T U M

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

Proceedings of the 5th International Workshop on Verification In New Orientations

Edited by: Maximilian Frey

TUM-I9720
April 97

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-04-I9720-80/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Proceedings of the
5th International Workshop on

Verification In New Orientations

(VINO'96)

May 16 -19
Rottach Egern, Germany

Edited by:
Maximilian Frey

Contents

Preface

VINO'96 was the fifth in a series of workshops. Previous workshops have been held in Sand in Taufers/Campo Tures, Italy and Maribor, Slovenia. The theme of VINO'96 was "New Directions in Formal Specification, Design and Validation of Software Systems". Recently, the growing power of computer systems as well as the increasing use of network facilties allow to develop more and more complex software which must be of high reliabilty. Formal methods support developers to get a precise description of the requirements and to validate the software during all phases of its development. Formal methods are necessary for an automatic support of verification and validation. The aim of VINO'96 is to give a forum for discussion of problems and presentations of recent results in the field of formal techniques and their theoretical background.

Papers were invited for the following four topics:

- Model Checking of Distributed Systems
- Modal and Temporal Logics
- Distributed Systems Design
- Functional Programming Paradigms

The presentations were divided into four sessions following the topics. The presentations were limited to 35 minutes with 10 minutes discussion.

The papers of all presentations are included in the workshop proceedings which contain seven high level papers indicating new directions in the field of formal methods and their therotical background.

VINO'96 and all previous workshops offered a great variety of social events which offered the participants the ability to bulid small groups for discussing special problems and results in an open atmosphere.

We hope that all participants in the workshop and all readers of this proceedings will find valuable new ideas which will open new research directions.

Bremen and Munich, 11.3.97

Dr. H. Schlingloff

M. Frey

Modelling Message Buffers with Binary Decision Diagrams

Bernd–Holger Schlingloff
Institut für Informatik der TU München
Orleansstr. 34, 81667 München
schlingl@informatik.tu-muenchen.de

Abstract

Binary decision diagrams (BDDs, [3]) have been recognized as an extremely efficient data structure for the representation of transition relations in the verification of finite-state reactive systems. With BDDs, it is possible to represent relations over domains with more than 2^{100} elements ([4]), provided the represented relation is well-structured. Asynchronous parallel systems such as communication protocols often use implicit or explicit buffering of messages which are sent between the processes. In these notes, we analyze the complexity of various possibilities to model the transition relation of a bounded buffer with BDDs, and discuss alternative approaches to this problem.

1 Binary Decision Diagrams

To make these notes self-contained, we quickly describe the symbolic representation of sets and relations with BDDs. For a detailed survey, the reader is referred to [3]. Consider a sequence of variables $V \triangleq (v_1, ..., v_k)$ over domains $(D_1, ..., D_k)$, where each D_i is finite. An *ordered decision diagram* (ODD) or *deterministic branching program* for V is a tuple $(N, \mathcal{L}, E, n_0)$, where

- N is a finite set of nodes,
- $\mathcal{L} : N \to V \cup \{\top, \bot\}$ is a labelling of nodes,
- $E \subset N \times D \times N$ is a set of edges ($D = \bigcup_i D_i$), and
- n_0 is the initial node.

The following conditions are imposed:

- E is functional on D_i: If $\mathcal{L}(n) = v_i$, then for each $(n, d, n') \in E$ it holds that $d \in D_i$, and for each $d \in D_i$ there is exactly one n_d such that $(n, d, n_d) \in E$, and
- E is acyclic: If $(n, d, n') \in E$ with $\mathcal{L}(n) = v_i$ and $\mathcal{L}(n') = v_j$, then $i < j$.

It is easy to see that this definition is equivalent to the one given, e.g., in [3]. Any ODD accepts (defines) a subset of $(D_1 \times ... \times D_k)$ via the following definition:

$$(N, \mathcal{L}, E, n_0) \models (d_1, ..., d_k) \quad \text{if} \quad (N, \mathcal{L}, E, n_0) \models_1 (d_1, ..., d_k).$$

In this definition, the notion $\models_m$ is declared by:

$(N, \mathcal{L}, E, n) \models_m (d_1, ..., d_k)$ if

- $\mathcal{L}(n) = \top$, or
- $\mathcal{L}(n) = v_i$ and $m < i$ and $(N, \mathcal{L}, E, n) \models_{m+1} (d_1, ..., d_k)$, or
- $\mathcal{L}(n) = v_i$ and $m = i$ and $(n, d_m, n') \in E$ and $(N, \mathcal{L}, E, n') \models_{m+1} (d_1, ..., d_k)$.

In other words, given a specific tuple, it can be determined whether it belongs to the set represented by an ODD by traversing its edges according to the components of the tuple.

When drawing ODDs, we usually omit the node labelled $\perp$ and all edges leading to it. For example, the ODD with two variables v, v' over $D_1 = D_2 = \{a, b, c, d\}$ given in Figure 1 below represents the set of tuples $\{(a, a), (a, b), (a, c), (a, d), (b, b), (b, d), (c, c), (c, d), (d, a), (d, d)\}$. *Binary* decision diagrams (BDDs) are ODDs where all domains are $\{0, 1\}$. Given any ODD, there exists a BDD of the same order of size which represents the same set: Choose any binary encoding of the domains, and replace each m-ary branch by a $\log m$-depth binary decision tree. Thus, in practice only BDDs are used; ODDs can be understood as abbreviations of the respective binary encoded BDDs. For example, choosing the encoding $a \mapsto 00$, $b \mapsto 10$, $c \mapsto 01$, and $d \mapsto 11$, the BDD given in the right half of Figure 1 represents the same set as the respective ODD on its left.

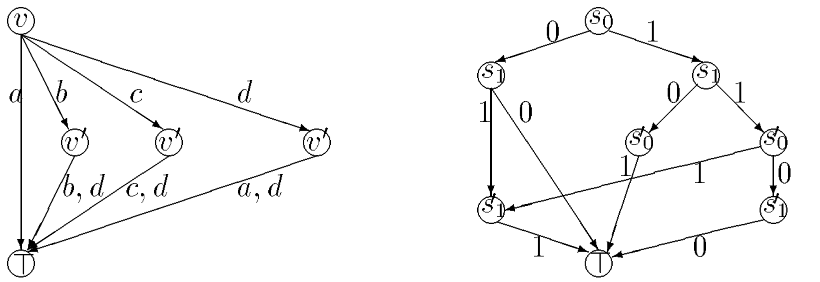


Figure 1: An ordered decision diagram and its binary encoding

The size of an ODD is the number of nodes it consists of. For a given ordering of the domains, and any set of values, there is a unique minimal ODD representing this set of values. The size of this minimal representation is not dependent on the size, but only on the structure of the represented set of values. E.g., the empty set and the set of all tuples both have an ODD representation of size one.

A *relation* is a subset of the product of its domain and range; therefore a relation $R : D \leftrightarrow D'$ can be represented by an ODD with two variables (over D and D', respectively), or by a BDD with $\lceil \log |D| \rceil + \lceil \log |D'| \rceil$ boolean variables. Our example BDD is a canonical representation of the relation whose matrix and graph are given in Figure 2. In general, the BDD representation of a relation can be much smaller than the representation by matrices or graphs.

$s'_0 s'_1$	a	b	c	d
	s_{00}	s_{01}	s_{00}	s_{01}
$s_0 s_1$	s_{10}	s_{10}	s_{11}	s_{11}
$a = s_{00}s_{10}$	x	x	x	x
$b = s_{01}s_{10}$		x		x
$c = s_{00}s_{11}$			x	x
$d = s_{01}s_{11}$	x			x

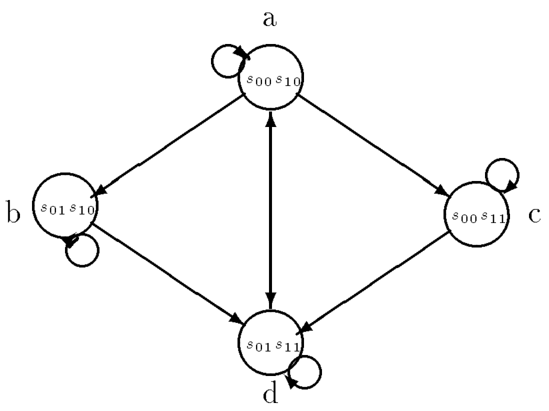


Figure 2: Matrix and graph of the encoded relation

Given a process P with state space D. Then the *transition relation* of P is a subset of $D \times D$. If P consists of k parallel processes $P_1, ..., P_k$ with state spaces $D_1, ..., D_k$, then the global state space of P is $D_1 \times \cdots \times D_k$. Therefore the transition relation can be described by $2k$ variables $s_1, ..., s_k, s'_1, ..., s'_k$, where s_i and s'_i are over domain D_i and describe the current and next state of process P_i. Again, if each D_i has up to m states, the global transition relation has up to m^{2k} elements and can be described by a BDD over $2k \cdot \lceil \log |m| \rceil$ boolean variables. For example, consider the elementary net of Figure 3; it models two processes synchronizing on a common transition. The states of the first process are $D_0 = \{s_{00}, s_{01}\}$, the states of the second are $D_1 = \{s_{10}, s_{11}\}$. Since these domains are binary, we can use boolean variables s_0, s_1, s'_0, s'_1 to describe the current and next state of the processes. The global states are $a \triangleq (s_{00}, s_{10})$, $b \triangleq (s_{00}, s_{11})$, $c \triangleq (s_{01}, s_{10})$, and $d \triangleq (s_{01}, s_{11})$. In state d, either both processes idle or both processes synchronize and go to state a; in each other state, process P_i can either idle or make a step from s_{i0} to s_{i1}, independently of the other process. The transition relation of this system is the one represented by our example.

The set of *reachable states* of a system is the image set of the initial state(s) under the reflexive transitive closure of the transition relation. With BDDs, the transitive closure of a relation usually is calculated as the smallest fixed point

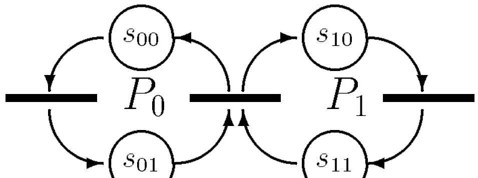


Figure 3: An elementary net model of synchronization

of the recursive equation $R^* = \mathrm{I} \cup R;R^*$. Relational composition is calculated by the definition $xR;Sy$ iff $\exists z(xRz \wedge zRy)$, and existential quantification over finite (binary) domains is replaced by a disjunction of the possible values of the domain.

Therefore, to calculate the set of reachable states with BDDs it is necessary to represent the complete transition relation. Since BDDs are graphs with a nonlocal connection structure, usually it is not possible to use virtual storage for BDD nodes; present technology limits the number of BDD nodes representing a transition function to approx. 10^6. The size of the BDD representation of the reachable states or reflexive transitive closure of a relation is often totally unrelated to the size of the representation of the relation itself; in our example, the transitive closure is the universal relation, and thus all states are reachable, with a BDD representation of size 1.

However, the size of a BDD crucially depends on the number and ordering of variables. In our example, consider the two processes as producer and consumer of messages which are passed at the synchronization step via handshake. That is, each process has an additional variable, m_0 and m_1, which are both over a domain $\mathcal{M}$ of, e.g., 4 messages $\{nil, x_1, x_2, x_3\}$. Process P_0 produces a message, i.e. sets variable m_0 to an arbitrary non-nil value, in the transition from s_{01} to s_{00}. On transition from (s_{00}, s_{10}) to (s_{01}, s_{11}) the value of m_0 is transferred to m_1, and m_0 is reset to nil. Process P_1 consumes (resets) variable m_1 in the transition from s_{11} to s_{10}. On idling transitions, the value of the message-variables is stable. The SMV-code (for SMV, see [9]) for this system is given in Figure 4, and the resulting BDD for variable ordering $(s_0, s_0', s_1, s_1', m_0, m_0', m_1, m_1')$ is shown in Figure 5.

