \in l)]$$

This abstraction function h_2 is distinct 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 $H_2^- = F_A$.

5.3 An Abstraction Methodology

The last example shows that the abstraction rule ($Abs\text{-}P_3$) can be regarded as proof support for the following methodological decision every proof designer must take if the first abstraction attempt fails: either one chooses a more complicated abstraction function — together with a new associated abstract model — or one stays with a simpler function and has to presume a further assumption about the environment instead. 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.

5.4 Abstraction Rules for Automata

Whereas §5.2 dealt with proof obligations of the form $(C, F_C) \models P$ with P being a TLS formula, this section shows how implementation relations can be abstracted, i.e. proof obligations of the form $(C, F_C) \preceq_L (P, F_P)$.

Theorem 5.15 (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 $C \leq_A A$ and $P^+ \leq_A P$. Then:

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

Proof.
As every abstraction function represents a particular forward simulation, we get $C \leq_S A$ and $P^+ \leq_S P$. The correctness result for forward simulations 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 5.16
Consider once more the alarm system $Cockpit_C$ and its abstract counterpart $Cockpit_A$ under the abstraction h. 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: $hd(stack) = a$	**pre**: $a = PonR \Rightarrow PonR\text{-}in$

It is easily shown that $Cockpit_C \leq_A Cockpit_A$ w.r.t. h 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-in*, which is initially false. The actions of P_4 are:

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.

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 [1] (see part II [20]).

Theorem 5.17 (Abstraction for Live I/O Automata)
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 $(C, F_C) \leq_A^L (A, F_A)$ and $(P^+, F_{P^+}) \leq_A^L (P, F_P)$. 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 5.8 we get $(C, F_C) \preceq_L (A, F_A)$ and $(P^+, F_{P^+}) \preceq_L (P, F_P)$ from the assumptions $(C, F_C) \leq_A^L (A, F_A)$ and $(P^+, F_{P^+}) \leq_A^L (P, F_P)$. Then, the result follows immediately from the transitivity of $\preceq_L$. □

Theorem 5.18 (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 $(C, H_2) \leq_A^L (A, H_2^-)$ and $(P^+, F_{P^+}) \leq_A^L (P, F_P)$. 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 5.17 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 $\preceq_L$. □

6 Related Work

As far as we now, there is no temporal logic for I/O automata until now. But of course, TLS is closely related to TLA [11]. However, there are some further remarkable differences.

First, TLA is a uniform logical setting whereas our hybrid approach is able to treat both liveness and safety in a tailored way, namely by automata theory and TLS, respectively.

Second, 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.

As far as we now, there is no abstraction theory for I/O automata until now. In contrast to the standard literature on abstraction in general [4, 12], our theory covers implementation relations as well and provides specific support for liveness abstractions. Furthermore, our methodology for abstraction appears to be new.

Most closely related to our work are probably the abstraction rules for TLA by Merz [18]. His approach, being in a uniform logical setting, makes 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 to use temporal formulas merely for liveness conditions.

Orthogonal to our theory are approaches that deal with data abstraction and data independence [25, 26]. 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 [3], carries over to I/O automata.

7 Conclusion

We defined TLS as a temproal logic tailored for I/O automata which can be used to specify properties over executions. As a further result we get a verification framework which is, at least 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. 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. 4.5 – 4.6). Previously, this has been done without temporal logic either by complicated reasoning about index mappings [13] or in a more informal way [22].

Furthermore, we presented an abstraction theory for the effective verification of I/O automata via a combination of theorem proving and model checking. The abstraction rules cover both temporal properties and implementation relations and are especially suited for handling liveness. They are accompanied by a methodology that reduces the required intuition by making reuse of simple abstraction functions.

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.

The entire approach has been implemented in the theorem prover Isabelle [21], together with appropiate translations to the STeP [2] model checker and μcke [1]. Within this toolbox a cockpit alarm system of industrial size, which could not be handled by traditional model checking, has been successfully verified [19]. See part II for details of the toolbox realization.

References

[1] 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.

[2] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification: 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, 1996.

[3] 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.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. 19th ACM Symp. Principles of Programming Languages*, pages 343–354. ACM Press, 1992.

[5] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):22–43, 1997.

[6] E. Dolginova and N. 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.

[7] D. Gabbay, A. Pnueli, S. Shelah, and J. 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.

[8] R. Gawlick, R. Segala, J. Sogaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Laboratory for Computer Science, MIT, Cambridge, MA., December 1993.

[9] R. Kurshan. Reducibility in analysis of coordination. In K. Varaiya, editor, *Discrete Event Systems: Models and Applications*, volume 103 of *Lecture Notes in Control and Information Science*, pages 19–39. Springer-Verlag, 1987.

[10] L. 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, Sept. 1983. IFIP, North-Holland.

[11] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[12] 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.

[13] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[14] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.

[15] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[16] N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[17] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, NY, 1995.

[18] S. 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, Dec. 1997. Springer-Verlag.

[19] O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Institut für Informatik, Technische Universität München, 1998.

[20] O. Müller. A verification environment for I/O automata — part II: Theorem proving and model checking. 1999. submitted.

[21] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[22] J. 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.

[23] J. Romijn and F. Vaandrager. A note on fairness in I/O automata. *Information Processing Letters*, 59(5):245–250, 1996.

[24] J. Sogaard-Andersen, N. Lynch, and B. 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.

[25] B. Steffen, K. 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.

[26] P. 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.