As can be seen, the size of this BDD is linear in the number $m \triangleq |\mathcal{M}|$ of possible messages. In this example, the linear complexity is caused only by "local diamonds", i.e., nodes branching into m successor nodes, which again join into one successor. This structure arises by the copying instructions `next(m0)=m0`, `next(m1)=m1` and `next(m1)=m0`. Variables m_0 and m_1 can be seen as consisting of w boolean variables $m_{01}...m_{0w}$ and $m_{11}...m_{1w}$, where $w \triangleq \lceil \log m \rceil$ is the *message width*. If we interleave the order of these variables, i.e., use variable ordering

```
MODULE main
VAR s0 : boolean; s1 : boolean; m0 : {nil,x1,x2,x3}; m1 : {nil,x1,x2,x3};
INIT    (s0 = 0 & s1 = 0)
TRANS   (s0 = 0 & s1 = 1 -> next(s0) = 0)
&       (s0 = 1 & s1 = 0 -> next(s1) = 0)
&       (s0 = 0 & s1 = 0 -> next(s0) = 0 & next(s1) = 0 |
                                next(s0) = 1 & next(s1) = 1)
& (s0 = 1 & next(s0) = 0 -> next(m0) in {x1,x2,x3})          -- produce
& (s0 = 0 & next(s0) = 1 -> next(m0) = nil)                  -- reset
& (s0 = next(s0)         -> next(m0) = m0)                   -- stable
& (s1 = 0 & next(s1) = 1 -> next(m1) = m0)                   -- transfer
& (s1 = 1 & next(s1) = 0 -> next(m1) = nil)                  -- consume
& (s1 = next(s1)         -> next(m1) = m1)                   -- stable
```

Figure 4: SMV-code for message passing between two processes

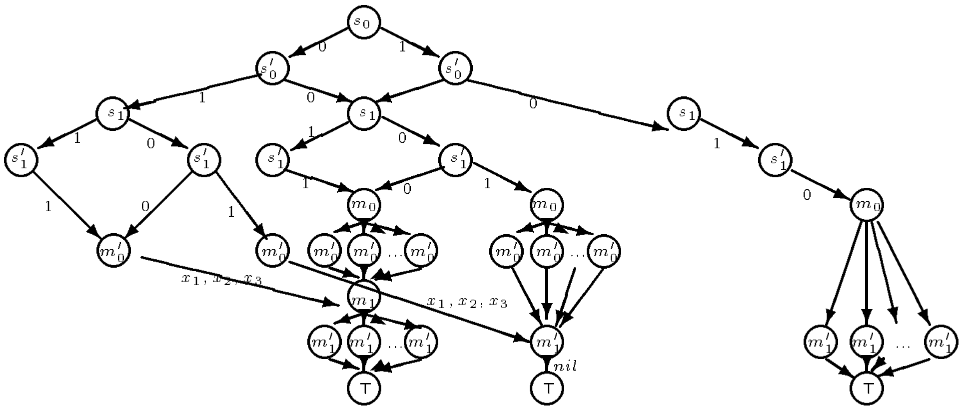


Figure 5: BDD for synchronous message passing

(m_{01}, m'_{01}, m_{11}, m'_{11}, ..., m_{0w}, m'_{0w}, m_{1w}, m'_{1w}), local diamonds are represented with complexity linear in w, see Figure 6. Thus, for the ordering (s_0, s'_0, s_1, s'_1, m_{01}, m'_{01}, m_{11}, m'_{11}, ..., m_{0w}, m'_{0w}, m_{1w}, m'_{1w}), the BDDs for the above SMV-code are logarithmic in m.

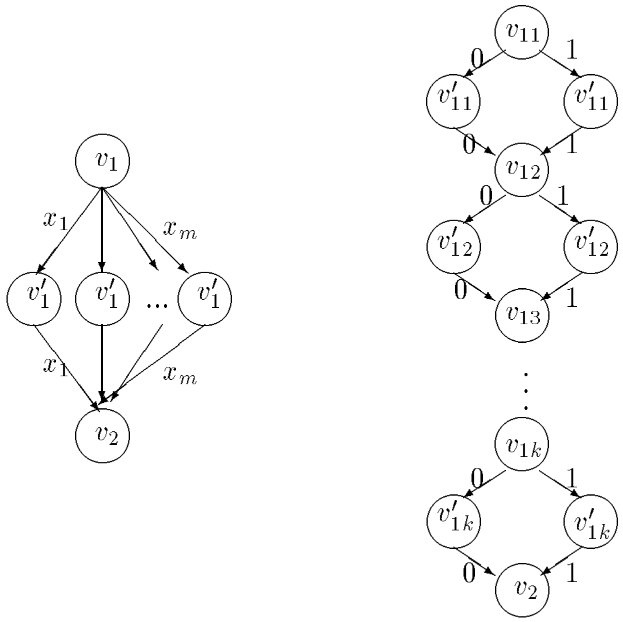


Figure 6: Interleaved encoding of a local diamond

2 Modelling of Message Buffers

Distributed parallel processes often use asynchronous (buffered) communication. Asynchronous message passing can be modelled with global variables by introducing a separate buffer process for each communication line. In many systems, the amount of messages which can be buffered is finite; in such systems buffer overflow often indicates erroneous behaviour of the system. For a fixed message alphabet $\mathcal{M} \triangleq \{nil, x_1, ..., x_{m-1}\}$, the formal specification of a bounded buffer of length n with input and output variables i and o over $\mathcal{M}$ can be given by the transition table 1.

In the right half of this table, an empty entry means that the respective variable is set by the environment. An input value of nil in i indicates that there is no message to be sent; in this case the next value of i is determined by the producer. If this process has put a non-nil value $x \in \mathcal{M}$ into i, then this value is appended to the buffer, and i is reset to nil. The last line indicates a condition of buffer overflow: If a message is to be sent with the message buffer already filled,

i	b	o		i'	b'	o'
nil	$\langle\rangle$	nil			$\langle\rangle$	nil
x	$\langle\rangle$	nil		nil	$\langle\rangle$	x
nil	$\langle x_1, \ldots, x_\nu \rangle$	nil			$\langle x_1, \ldots, x_{\nu-1} \rangle$	x_ν
x	$\langle x_1, \ldots, x_\nu \rangle$	nil		nil	$\langle x, x_1, \ldots, x_{\nu-1} \rangle$	x_ν
nil	$\langle\rangle$	y			$\langle\rangle$	
x	$\langle\rangle$	y		nil	$\langle x \rangle$	
nil	$\langle x_1, \ldots, x_\nu \rangle$	y			$\langle x_1, \ldots, x_\nu \rangle$	
x	$\langle x_1, \ldots, x_\nu \rangle$	y	$(\nu < n)$	nil	$\langle x, x_1, \ldots, x_\nu \rangle$	
x	$\langle x_1, \ldots, x_n \rangle$	y		x	$\langle x_1, \ldots, x_n \rangle$	

Table 1: Specification of the transition relation of a bounded buffer

i remains stable. If the output variable o is nil and there is a message to deliver, it is copied into o. The consumer receives a message y from o by resetting o to nil.

The content of the buffer b is given as a sequence $\langle x_1, \ldots, x_\nu \rangle$ of messages, where $\langle\rangle$ denotes the empty buffer. There are various possibilities to implement sequences of messages with BDDs. The most obvious choice is to use n variables $b_1, \ldots, b_n$ over $\mathcal{M}$, such that b_1 contains the front element of the message queue, and incoming messages are appended into the smallest b_ν which is empty (contains nil as value). The necessary assignment operation for this modelling is given in Figure 7.

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : b[j+1];
  !(i=nil) & !(o=nil) : if !(b[j-1]=nil) & b[j]=nil then i
                        else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j]=nil then nil
                        else if b[j+1]=nil then i
                        else b[j+1] fi fi;
  esac;
```

Figure 7: Bottom-version of buffer slot assignment

In this modelling, we rely on the fact that whenever $b_j = nil$, then for all $k \geq j$, also $b_k = nil$. This assumption only holds for the reachable states of a buffer which is initially empty; there are many transitions from illegal, i.e., nonreachable states to other illegal states in this model. In an explicit representation of the transition relation, one should try to avoid these redundant entrys. With BDDs, however, even though the size of the transition relation is much bigger than the transition relation restricted to the reachable states, its representation is much smaller. Since the value of each buffer slot depends only on its immediate neighbours, in fact the size of the representation is linear in the number of slots.

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : if (b[j-1]=nil) then nil else b[j];
  !(i=nil) & !(o=nil) : if (b[1]=nil) then b[j+1] else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j]=nil then nil else b[j+1] fi;
  esac;
```

Figure 8: Top-version of buffer slot assignment

In the above implementation, the buffer content is shifted upon output. We refer to this modelling as the *bottom* version, because sent messages can be imagined to "sink to the ground". A dual implementation of the buffer shifts down the content one slot whenever an input is performed, and inserts the new element into the topmost slot b_n. Consequently, we call this modelling, where messages "float to the surface", the *top-version* of a bounded buffer. To perform an output in this version, the content of the lowest non-*nil* slot is copied into the output variable o. The respective code segment is given in Figure 8.

A third possibility is to use a *circular* implementation of the buffer: On input, the value of the input variable is copied into slot b_i, where $b_i = nil$ and $b_{i-1} \neq nil$; on output, o is set to b_j, where $b_j \neq nil$ and $b_{j-1} = nil$. To be able to distinguish between first and last element of the queue in this version, we have to make sure that there is at least one slot with content *nil*; therefore there has to be one more place than the actual capacity of the buffer. In the assignment clause in Figure 9, subtraction and addition of one is to be understood modulo n.

An alternative to the use of an empty slot would be to introduce queue-pointers for the position of the first and/or last element of the queue; this idea can be applied to all three of the above modellings. However, these alternative versions turn out to be worse than the direct encoding via *nil*-test which is given above. In general, the queue-pointers would be functionally dependent of the content of the buffer; such functional dependencies can blow up the BDD size significantly ([8]).

Similarly, we can introduce additional BDD-variables indicating whether the buffer is empty or full; however, these variables tend to increase the size of the representation by a linear factor and usually can be replaced by appropriate boolean macro definitions. On the other hand, such variables can be important

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : if b[j-1]=nil then nil else b[j];
  !(i=nil) & !(o=nil) : if !(b[j-1]=nil) & b[j]=nil & b[j+1]=nil
                        then i else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j-1]=nil then nil
                        else if b[j]=nil then i else b[j] fi;
  esac;
```

Figure 9: Circular version of buffer slot assignment

if the BDD is represented as a conjunction of partitioned transition relations, see [5].

Finally, it is not always advisable to test whether a slot b_i contains the value *nil* by the test `b[i]=nil`. As we will see in the next section, it can be better to increase the message width w by one, such that the first bit of each message is a kind of checksum, indicating whether this message is *nil* or not.

3 Complexity Considerations

The BDD for the bottom version of a buffer of size n consists of two parts, one for the case that the buffer content remains stable, and one for the case that the buffer content is shifted down by one slot. The first part consists of a sequence of local diamonds for each slot, similar as in the example above. The BDD for the second part is depicted in Figure 10 for the special case $n = 2$ and $\mathcal{M} = \{x_1, x_2, x_3\}$.

As can be seen, for a new buffer slot b_{n+1}, $O(m^2)$ nodes are added to the BDD for a buffer of length n. Therefore the representation is of order $n \cdot m^2$, i.e., linear in the length and exponential in the width of the buffer. Since the transition relation is "almost" a function, a matrix representation would require $O(m^n)$ entrys, whereas a boolean algebra or programming language representation such as the SMV code above, is of order $m + n$ (or even constant, if array subscripts are allowed).

For the top version, the complexity of the representation is comparable to the bottom version. In the circular version, b'_n depends on b_n, i, b_{n-1}, and on b_1. This non-local dependency causes a blowup of factor 2, since the emptyness of b_1 has to be decided while testing b'_n. Moreover, to test whether a buffer is full or not we have to test whether any two adjacent slots are nil. This nonlocal test again blows up the complexity of this modelling.

As was to be expected, the number of reachable states is identical in the bottom and top modellings; of course, this number is exponential in the length

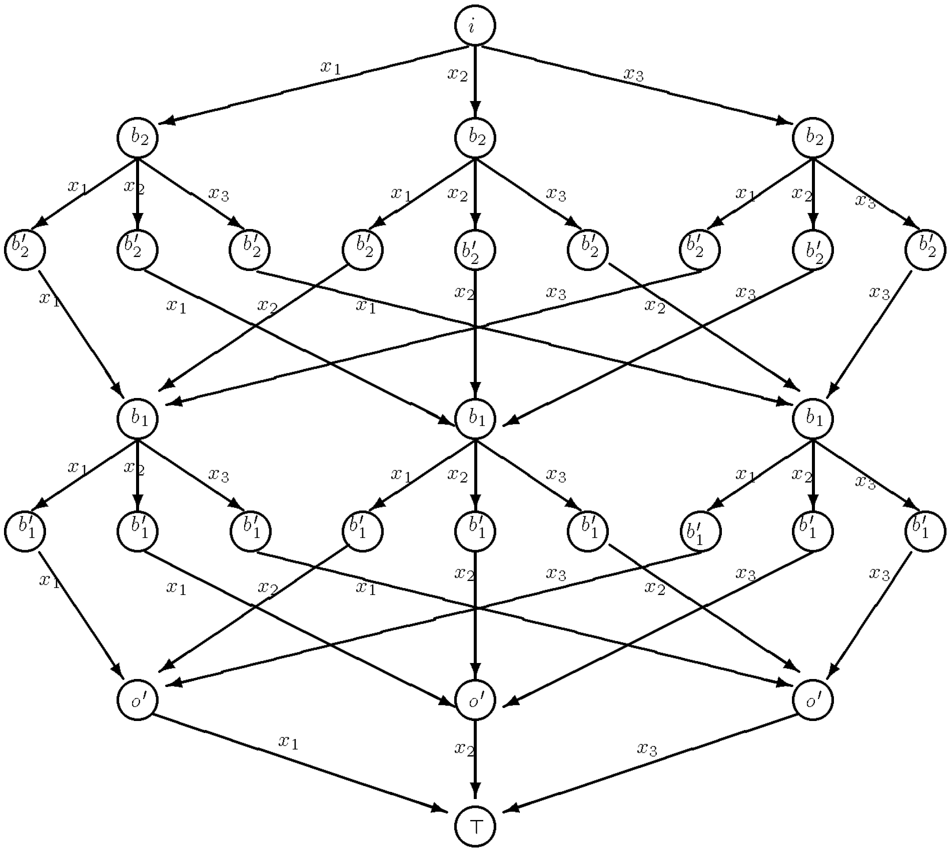


Figure 10: BDD for shifting down the buffer content

of the buffer. For the circular implementation, the number of reachable states is approximately m times as much, since it contains an additional slot.

Table 2 summarizes the size of the BDDs of the transition relation for $m = 4$ (i.e., $w = 2$), and order $i < b_n < ... < b_1 < o$. All results were obtained with the public-domain SMV system; other BDD-based verification tools yield similar results. The buffers were embedded in a simple producer-consumer environment, where the producer and consumer are asynchronous, and the message to be sent or received does not depend upon or influence the state of the sender or receiver, respectively.

A critical factor in this approach is the message width w. As indicated in table 2, e.g. the bottom implementation of a buffer of length 5 and width 2 has size 1458. For $w = 3$, this size is 11774, and for $w = 4$, it is 108357. In [2] it is proved that for any finite function, a BDD of polynomial size exists iff the function can be realized by a polynomially bounded depth circuit. For message buffer, certainly the transition function can be realized by such a circuit; thus there exists a BDD which is polynomial both in n and w.

length	3	5	7	9	11
bottom	714	1458	2204	2950	3696
top	599	1113	1627	2141	2655
circular	1038	2350	3999	5307	6833
reach	1400	12740	2^{16}	2^{20}	2^{23}

Table 2: BDD size of transition relation and reachable state set

If there is no constraint on the order of variables, then such a BDD can be constructed by interleaving the bits of all slots: Let $i = i_1 ... i_w$, $b_j = b_{j1} ... b_{jw}$, and $o = o_1 ... o_w$. Then for each $k \leq w$, $(i_k, b_{nk}, ..., b_{1k}, o_k)$ can be regarded as a buffer of width 1. The only "nonlocal test" in this buffer of length 1 is whether some slot b_j is empty: if this is determined by comparison of b_{j1} and ... and b_{jn}, then we still have an exponential growth. If we introduce additional bits $(i_0, b_{n0}, ..., b_{10}, o_0)$ which are 0 iff the corresponding message is nil, then each bit-slice is linear in the length of the buffer. For the order $i_0 < b_{n0} < ... < b_{10} < o_0 < i_1 < b_{n1} < ... < b_{11} < o_1 < ... < i_w < b_{nw} < ... < b_{1w} < o_w$, these small BDDs are simply added, and the overall complexity is $O(w \cdot n)$.

Unfortunately, in many practical examples it is not possible to choose such a bitwise interleaved order. Usually, the input and output variables are imported from other processes, and their order cannot be chosen arbitrarily. An argument similar to the one from section 1 shows that for any order, in which i is before all buffer bits, the representation is exponential in w. Therefore, in practical verification, w should be kept as small as possible. There are several ways to do so:

- For every channel, define a separate message alphabet;
- replace a parametrized message $m(t)$ with $t \in \{t_1, ..., t_k\}$ by a list of messages m_{t1}, ..., m_{tk};
- replace a compound message by a sequence of messages, and
- abstract several different messages into one.

When using the latter two methods, one has to be careful to preserve the semantics of the original model ([6]). Using these techniques, we have been able to verify systems with up to 2^7 different messages.

4 Alternative Approaches

In recent papers ([7],[1]) it is suggested to extend the BDD data structure for the representation of message buffers. The new data structures are called QBDDs and QDDs, respectively. The basic idea is to replace the consecutive testing of buffer variables by a repeated test of one and only one variable. Therefore, the representation of the transition relation is independent of the buffer size. Moreover, even systems of which the maximum amount of buffer space is not statically known can be verified.

However, as we have shown above, the (static) length of a buffer may not be the most important factor in the representation of the transition relation. Moreover, "buffer overflow" errors in the system can only be detected with a bounded buffer. Even worse, in systems on which a full buffer forces delay of the sender, with QBDDs we have to introduce an additional counter variable. For these type of systems, BDDs seem to be more adequate than QBDDs or QDDs.

Being able to represent the transition relation is only a necessary prerequisite for the verification of a system. Equally important is the size of the representation of the *reachable states* $\mathcal{R}$ of the system. Unfortunately, the size of the BDD for $\mathcal{R}$ has no predictable connection to the size of the BDD for the transition relation.

In many systems both the number of reachable states and its representation are linear in the number of iteration steps of the model, iff the system is correct. This is due to the fact that on reachable states, the transition relation is "almost" functional, yielding either a single or a small number of successor states. On the other hand, from an "impossible" state usually many other "impossible" states are reachable. A drastic example is Valmari's elevator for which the reachable state set (in any representation) explodes as the elevator breaks through the ceiling and skyrockets into the air. Thus an exponential increase in (the representation of) $\mathcal{R}$ after some number of steps almost certainly indicates an error.

In [7] it is claimed that "there are cases where the QBDD representation is strictly more concise than the BDD representation". Assume our buffer in a context where the producer sends one fixed sequence of messages x_1, x_2, ... x_ν. That is, the reachable buffer content is $\{\langle\rangle$, $\langle x_1\rangle$, $\langle x_2 x_1\rangle$, ..., $\langle x_\nu ... x_2 x_1\rangle$, $\langle x_2\rangle$, ... $\langle x_\nu ... x_2\rangle$, ..., $\langle x_\nu\rangle\}$. With the top- and bottom version of the buffer, the representation of this set is quadratic in ν, whereas with the circular representation and also with QBDDs it is linear in ν.

On the other hand, consider the case that the producer can send an arbitrary sequence of messages. In this case, the top- and bottom-versions are of constant size, whereas the QBDD implementation is linear in the number of sent messages.

In practical examples, such extreme cases are rare. In our experiments, we have found no significant difference in the size of the reachability sets of the various alternatives. The number of parallel processes and their relative order has a much bigger impact on the size of the BDD for $\mathcal{R}$ than the actual implementation of the buffer. Typically we can handle systems of up to 5 processes, each with approx. $2^4 - 2^5$ local states, where each process is equipped with a buffer of $n, w \leq 5$. However, there still is a need for heuristics which use dependencies between the processes to obtain a "good" order for the process state variables.

An important observation is that the content of a message buffer used to coordinate processes shows regular patterns, which also depend on the state of the processes. E.g., in a certain process state the buffer might always contain only copies of two different messages from $\mathcal{M}$. As another example, some specific message might always be followed by some other specific message in the buffer. Currently we are investigating methods how these regularities can be exploited to reduce the size of the representation of the reachability set.

References

[1] B Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of* 5^{th} *CAV*, New Brunswick, July 1996.

[2] R B Boppana and M Sipser. The complexity of finite functions. In J van Leeuwen, editor, *Handbook of theoretical computer science, Vol. A*, chapter 14, pages 757–805. Elsevier, Amsterdam, 1990.

[3] Randal E Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-92-160, CMU School of Computer Science, Pittsburgh, July 1992.

[4] Jerry Burch, Edmund M Clarke, David Dill, and Ken McMillan. Symbolic model checking: 10^{20} states and beyond. In 5^{th} *IEEE LICS*, June 1991.

[5] Jerry Burch, Edmund M Clarke, and David Long. Symbolic model checking with partitioned transition relations. In *Proc. IFIP Conf. on VLSI*, Edinburgh, August 1991.

[6] Edmund M Clarke, Orna Grumberg, and David Long. Model checking. In Manfred Broy, editor, *Deductive Program Design*, Nato ASI Series F, pages 305–350. Springer, Berlin, 1996.

[7] Patrice Godefroid and David E Long. Symbolic protocol verification with queue BDDs. In *Proceedings of IEEE LICS*, New Brunswick, July 1996.

[8] Alan Hu and David Dill. Reducing BDD size by exploiting functional dependencies. In *Proc.* 30^{th} *ACM/IEEE DAC*, 1993.

[9] Ken McMillan. *Symbolic model checking.* Kluwer, 1993.

Using Slice Sets for Model Checking Causal Nets

Maximilian Frey
Institut für Informatik, Technische Universität München
Orleansstr. 34, 81667 Munich, Germany
frey@infromatik.tu-muenchen.de

Abstract

A new method for model checking parallel and distributed runs is introduced. This method is based on slice sets which contain all global states of a run causally ordered between an upper bound and a lower bound global state. Using slice sets, it is not necessary to construct a global model of the program run. Instead a model containing only local states of the run together with the causal relation between them is used.

1 Introduction

Model checking is more and more applied to increase the reliability of safety relevant industrial applications by early detection of errors [2, 1]. During the last decade tools have been build which automatically check whether a given model of a parallel and distributed system satisfies its requirements (cf.[2, 7]). Normally, these tools are checking global models of systems consisting of all global states together with the connections between them. Recent tools are able to verify systems with up to 10^{120} states [2]. To verify systems during the early software engineering phases this amount of states is sufficient. However, it is not enough to verify implementations of bigger systems.

In contrary to model checking systems, there has not been done much work in the field of model checking program runs, though the state explosion by different input data is avoided. This way, runs of bigger systems can be checked automatically. Model checking of runs can, of course, not be used to verify systems but they are helpful to test and debug systems.

The systems our model checking method can be used for consist of several threads of control running in parallel. Threads can use a common address space or can be distributed over different address spaces, where each threads runs in only one address space. They may communicate by common variables or by messages and they may be created dynamically during the program run. This model of parallel and distributed systems contains most of the paradigms for parallel and distributed programming.

The first part of the paper describes the model checking method. Section 2 introduces the model of parallel and distributed program runs. Section 3 describes the temporal logic to specify the requirements of the program which have to be satisfied. In section 4 we describe how it is automatically checked whether a temporal logic formula is satisfied by a model of a parallel and distributed program run.

The second part of the paper describes in section 5 how the model checking can be applied to testing and debugging.

2 Modelling Parallel and Distributed Runs

We use finite causal nets (cf. [9]) to model program runs. Finite causal nets describe bipartite graphs (P, T, R) consisting of a nonempty and finite set P of places and a nonempty and finite set T of transitions. The graph is defined by the relation R which is acyclic and each place of the graph has maximally one predecessor and one successor.

Different states of a thread are distinguished by a nonempty and finite set of predicates *Pred*. The elements of *Pred* are specific for one thread. *Pred* contains for each of its elements p also the negation of p. Therefore, a idempotent and bijective function $\mathfrak{Neg}$ is defined over *Pred* which assign to each predicate its negation. In this way, a global state can only contain one place in which a predicate or its negation is satisfied.

Global states are represent by a maximal set of local states, called slices, where all elements of a slice are not causally ordered to another element by R^+ [10].

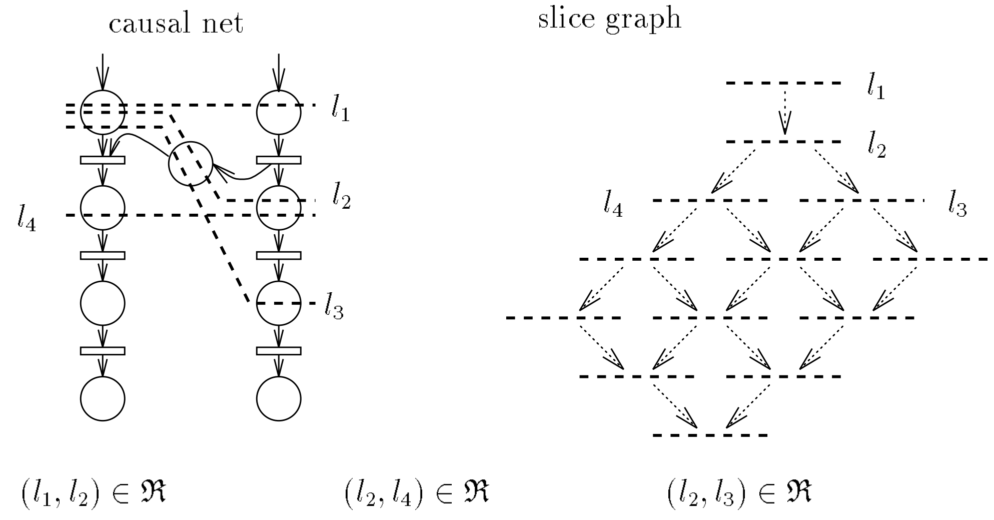


Figure 11: Causal net and its slice graph

The slice-graph $(\mathfrak{W}, \mathfrak{R})$ of a finite causal net consists of a set of slices $\mathfrak{W}$ as set of nodes and a relation $\mathfrak{R}$ describing the edges of the graph. An element of

$\mathfrak{R}$ describes how slices are changing when a transition is fired. A slice graph of a finite causal net contains an initial and a terminal slice which are predecessor and resp. successor of all slices of the slice graph. In this way, a slice graph is a complete lattice. Slice graphs differ from case graphs (cf.[9]), because the edges of a slice graph describe only the firing of one transition and an edge of the case graph describes the firing of some transitions at the same time. Furthermore, the relation of a slice graph is the minimum of all relations having the same transitive closure as the slice graph.

Figure 11 shows an example for a causal net and its slice graph. Circles of the causal net represent places, boxes transitions and solid arcs elements of the relation R. Dashed lines of the slice graph represent slices and dotted arcs represent elements of $\mathfrak{R}$.

3 Temporal Logic

To specify the requirements which have to be satisfied by all runs of a program we use a temporal logic based on the logic proposed in [10]. The syntax of the logic contains predicates of $Pred$ as atomic formulas as well as the logic operations **and**, **not**, **next**, **sometime** and **until**.

The logic is three valued. A formula p can be satisfied ($l \models p$), unsatisfied ($l \not\models p$) or undefined ($l \models\!\!\!\!\!\ominus\, p$) in a slice l. A formula can be undefined, because predicates p and places s may exist such that neither $p \in \mathfrak{B}(s)$ nor $\mathfrak{Neg}(p) \in \mathfrak{B}(s)$. In this way, it can be modeled that a variable of a formula is not visible in all places of a slice. The semantics is defined as follows, where l is a slice and p, q are formulas:

1. $l \models p \in Pred$ iff a place $s \in l$ exists such that $p \in \mathfrak{B}(s)$.
   $l \not\models p \in Pred$ iff a place $s \in l$ exists such that $\mathfrak{Neg}(p) \in \mathfrak{B}(s)$.
   $l \models\!\!\!\!\!\ominus\, p \in Pred$ iff for all places $s \in l$ it is valid that $p \notin \mathfrak{B}(s)$ and $\mathfrak{Neg}(p) \notin \mathfrak{B}(s)$.

2. $l \models$ **not** (p) iff $l \not\models p$.
   $l \not\models$ **not** (p) iff $l \models p$.
   $l \models\!\!\!\!\!\ominus$ **not** (p) iff $l \models\!\!\!\!\!\ominus\, p$.

3. $l \models (p$ **and** $q)$ iff $l \models p$ and $l \models q$.
   $l \not\models (p$ **and** $q)$ iff $l \not\models p$ or $l \not\models q$.
   $l \models\!\!\!\!\!\ominus\, (p$ **and** $q)$ iff ($l \models\!\!\!\!\!\ominus\, p$ and $l \models q$) or ($l \models p$ and $l \models\!\!\!\!\!\ominus\, q$) or ($l \models\!\!\!\!\!\ominus\, p$ and $l \models\!\!\!\!\!\ominus\, q$).

4. $l \models$ **next** p iff a slice l' exists such that $(l, l') \in \mathfrak{R}$ and $l' \models p$.
   $l \not\models$ **next** p iff for all slices l' with $(l, l') \in \mathfrak{R}$ not $l' \models p$.
   Not $l \models\!\!\!\!\!\ominus$ **next** p.

5. $l \models$ **sometime** p iff a slice l' exists such that $(l,l') \in \mathfrak{R}^+$ and $l' \models p$.
   $l \not\models$ **sometime** p iff for all slices l' with $(l,l') \in \mathfrak{R}^+$ it is not valid that $l' \models p$.
   Not $l \models\!\!\!\!\!\!\!\ominus$ **sometime** p.

6. $l \models$ (p **until** q) iff a slice l' exists such that $(l,l') \in \mathfrak{R}^+$ and $l' \models q$ and for all slices l'' with $(l,l'') \in \mathfrak{R}^+$ and $(l'',l') \in \mathfrak{R}^+$ it is not valid that $l'' \not\models p$.
   $l \not\models$ (p **until** q) iff for all slices l' with $(l,l') \in \mathfrak{R}^+$ and $l' \models q$ it is valid that a slice l'' exists such that $(l,l'') \in \mathfrak{R}^+$ and $(l'',l') \in \mathfrak{R}^+$ and $l'' \not\models p$.
   Not $l \models\!\!\!\!\!\!\!\ominus$ (p **until** q)

A formula is satisfied in a causal net iff the formula is satisfied or undefined in all slices of the causal net. A formula is unsatisfied in a causal net iff as slice exists in which the formula is unsatisfied.

The formula (**not** p **or always not** (p_1 **and** p_2)) specifies that beginning with the first time p is valid never p_1 and p_2 are valid at the same time.

4 Model Checking

The main idea of the model checking method is to mark all subformulas of a formula with slices which satisfy the formula. The aim of the method especially if it is applied to testing and debugging, is to state why a property and its formula is not satisfied. To find these causes, the slices and places are necessary in which the formula of the property and its subformulas are unsatisfied. In this way, the first step of the method is to negate the formula.

To calculate the slices which satisfy a formula of type **not** p, the complement of the slices which make p satisfied or undefined, is necessary. If p is an element of $Pred$, **not** p can be exchanged by $\mathfrak{Neg}(p)$. If p is of type **next** (q), **sometime** (q) or (q **until** q'), no slices exist which make p undefined and only the slice which make p unsatisfied are necessary to calculate the complement. In all other cases the slices which satisfy p as well as the slices which make p unsatisfied have to be calculated. To avoid the calculation of the slices which make **not** p undefined, **not** p has to be transformed to a formula p', where all negations are placed directly before the predicates and the calculation of the complement is only necessary at the temporal logic operations. Therefore, a new logical basis is defined which does not contain the negation and is extended by the operations **or**, **allnext**, **always** and **before**. These operations are defined as follows:

- (p **or** q) $\equiv$ **not** (**not** p **and not** q)
- **all_next** $p \equiv$ **not** (**next not** p)
- **always** $p \equiv$ **not** (**sometime not** p)

- (p **before** q) $\equiv$ **not** (**not** p **until** q)

Following the main idea of the model checking method, slices are calculated for all subformulas of a formula to evaluate it. The order of calculating the slices for subformulas is defined by a formula tree which contains subformulas of the formula in its nodes. It is similar to the syntax tree and contains predicates in its leaves. The successors of inner nodes are defined by the syntactic composition of the formula of the node out of subformulas and an operation. A formula tree is defined as follows:

1. A node with formula (p **and** q), (p **or** q) oder (p **before** q) has two successors with formulas p and q.
2. A node with formula **next** p or **sometime** p has one successor with formula p.
3. A node with formula **allnext** p or **always** p has one successor with formula **not** p.
4. A node with formula (p **until** q) has two successors with formulas **not** p and q.
5. The root contains the negation of the formula which has to be checked.

Figure 12 shows an example for a formula tree. The formula tree represents the negated formula of section 3.

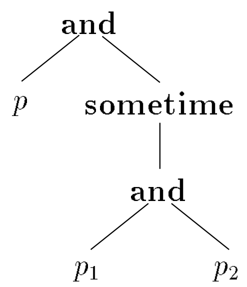


Figure 12: Formula tree to check the formula of section 3

The formula tree is evaluated bottom-up by calculating the slices of a node from the slices of successor nodes by a function only depending on the operation of the node.

The calculation of slices can be done more efficient if sets of slices instead of single slices are calculated at one time. Sets of slices which are represented by two slices l_o end l_u and contain all slices l with $(l_o, l) \in \mathfrak{R}^*$ and $(l, l_u) \in \mathfrak{R}^*$ are called slice sets $((l_o, l_u))$ (see Figure 13).

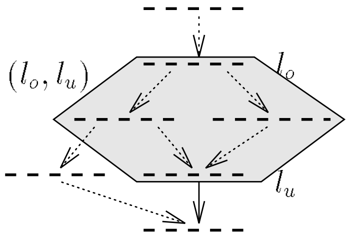


Figure 13: A slice set

In this way, all slices which contain a place s are represented by a slice set $(lConc(s) \cup \{s\}, gConc(s) \cup \{s\})$. $lConc(s)$ is the set of places s' not causally ordered with s and all predecessors of s' w.r.t. R are causally ordered with s. $gConc(s)$ contains all places which are not causally ordered with s and all successors of s' w.r.t. R are causally ordered with s. In Figure 14 the filled box of the slice graph represents the slice set consisting of all slices which contain the filled place of the causal net.

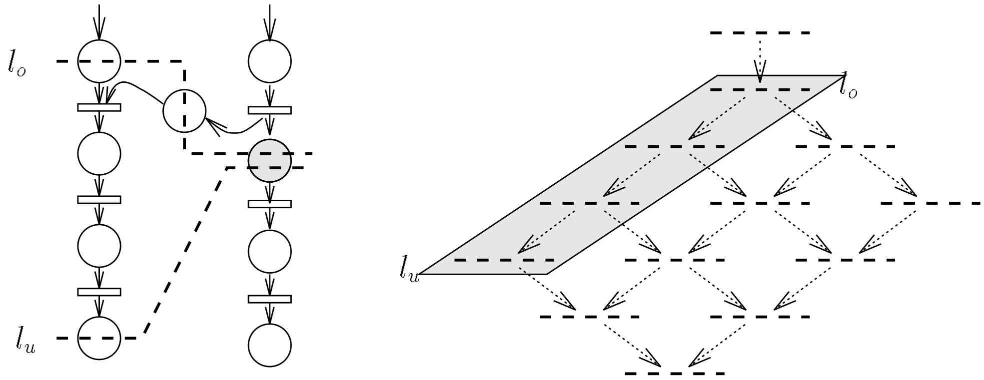


Figure 14: Calculating slice sets from one place

The slice sets of the leaf nodes of the formula tree are all slices sets consisting of all slices containing a place s for all places s satisfying the predicate of the leaf node.

The slice sets of nodes of the formula tree containing a formula of the type **sometime** p are the slices sets (l_I, l') for all slice sets (l_o, l_u) of the successor node and all predecessors l' of l_u (see Figure 15).

The complement necessary to calculate the slices sets of nodes with formulas of the type **always** p, **allnext** p, or (p **before** q) is calculated by building slice sets containing the predecessors of l_o in $\mathfrak{R}^+$, the successors of l_u in $\mathfrak{R}^+$ and the slices not causally related by $\mathfrak{R}^*$ to l_u for all slice sets (l_o, l_u) of **sometime** p, **next** p, or (p **until** q). In Figure 16 the light filled box represents a slice set of

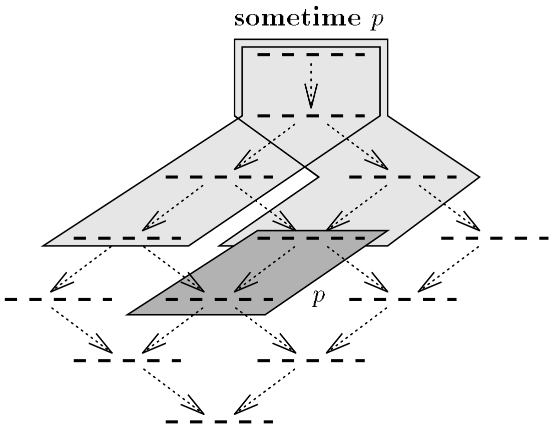


Figure 15: Slice sets of **sometime** p

(l_o, l_u) of **sometime** p, **next** p, or (p **until** q) and the dark filled boxes represent the complement of (l_o, l_u).

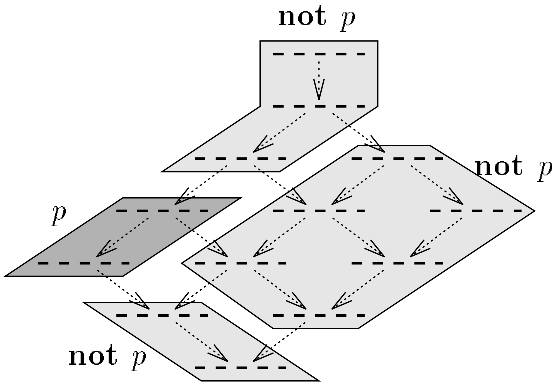


Figure 16: The complement of the slice set of p

Slice sets of nodes with formulas of the type (p **and** q) are the sections of a slice set (l_o, l_u) and a slice set (l'_o, l'_u) for all slice sets (l_o, l_u) of one successor node and all slice sets (l'_o, l'_u) of the other successor node (see Figure 17).

When a slice set is calculated in the root during the evaluation of the formula tree, the checked formula is not satisfied. Using this slice set and the slice sets of the formula tree nodes needed to calculate it, a sub net containing the slice sets describes why the formula is not satisfied.

Generally, model checking methods can be divided into two classes: tableau based methods (cf.[11, 6]) and global methods (cf.[2]). Tableau based methods mark all states with subformulas which are satisfied in this state, while global methods mark subformulas of a formula with states in which this subformula is satisfied.

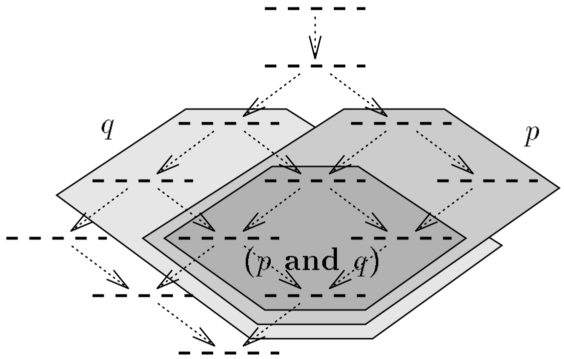


Figure 17: Slice sets of (p **and** q)

The proposed method is a global method, because subformulas of nodes of the formula tree are marked with slices satisfying the subformula.

In contrary to tableau based and global methods the proposed method needs not a global model. Only those slices are calculated with are necessary to decide whether a formula is satisfied or not. In this fact it is similar to the method in [3]. This method does only construct one set of slices represented by a set of upper bound slices and a set of lower bound slices. This set contains all slices which are predecessors of a slice of the lower bound and not predecessor of a slice of the upper bound. This method is not able to check formulas containing **next** or **until**.

5 Applications

The method introduced in section 4 is not able to verify programs but it can be applied to testing and debugging. Figure 18 describes, how it is automatically

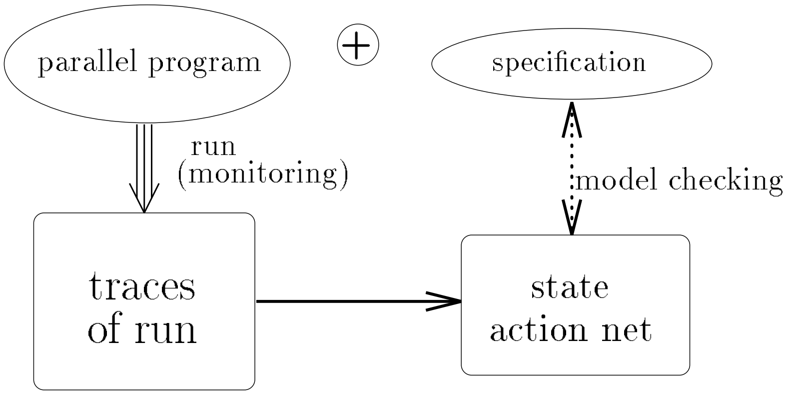


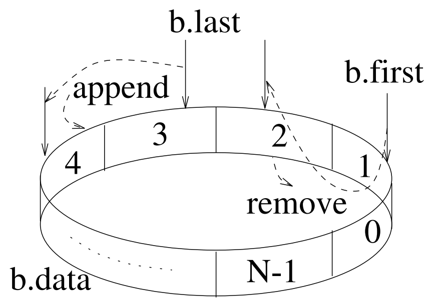
Figure 18: Checking program runs

checked whether the specified properties are satisfied during a program run.

During the program run events as for example read and write access to shared variables, sending and receiving of messages, execution of a statement of the source code, or entering and leaving of functions are recorded in a trace. This trace is used to generate a causal net called state action net, because its nodes have special semantic meanings. Finally, the model checking method checks whether the specification is satisfied in the state action net.

The traces describe the sequential dependencies within one thread as well as dependencies between threads as for example between a read access and the last previous write access to the same shared variable. In this way, traces describe a partial order on the events of a program run. This partial order does not only describe the original program run but a class of program runs, because some arbitrary dependencies within the program run have been eliminated. Because this class of program runs can contain an erroneous program run, errors can be detected even if they have arbitrarily not been occurred during the original program run.

To use the model checking method for testing and debugging, causal nets and the temporal logic as well as the method itself have to be extended.



Definition of the sequential functions remove and append

```
#define N 200
struct queue { int data[N];
               int first;
               int last } b={{0},0,0};
put(int elem) {
  mutual exclusion w.r.t. put:
    wait until b not full;
    append(elem); }
get(int *elem) {
  mutual exclusion w.r.t. get:
    wait until b not empty;
    remove(elem); }
```

Figure 19: A ring buffer with parallel access

We demonstrate the usage of the model checking method for testing and debugging by the example of Figure 19

This example shows a fragment of a parallel program which implements a ring buffer. The buffer consists of a data field *data*, a pointer *first* pointing at the element in the data filed before the first element of the buffer, and a pointer last pointing at the last element of the buffer. The sequential function *append* increments last and inserts an element in the buffer at the position last points on, afterwards. The sequential function *remove* increments *first* and fetches afterwards the element *first* points on out of the buffer. *put* and *get* are functions which are executed in parallel. For each call of *put* or *get* a thread is generated which executes *put* or *get*. *put* waits until the buffer is not full and calls *append* with mutual exclusion to other threads executing *append*. *get* waits until the buffer is not empty and calls *remove* with mutual exclusion to other threads executing *remove*.

Some runs of the program show the erroneous behaviour that values returned by *get* were never inserted in the buffer before with *put*. This behaviour may be caused by hurt mutual exclusion. There exist three kinds of mutual exclusions in the example:

- Mutual exclusion between different threads executing *append*.
- Mutual exclusion between different threads executing *remove*.
- Mutual exclusion between *append* and *remove* if the buffer contains maximally one element.

To describe parallel and distributed program runs transitions of the causal net representing the run describe the execution of statements of the source code which has to take place at the same time, because of the semantics of the program. An example for such a execution is the execution of a sending and a receiving statement with synchronous communication between them. Transitions having such a meaning are called actions.

The places of the causal net representing a run contain only thread local information. They contain the identification of the thread, the functions executed at the state by the thread in the order they have been entered and values of variables which can be access by the thread at the local state. Places which have such a meaning are called local states.

A causal net which contains only actions as transitions, and which contains local states, is called state action net.

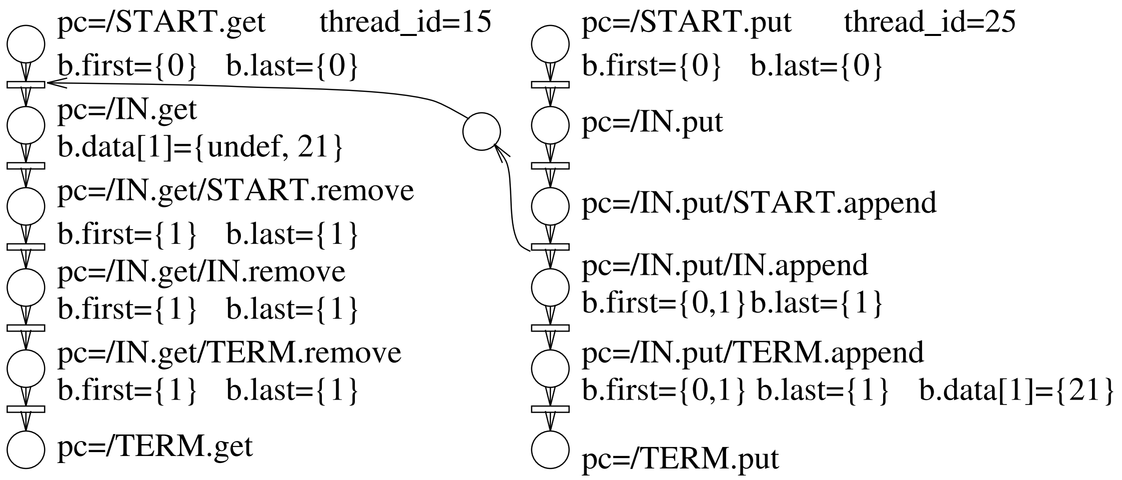


Figure 20: State action net of a run of the ring buffer

Figure 20 shows a state action net generated during a program run of the ring buffer of Figure 19. During this run *put* as well as *get* have been called for one time. In this way, two threads have been generated. During the execution of **put** the value 21 has been inserted in the buffer. The local states of the state action net contain information about the functions which have been executed at the local state by the thread of the local state. The attributes START, IN and TERM distinguish whether a state is at the beginning of the execution, somewhere during the execution or at the end of the execution of a function. A local state can contain for each variable which is visible at the local state a set of values. A variable can have more than one value in a local state s if this variable is written in an action not causally ordered to s. The variable can take on the value before the write access or after the write access.

To specify the mutual exclusion properties, we have to extend the temporal logic of section 3 in two directions: First we have to exchange predicated with specifications of properties of local states. This specifications have to describe the execution of functions as well as comparisons between values of variables. Second, states of different threads have to be distinguishable. In systems with dynamic creation of threads an identification of threads before the run is not possible. The syntax of the logic is therefore extended by place holders for identifiers of threads, called thread identification variables (TI-variables). Each predicate p contains additionally a TI-variable t (t:p). To describe equality or inequality or the creator relation between threads, it can be specified that a formula p is only defined iff TI-variables t_1, t_2 are related (p **if** $t_1 = t_2$, p **if** $t_1 \neq t_2$, p **if** $t_1 =$ **caller of**(t_2)). The semantics of the logic is defined additionally to slices by assignments of TI-variables to identifiers of threads of the program run (TI-assignment).

Properties of local states are specified by a propositional logic (local logic). This logic contains comparisons of variables, **start.**f, **in.**f and **term.**f as atomic predicates. **in.**f describes the execution of a function f. **start.**f and **term.**f describe the entering and leaving of a function f.

Using these extensions the mutual exclusion properties can be specified as follows:

- **not**(t1:(in.*append*) **and** t2:(in.*append*)) if t1 != t2
- **not**(t1:(in.*remove*) **and** t2:(in.*remove*)) if t1 != t2
- **not**(t1:(in.*append* and (*b.last*-*b.first*) modulo N<=1) **and**
  t2:(in.*remove* and (*b.last*-*b.first*) modulo N<= 1))

Finally, the model checking method has to be extend, because the semantics of the temporal logic has changed: First, the leaves of the formula tree contains formulas of the local logic instead of predicates such that for all local states the formulas in the leaf nodes have to be checked. Because atomic formulas of the temporal logic are extended by a TI-variable for each local state s which satisfies the formula of the leaf node, the slice set consisting of all slices which contain the local state together with a TI-assignment which assigns the thread identifier of the local state to the TI-variable of the formula, have to be calculated.

Second, the structure of nodes of the formula tree has to be changed such that it contains a partition of slice sets for each TI-assignment assigning thread identifiers of the program run to TI-variables of the formula of the node.

To calculate the slice sets of an inner node of the formula tree for all TI-assignments of the successor nodes it has to be checked whether they fit together. In this case, a new partition is created in the node with a TI-assignment merged together from the TI-assignments of the successor nodes and the slice sets of the partition are calculated from the slice sets of the partitions of the successor nodes as described in section 4.

The properties for testing parallel and distributed programs may be generated from the requirements during the program development phases. To test programs it is necessary to eliminate the nondeterminism such that in all replays of a program run the same formulas of the specification are unsatisfied. This means that the result of test cases have to be reproducible. State action nets allow to reduce nondeterminism, but they eliminate the nondeterminism not totally. Different access sequences to synchronization objects can lead to different state action nets (see [5]).

If an error has occurred during testing the description and visualization of the causes why a formula of the specification is not satisfied helps to formulate hypothesis about the causes of the error. In this way the method can also be used for an incremental approach to the error. This approach leads always to an unambiguous localization of the error which allows its elimination (see[8, 4]).

6 Conclusions

We introduced a model checking method for parallel and distributed program runs. The method uses causal nets to describe runs of parallel and distributed programs. The logic to specify properties which have to be checked is based on [10]. The method builds not a global model of the causal net. It only generates those global states which are necessary to check the formulas. To reduce the complexity of the method global states are put together to sets which are represented by an upper bound slice and a lower bound slice. In this way, the time complexity for checking formulas containing only the operations **and**, **or** and **always** is linear in the size of the causal net and exponential in the length of the formula. Furthermore we described how the method can be applied to testing and debugging.

References

[1] T. Cattel. Using concurrency and formal methods for the design of safe process control. In I. Jelly, I. Gorton, and P. Croll, editors, *Software Engineering for parallel and Distributed Systems*, pages 183–194, London, März 1996. IFIP, Chapman & Hall.

[2] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency, REX School/Symposium, Noordwijkerhout, The Netherlands*, LNCS 457, pages 840–851. Springer, Berlin, Juni 1993.

[3] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.

[4] M. Frey. Debugging parallel programs using temporal logic specifications. In I. Jelly, I. Gorton, and P. Croll, editors, *Software Engineering for parallel and Distributed Systems*, pages 122–133, London, März 1996. IFIP, Chapman & Hall.

[5] M. Frey and M. Oberhuber. Testing parallel and distributed programs with temporal logic specification. to appear.

[6] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *6th annual IEEE Symposium on Logic in Computer Science, Amsterdam*, pages 406–416, Los Alamitos, Juni 1991. IEEE Computer Society Press.

[7] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[8] G.J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[9] W. Reisig. *Petrinetze*. Springer, Berlin, 1986.

[10] W. Reisig. Temporal logic and causality in concurrent systems. In *Concurrency 88*, LNCS 335, pages 121–139. Springer, Berlin, 1988.

[11] G. Winskel. A note on model checking the modal μ-calculus. *Theoretical Computer Science.*, 83(1):157–167, 1991.

A Short Completeness Proof for Linear Temporal Logic

Anton Kölbl
Ludwig-Maximialians-Universität München
Oettingenstr. 67, D-80538 München
koelbl@informatik.uni-muenchen.de

Abstract

Linear temporal logic has received much attention in computer science, because it is very well suited for the description and the proof of program properties. In order to show completeness for the propositional fragment, there exist two standard ways: The model which has to be constructed therefore can be built up as a tree, whose nodes contain sets of formulae, as in [2]. Alternatively there is a proof in [1], which extracts the model out of the canonical model (in the sense of modal logic). But two complicated steps are necessary: First the big model is collapsed to a finite one, and since this contains clusters, it must be blown up to a linear model. Here a proof is presented, which also starts with the canonical model, but can read off the desired model as a sequence directly in the canonical model.

First I present the calculus whose completeness is to be shown. It contains the following axioms:

(taut) All propositional tautologies

(distr) The distribution axioms
$\circ(A \to B) \to (\circ A \to \circ B)$ und
$\Box(A \to B) \to (\Box A \to \Box B)$

(next) $\neg \circ A \leftrightarrow \circ \neg A$

(mix) $\Box A \to A \wedge \circ \Box A$

(ind) $\Box(A \to \circ A) \to (A \to \Box A)$

It contains the rules

(mp) Modus ponens: A, $A \to B \vdash B$

(nec) Necessitation: $A \vdash \circ A$ und $A \vdash \Box A$

The corresponding frame class are all frames in which $R_\circ$ is a linear ordering and $R_\Box$ is the tranitive-reflexive closure of $R_\circ$.

The correctness of the calculus is evident: If a formula α is derivable, then it is valid in all such frames. But now for the other direction: It has to be shown, that there exists for every non-derivable α a frame $S = (S, R_\circ, R_\Box)$, which falsifies α, i.e. there exists a model based on S, and a world w in this model, in which α does not hold.

Our starting point is the canonical model $M = (W, R_\circ, R_\Box, V)$. Therein we find a point s_0, in which α is not valid. This is because α is not provable, $\neg\alpha$ is consistent, and the points of the canonical model are exactly the maximal consistent sets of formulae. But this is not yet our desired model. Since the set $\{\neg\Box p, p, \circ p, \circ\circ p, ...\}$ is consistent, there are points s, for which $R_\Box(s)$ is a proper superset of $R_\circ^\star(s)$. So the canonical model is not a falsifying model, since is is based on a frame which is not suitable for our purpose.

Anyway, $R_\circ$ is functional in M; this follows from the axiom (next), which determines uniquely the next time point. We may assume that starting from each point s in W we find a ω-sequence by $R_\circ^\star$; for if we have a reflexive point, we can replace it by an $R_\circ$-sequence of irreflexive points with the same truth conditions.

Define Γ to be the least set of formulae such that:

- $\alpha \in \Gamma$
- $A \in \Gamma \Rightarrow$ subformulae$(A) \subseteq \Gamma$
- $\Box A \in \Gamma \Rightarrow \circ\Box A \in \Gamma$
- $\neg\Box A \in \Gamma \Rightarrow \circ\neg\Box A \in \Gamma$

Lemma 1 *If $s \in W$, $\neg\Box B \in \Gamma$, $\neg\Box B \in s$, then there exists an $R_\circ^\star$-variant t of s with $\neg B \in t$.*

An $R_\circ^\star$-variant is a point, which is reachable in fintely many $R_\circ$-steps, whereby points may be exchanged with other points equivalent modulo Γ. This does not change the truth conditions of subformulae of α.

This lemma is the central point in my proof, because it says, that for formulae $\neg\Box B$ the formula $\neg B$ can be realized already in $R_\circ^\star$ and not only in the perhaps much bigger $R_\Box$.

Now for the proof: Let $\Gamma = \{\gamma_1, ..., \gamma_n\}$.
Consider the infinite set of points P with the following properties:

- $s \in P$ (s as in the lemma)
- $t \in P$, $t \equiv_\Gamma u \Rightarrow u \in P$
- $t \in P, tR_\circ u \Rightarrow u \in P$

For each $p \in P$ consider the formula $A_p = (\neg)\gamma_1 \wedge (\neg)\gamma_2 \wedge ... \wedge (\neg)\gamma_n$ with $A_p \in p$. Exactly one of these formulae is in p, since p is maximally consistent. Set $A = \bigcup_{p \in P} A_p$. This is a finite formula, because there exist only finitely many combinations of the negated or unnegated formulae γ_i. Now we find $\Box(A \rightarrow \circ A) \rightarrow (A \rightarrow \Box A) \in s$ as an instance of the induction axiom.

Also the antecedens $\Box(A \rightarrow \circ A)$ is in s: We have to show $A \rightarrow \circ A \in u$ for all $sR_\Box u$. Therefore let $A \in u$. So $u \in P$ by definition of the formula A. We find: The uniquely determined $R_\circ$-successor of u is in P, and consequently $\circ A \in u$.

Therefore the conclusion $A \rightarrow \Box A$ is in s, too, and because of $A \in s$ also $A \in u$ for all $sR_\Box u$. If $\neg\Box B \in s$, we can find a u such that $sR_\Box u$ and $\neg B \in u$. Because of the above consideration there is a point $p \in P$, which realizes $\neg B$, and by construction of P this is exactly a $R_\circ^\star$-variant of s. So the lemma is shown.

Now we can define the desired model $(S, R_\circ, R_\circ^\star, V_S)$:
Since α is not provable, there exists a point $s_0 \in W$ (in the canonical model), in which α does not hold. Consider the first formula $\neg\Box B$ in $\Gamma \cap s_0$. If no such exists, choose $S = R_\circ^\star(s_0)$ (ω-sequence). Otherwise choose a sequence of $R_\circ$-steps with exchanges modulo Γ as in the lemma, such that in one point $\neg B$ is realized; call this point t. From there on choose the $R_\circ$-successors of t. If there is another formula $\neg\Box C$ in $\Gamma \cap s_0$, iterate this construction starting from t, until all formulae of the special type have been considered..

This defines an ω-sequence S and a model $(S, R_\circ, R_\circ^\star, V_S)$ with the following valuation: For propositional variables $p \in \Gamma$ and points $s \in S$ let $V_S(p, s) = V_M(p, s)$, i.e. the valuations are inherited from the canonical model. All other valuations are clearly irrelevant for the truth value of α.

Lemma 2 *For $s \in S$ and for subformulae A of α:*
$S \models A[s]$ iff $M \models A[s]$.

Proof by induction on A:

- Propositional variables q: claim is valid by definition
- $\neg A$, $A \rightarrow B$: clear

- $\circ A$ clear by inductive hypothesis for A, since we have only $R_\circ$-steps in S, and we identify points, which are equivalent modulo Γ ($A \in \Gamma$).
- $\Box A$: That's the interesting case.

  i) Let $M \models \Box A[s]$. Then $S \models \Box A[s]$, since the relation $R_\Box$ from M has been made smaller and by inductive hypothesis for A.

  ii) Let $M \models \neg\Box A[s]$. By lemma 2 and the inductive hypothesis for $\neg A$ also $S \models \neg\Box A[s]$

The newly constructed model S is, by definition, based on a suitable frame. So we obtain the following:

Theorem 1 (completeness) *For each unprovable formula α of linear temporal propositional logic, there is a frame falsifying it.*

References

[1] Robert Goldblatt. *Logics of Time and Computation*, volume 7 of *CSLI Lecture Notes*. Stanford, 1987.

[2] Fred Kröger. *Temporal Logic of Programms*. Springer,Berlin, 1987.

A Note on the Frame Semantics of Modal Logic

Alexander Kurz
Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, Germany
kurz@informatik.uni-muenchen.de

Abstract

A new class of frames for modal logic is introduced. They are defined like general frames, but with a different closure condition imposed on the set of allowed valuations. Not closure under modal operators but closure under (arbitrary) unions is demanded. To compare these so-called complete frames with general frames, a common generalization—Boolean frames—is introduced. Then three main results on complete frames are presented. First, complete frames are semantically complete in the sense that every normal modal logic is characterized by a complete frame. Second, frames that are both general and complete are bisimilar to Kripke frames. Third, complete frames can be regarded as Kripke frames where the set of worlds is replaced by a multiset of worlds. We give an example of how this concept may be applied to the theory of concurrent processes.

1 Introduction

A new class of frames is introduced. They are defined like general frames (cf. van Benthem[5]), but with a different closure condition imposed on the set of allowed valuations. Not closure under modal operators but closure under (arbitrary) unions is demanded. To be more precise: A *complete frame* $(W, R, \mathcal{V})$ consists of a set of worlds W, a relation $R \subset W \times W$ and a subset $\mathcal{V}$ of the powerset of W that is closed under Boolean operators and unions, i.e., $(\mathcal{V}, -, \cup)$ has to be a complete Boolean algebra where complements are taken w.r.t. W. A general frame is defined similar but, instead of being complete, $\mathcal{V}$ has to be closed under the modal operator m (see the next section for a definition).

In this paper we will take a closer look at complete frames. Our first concern is a comparison with general frames. As the latter, complete frames are semantically complete for normal modal logics, that is, every modal logic characterized by a general frame is also characterized by some complete frame. To show this we give a construction from general frames to modally equivalent complete frames in section 3. On the other hand, not every theory of a given complete frame is characterized by a general frame. This is because there are theories of complete frames that are not closed under modal substitutions.[1] Thus, from the point of view of modal theories, complete frames may be regarded as a generalization of general frames. But from a model theoretic point of view we get a different picture: We will show in section 4 that a frame that is general and complete is bisimilar to a Kripke frame.

Next, we show that complete frames can be regarded as Kripke frames where the set of worlds is replaced by a multiset (or sequence) of worlds. This allows for frames explicitly mentioning repetition of identical worlds (states). We sketch an application to the theory of concurrent processes and temporal logic.

Before starting on these topics, a common generalization of complete and general frames—Boolean frames—is introduced and the notion of bisimulation is adapted.

2 Basic Definitions

Let us first fix the notation. We consider a language $\mathcal{L}$ of propositional modal logic built in the usual way from propositional variables $p, q, \ldots$, parentheses, the unary operator $\neg$, the binary operator $\vee$ and one unary modality $\Diamond$. The connectives $\Box, \wedge, \rightarrow$ are defined as usual. The restriction to one modality is not essential and only simplifies the notation. A *model* $M = (W, R, V)$ for this language consists of a set of worlds W, a relation R over W and a function V from proposition letters to subsets of W. $V(p)$ denotes the set of worlds where p is true. We write $M, w \models \alpha$ if the model M satisfies the modal formula α in w.[2] We say that $M \models \alpha$ iff $M, w \models \alpha$ for all $w \in W$. A *Kripke frame* $K = (W, R)$ abstracts from a concrete valuation, therefore $K \models \alpha$ iff $(W, R, V) \models \alpha$ for all valuations V. A *general frame* $G = (W, R, \mathcal{V})$ is a Kripke frame plus a restriction $\mathcal{V}$ (that we will call the *valuation range* of G) on the allowed valuations: $\mathcal{V}$ is a subset of the powerset of W and $G \models \alpha$ iff $(W, R, V) \models \alpha$ for all valuations V such that $V(p) \in \mathcal{V}$ for all proposition letters p. Such a model (W, R, V) is said to be *based on* G. Furthermore $(\mathcal{V}, -, \cup, m)$ has to form a modal algebra, i.e., $\mathcal{V}$ is closed under Boolean operations as well as under the modal operator m that is defined by $m(X) = \{y \in W : \exists x \in X : yRx\}$.[3] The (modal) theory of a frame F is the set $\{\alpha : F \models \alpha\}$. When we call two frames *equivalent*, we always mean that they have the same theory. Note that a Kripke frame (W, R) is equivalent to the general frame $(W, R, 2^W)$. In the following we will not distinguish between the general frame $(W, R, 2^W)$ and the Kripke frame (W, R).

In contrast, a **complete frame** $C = (W, R, \mathcal{V})$ may have a valuation range $\mathcal{V}$ that is not closed under the modal operator. Instead $\mathcal{V}$ has to be a complete

[1] In this respect (and some others to be explained later) complete frames are more similar to models than to Kripke frames.

[2] $M, w \models p$ iff $w \in V(p)$, Boolean connectives as usual, $M, w \models \Diamond\alpha$ iff $\exists v : wRv \;\&\; M, v \models \alpha$.

[3] X is the set of worlds where α is true iff $m(X)$ is the set of worlds where $\Diamond\alpha$ holds.

Boolean algebra. For comparison of general and complete frames we define a **Boolean frame** as a frame $B = (W, R, \mathcal{V})$ where $\mathcal{V}$ just has to form a Boolean algebra.

One aspect of these different structures is what kinds of logics they allow to describe. In contrast to models, the theory of a general frame is closed under substitutions of formulas for propositional variables. The closure under substitutions follows from the fact that the valuation range of a general frame is closed under Boolean and modal operators. It is therefore not surprising that the theory of a Boolean frame is closed only under substitutions of Boolean formulas. Moreover we have the following: The theories of Boolean frames are exactly those theories obtained from the standard formal systems for the modal logic **K** when we replace the rule of substitution by a rule allowing only substitution of Boolean formulas. More precisely, define $\models_b, \vdash_b$ as follows: $\Sigma \models_b \alpha$ iff for all Boolean frames B it holds that $B \models \Sigma$ implies $B \models \alpha$. And $\Sigma \vdash_b \alpha$ iff α may be derived from Σ using as axioms only propositional tautologies and formulas of the form $\Box(\alpha \rightarrow \beta) \rightarrow (\Box\alpha \rightarrow \Box\beta)$ and as rules only modus ponens, necessitation[4] and the substitution of Boolean formulas for propositional variables. Then

$$\Sigma \vdash_b \alpha \iff \Sigma \models_b \alpha.$$

The proof uses the canonical model of Σ and is the same as for $\Sigma \vdash \alpha \Leftrightarrow \Sigma \models \alpha$ where $\vdash$ denotes derivability with arbitrary substitutions and $\models$ is the consequence relation w.r.t. general frames (cf. van Benthem[5]).

Bisimulation between models is an important notion. Between Kripke frames it is less interesting because it degenerates to isomorphism. But in the case of Boolean (general, complete) frames bisimulation gets interesting again. An example for a general and complete frame that is bisimilar to a Kripke frame but looks rather differently is given in section 5. In order to shorten the definition we introduce the following notation: a relation $Z \subset W \times W'$ induces two functions $\overrightarrow{z} : 2^W \rightarrow 2^{W'}$ and $\overleftarrow{z} : 2^{W'} \rightarrow 2^W$ defined by $\overrightarrow{z}(X) = \{y \in W' : \exists x \in X : xZy\}$ and $\overleftarrow{z}(Y) = \{x \in W : \exists y \in Y : xZy\}$.

Def 2.1 *Let $B = (W, R, \mathcal{V})$, $B' = (W', R', \mathcal{V}')$ be two Boolean frames. A relation Z is called a* **bisimulation** *between B and B' if Z respects the usual conditions on R and $\overrightarrow{z} \circ \overleftarrow{z} = \mathrm{id}_{\mathcal{V}'}$ and $\overleftarrow{z} \circ \overrightarrow{z} = \mathrm{id}_{\mathcal{V}}$. An* **isomorphism** *is a bisimulation where Z is a bijective function.*

A consequence of this definition is that if two frames bisimulate then for any model based on one of them we can find a bisimilar one based on the other.[5] This is made possible by the condition that $\mathcal{V}$ and $\mathcal{V}'$ are isomorphic via $\overrightarrow{z}$. Therefore, it follows from the corresponding lemma on bisimulation between models that if two frames are bisimilar then they are equivalent.

[4]Necessitation means the rule $\alpha \vdash \Box\alpha$.

[5]Note that in case of complete frames the definition of isomorphism is equivalent (via theorem 5.1) to the one given in footnote 9.

3 Completeness

Here we show that complete frames are semantically complete for normal modal logics by proving that every modal logic that is characterized by a general frame is also characterized by some complete frame. The proof of the following theorem gives a construction that transforms any general frame into an equivalent complete frame. Then an example illustrating this construction is presented. (It may be helpful to look at the example before going through the proof.)

Theorem 3.1 *For any general frame there is an equivalent complete frame.*

Proof. Let $G = (W, R, \mathcal{V})$ be a general frame. We will define an equivalent complete frame $B^G = (W^G, R^G, \mathcal{V}^G)$.

For every modal formula α let $Var(\alpha)$ denote the set of propositional variables occurring in α. For every valuation V with $V(p) \in \mathcal{V}$, for all p, and every modal formula α define $\mathcal{V}_{V,\alpha}$ as the Boolean algebra generated by $\{V(p) : p \in Var(\alpha)\}$. Since $\mathcal{V}_{V,\alpha}$ is finite it is also complete and each $(W, R, \mathcal{V}_{V,\alpha})$ is a complete frame. Now define B^G as the disjoint union[6] of all the $(W, R, \mathcal{V}_{V,\alpha})$. Since completeness is preserved under disjoint unions this is indeed a complete frame. It remains to show $G \models \alpha \Leftrightarrow B^G \models \alpha$.

"$\Rightarrow$": Suppose $B^G \not\models \alpha$. That is, one of the disjoint frames of which B^G is composed, say $B = (W, R, \mathcal{V}_B)$, rejects α. It follows the existence of a valuation V with $V(p) \in \mathcal{V}_B$ and $(W, R, V) \not\models \alpha$. But since the valuation ranges of general frames are closed under Boolean operators we get $V(p) \in \mathcal{V}$. Hence $(W, R, \mathcal{V}) \not\models \alpha$.

"$\Leftarrow$": Suppose $G \not\models \alpha$. That is, there is a valuation V with $V(p) \in \mathcal{V}$ and $(W, R, V) \not\models \alpha$. Hence $(W, R, \mathcal{V}_{V,\alpha}) \not\models \alpha$ and therefore $B^G \not\models \alpha$. **QED.**

Corollary 3.2 *For any Boolean frame there is an equivalent complete frame.*

Proof. The same as above. We didn't use that the valuation ranges of general frames are closed under modal operators. **QED.**

We give an example. Let $G = (\mathbf{N}, \leq, \mathcal{C})$ be the general frame consisting of the natural numbers, the usual $\leq$ relation and the set of all finite and cofinite subsets of $\mathbf{N}$. Now we will build an equivalent complete frame from disjoint unions of "restrictions" of G. Let $B_i = (\mathbf{N}, \leq, \mathcal{C}_i)$ where $\mathcal{C}_i$ is the Boolean algebra generated by $\{\{0\}, \dots \{i-1\}\}$ for $i \geq 0$. B is defined as the disjoint union of the B_i.[7] It is not difficult to see that both frames are equivalent. The intuitive reason is the following. $\mathcal{C}$ prevents modal formulas to change their validity infinitely often.

[6] In the disjoint union $(W, R, \mathcal{V}) = \biguplus_I (W_i, R_i, \mathcal{V}_i)$ W, R are as usual. $\mathcal{V}$ contains all unions of sets of the form $\{(i, w) : w \in X\}$ for some X in $\mathcal{V}_i$.

[7] Comparing the definition of B with the one of B^G in the proof, it is conspicuous that here the disjoint union is formed of fewer frames. The difference is not essential and is only to simplify both the example and the proof.

A consequence is that $G \models \Box\Diamond\alpha \rightarrow \Diamond\Box\alpha$. This formula does not hold in the Kripke frame $(\mathbf{N}, \leq)$ because $(\mathbf{N}, \leq)$ allows counterexamples that are prevented by $\mathcal{C}$. Now, we can express the same constraint on the possible valuations using infinitely many copies of this Kripke frame and for each of them a much simpler definition of the valuation ranges. The fact that modal formulas does not change their value infinitely often is at least as immediate for B than for G. I think that completeness is easier to visualize than closure under modal operators. The reason is that completeness allows the construction of atoms as intersections of ultrafilters of worlds (see the next section).

The reader may have noted that in our example the valuation ranges of the frames B_i are closed under the modal operator m. That is, B is not only a complete frame but also a general frame. But this is an exception. Generally this disjoint-union-construction will destroy modal closure. This follows from theorem 4.2 and the incompleteness of Kripke frames. As an example consider the general frame $G = (\mathbf{N}, <, >, \mathcal{C})$, $\mathcal{C}$ as above. The appropriate language should now contain two modalities $\Diamond_<$, $\Diamond_>$. Note that $\mathcal{C}$ is closed under the operators $m_<$ and $m_>$ and that we still have $G \models \Box_<\Diamond_<\alpha \rightarrow \Diamond_<\Box_<\alpha$. Now, make the same disjoint union construction as above, that is, let B be the disjoint union of the frames $B_i = (\mathbf{N}, <, >, \mathcal{C}_i)$, $\mathcal{C}_i$ as above. Again B and G are equivalent. But the $\mathcal{C}_i$ are not closed under $m_>$. Hence B is not a general frame.

We hope to have shown that completeness is one reason why complete frames may be useful: they allow (at least in some examples) a quite natural semantics in cases where the usual Kripke frames don't suffice.

4 A Comparison of General and Complete Frames

We first characterize complete frames in terms of Kripke frames plus an equivalence relation over the set of worlds. Then we show that a frame that is general and complete is bisimilar to a Kripke frame.

Any Kripke frame (W, R) together with an equivalence relation $\sim$ over W determines uniquely a complete frame $(W, R, \mathcal{V}^\sim)$ if we define $\mathcal{V}^\sim$ to be the set of all unions of equivalence classes of $\sim$, i.e., $\mathcal{V}^\sim = \{X \subset W : \forall x, y \in W : x \sim y \;\&\; x \in X \Rightarrow y \in X\}$. Conversely, every complete frame $C = (W, R, \mathcal{V})$ determines uniquely an equivalence relation $\sim_\mathcal{V}$, where $\sim_\mathcal{V}$ is defined by the partition of W containing the atoms[8] of $\mathcal{V}$. That the atoms of the complete Boolean algebra $\mathcal{V}$ form indeed a partition of W follows from the fact that a complete field of sets is atomic (cf. Sikorski[4], chapter 25).

In a complete field of sets the atoms are exactly the intersections of the ultrafilters. Therefore they are minimal classes of worlds being forced to fulfill the same propositions. In the next section we will pursue this idea and consider the atoms as states (of a program execution, for example). There (cf. theorem 5.1)

[8] $X \in \mathcal{V}$ is an atom iff $X \neq \{\}$ and $\forall Y \in \mathcal{V} : \{\} \neq Y \subset X \Rightarrow Y = X$.

it will be important to note that the above constructions of an equivalence relation out of a valuation range and that of a valuation range out of an equivalence relation are inverse to each other. That is $\sim_{(\mathcal{V}^{\sim})} = \sim$ and $\mathcal{V}^{(\sim_{\mathcal{V}})} = \mathcal{V}$.

Now we want to show that a frame that is general and complete is bisimilar to a Kripke frame. Given a complete frame $C = (W, R, \mathcal{V})$ we define a Kripke frame $C^{\sim} = (W^{\sim}, R^{\sim}, 2^{W^{\sim}})$: $W^{\sim}$ is the partition of W induced by $\sim$ (writing $\sim$ for $\sim_{\mathcal{V}}$) and $R^{\sim}$ is defined by $[w]R^{\sim}[v] \Leftrightarrow \exists w_1 \sim w : \exists v_1 \sim v : w_1 R v_1$, where $[w]$ denotes the equivalence class of w w.r.t. $\sim$.

Proposition 4.1 *$wRv \ \& \ w_1 \sim w \Rightarrow \exists v_1 : v_1 \sim v \ \& \ w_1 R v_1$ iff $\mathcal{V}$ closed under m.*

Theorem 4.2 *Let $C = (W, R, \mathcal{V})$ be a complete frame and $C^{\sim}$ as above. Then $\mathcal{V}$ is closed under m iff the relation $Z = \{(w, [w]) : w \in W\} \subset W \times W^{\sim}$ is a bisimulation between C and $C^{\sim}$.*

We have shown that a frame that is complete and general is bisimilar to a Kripke frame. Now we will show that the converse also holds.

Proposition 4.3 *Let $(W, R, \mathcal{V})$ and $(W', R', \mathcal{V}')$ be two Boolean frames and Z a bisimulation between them. Then it holds for all $X, X_i \in \mathcal{V}$:*

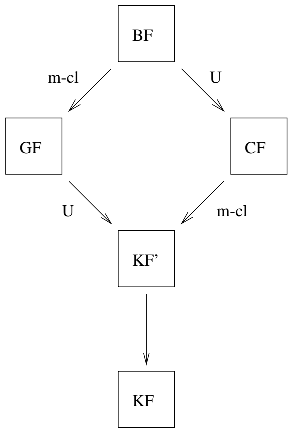
- $\overleftarrow{z}(m'(\overrightarrow{z}(X))) = m(X)$,
- $\overleftarrow{z}(\bigcup_i(\overrightarrow{z}(X_i))) = \bigcup_i(X_i)$.

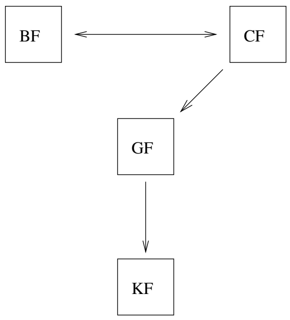
It follows that modal closure and completeness are preserved by bisimulations. We therefore have the following theorem.

Theorem 4.4 *A frame that is bisimilar to a Kripke frame is a general, complete frame.*

The interest in the class of general and complete frames may be explained as follows: As indicated above the atoms are minimal classes of worlds being forced to fulfill the same propositions. If we want to interpret the atoms moreover as states being a set of complete information, it seems sensible that identical states not only fulfill the same propositions, but also agree in all modal formulas. A sufficient condition for this is that these identical states may be identified by some bisimulation. But theorem 4.2 shows that there is a bisimulation identifying all identical states if and only if the complete frame is also a general frame.

We can summarize the results of the last two sections in two pictures drawing hierarchies of classes of frames. The first one gives a semantic interpretation in terms of closure conditions and bisimulation. The second one classifies according to classes of logics that are characterizable.





In the first picture $A \rightarrow B$ signifies $A \supset B$. The class KF' is meant to contain all frames $(W, R, \mathcal{V})$ that are bisimilar to a Kripke frame. Because of the theorems above, we have $\mathrm{KF}' = \mathrm{GF} \cap \mathrm{CF}$. An example of an interesting frame of this class is presented in section 5. 'm-cl' stands for modal closure and 'U' for completeness. In the second picture $A \rightarrow B$ signifies: for every frame in B there is an equivalent frame in A.

5 Multisets of Worlds - An Application

In this section we will see how complete frames can be understood as Kripke frames with multisets of worlds. An example of a possible application in temporal logics of concurrent programs is given.

Let us first define the new concept and only then show how it relates to complete frames. We will speak of sequences of states rather than of multisets of worlds in view of the example at the end of the section. In our terminology a sequence of states is just a mapping $s : W \rightarrow S$ where S is the set of states and W an index set. We will see later that the index set and not the set of states corresponds to the set of possible worlds in a Kripke style semantics, hence the notation. The index set may be ordered by a relation R, that is, $R \subset W \times W$. On the other hand, as the notion of state already suggests, the valuations V will be over S. What do we gain by this construction? Multiple occurrences of the same state become representable on the level of frames. Of course, on the level of models we already had the ability to force two worlds to fulfill the same propositions. But with Kripke frames this was not possible. The merits of this new structures will be illustrated in the example below.

In view of the above discussion we define a **sequence frame** to be a structure $(s : W \rightarrow S, R)$ where s is a function and $R \subset W \times W$. A *sequence model* M consists of a frame and an assignment of truth values to propositional variables. M *is based on* $P = (s : W \rightarrow S, R)$ iff $M = (s : W \rightarrow S, R, V)$ and V a function from the set of propositional variables into the powerset of S. $V(p)$ denotes the set of states where p is true. The definition of validity w.r.t. a point w in a sequence model M goes as follows:

$$\begin{array}{lll} M, w \models p & \text{iff} & s(w) \in V(p) \\ \neg, \vee & & \text{as usual} \\ M, w \models \Diamond\alpha & \text{iff} & \text{there is a } v \text{ with } wRv \ \&\ M, v \models \alpha \end{array}$$

Only the first clause of the definition has been changed as compared to the validity definition of Kripke models. As usual, $M \models \alpha$ iff $M, w \models \alpha$ for all $w \in W$ and $P \models \alpha$ iff $M \models \alpha$ for all sequence models M based on P.

Now, we will show that sequence frames offer just a different notation for complete frames. Recall from section 4 that every complete frame $C = (W, R, \mathcal{V})$ is uniquely determined by the Kripke frame (W, R) and the equivalence relation $\sim_{\mathcal{V}}$. $W^{\sim} = \{[w] : w \in W\}$ has been defined as the respective set of equivalence classes. Given $C = (W, R, \mathcal{V})$, we define the corresponding sequence frame $C^s = (s : W \rightarrow W^{\sim}, R)$, where s is given by $s(w) = [w]$. Conversely, given a sequence frame $P = (s : W \rightarrow S, R)$ we define the corresponding complete frame $P^c = (W, R, \mathcal{V}^c)$ where $\mathcal{V}^c$ contains all unions of equivalence classes of $w \sim v \Leftrightarrow s(w) = s(v)$.

Theorem 5.1 *Given a complete frame $C = (W, R, \mathcal{V})$ and a sequence frame $P = (s : W' \rightarrow S', R')$ it holds:*

- *$C^s \models \alpha$ iff $C \models \alpha$ and $P^c \models \alpha$ iff $P \models \alpha$,*
- *$(P^c)^s$ isomorphic to P and $(C^s)^c$ isomorphic to C.*

The proof uses the characterization of complete frames as Kripke frames plus an equivalence relation (see section 4) and the fact that a sequence frame $P = (s : W \rightarrow S, R)$ is determined up to isomorphism[9] by the Kripke frame (W, R) and the equivalence relation $w \sim v \Leftrightarrow s(w) = s(v)$.

Next, we will give an example of the use of complete frames. We consider propositional linear temporal logic (LTL). A Kripke frame for LTL is $N = (\mathbf{N}, \leq, \sigma)$, where $\mathbf{N}$ is the set of natural numbers, $\leq$ as usual and σ the successor relation. The language contains two modal operators $\Box$ and $\circ$ that are interpreted by the relations $\leq$ and σ respectively. LTL itself is defined as the set of all modal formulas valid in N (for details see e.g. Goldblatt[1]). Now, in computer science, a well-known problem with this semantics is that it is not stutter invariant (see, e.g., [3] or [2] for a discussion). That is, given a formula α and a model M for α there may be a stutter variant M' of M with $M' \not\models \alpha$. A stutter variant of M is a model that differs from M only because some repetitions of identical

[9] An *isomorphism between sequence frames* $(s : W \rightarrow S, R)$, $(s' : W' \rightarrow S', R')$ is a bijection $f : W \rightarrow W'$ such that f is an isomorphism between (W, R) and (W', R') and furthermore $s(w) = s(v) \Leftrightarrow s'(f(w)) = s'(f(v))$.

states have been inserted or eliminated. A semantics that is not stutter invariant causes problems concerning composition and refinement. Now, well-known again, this defect is not an inherent property of LTL itself but depends strongly on the semantic assumptions. For example, LTL may be rendered stutter invariant by replacing the relation σ by a relation σ_s with the following property: $w\sigma_s v$ if, given w, v is the next point whose state is different and, if no such state exists, $w\sigma_s(w+1)$. To prove that these two semantics are indeed equivalent the easiest way is the following: Let us call LTL$_s$ the logic characterized by the class of sequence frames $(s : \mathbf{N}\rightarrow\mathbf{N}, \leq, \sigma_s)$. Now, identity of LTL and LTL$_s$ follows easily from the fact that each frame $P = (s : \mathbf{N}\rightarrow\mathbf{N}, \leq, \sigma_s)$ is bisimilar (in the sense of definition 2.1) to the frame $P' = (s' : \mathbf{N}\rightarrow\mathbf{N}, \leq, \sigma)$ where s' is defined as the sequence that is obtained from s by eliminating all finite repetitions of identical states.

An objection to this example might be that the well-established bisimulation between models is enough to prove the identity of LTL and LTL$_s$. So why should we work with sequence frames? The relationship of sequence frames to models is similar to that of Kripke frames to models. One reason of their usefulness is that they offer a convenient way to define the appropriate classes of models. Often, one prefers to work with classes of models being based on certain frames with some agreeable properties. On the other hand, as the example suggests, there are cases where Kripke frames do not allow to define the models one is interested in. Sequence frames may repair this defect. For example, here, sequence frames allow a notion of indistinguishability between worlds which is independent of the interpretation of any particular proposition. Another reason why sequence frames are useful is that the notion of a bisimulation is lifted to the level of frames. For example, the class of frames bisimilar to the Kripke frame $(\mathbf{N}, \leq, \sigma)$ contains 'stutter invariant' frames $(s : \mathbf{N}\rightarrow\mathbf{N}, \leq, \sigma_s)$.

6 Conclusion

Besides general frames, complete frames are a second generalization of the concept of a Kripke frame. This can also be formulated in the following way: There are classes of Kripke models that are based on a complete frame but on no Kripke frame. One consequence of this fact was that we get semantical completeness (section 3). Another was that the theories may not be closed under (modal) substitutions (section 2). When we want to avoid this, one possibility is to consider frames that are complete and general.[10] We showed that these are exactly those (Boolean) frames that are bisimilar to Kripke frames (section 4). We therefore can transfer all we know about the logics of Kripke frames. On the

[10]Another is to work with classes of complete frames whose theories are closed under substitutions.

other hand, we have the possibility to define frames with properties (like stutter invariance) that are not shared by Kripke frames (section 5).

Another interesting point is the way we can look at complete frames. The starting point was a rather abstract variation of the closure condition of the valuation ranges of Boolean frames. But then we saw that we can conceive complete frames as Kripke frames plus an equivalence relation on the set of possible worlds (section 4), or as Kripke frames with the set of worlds replaced by multisets of worlds (or sequences of states). The reason was that completeness of a field of sets guarantees that the algebra is atomic, that is, we can rebuild any set from atoms. These atoms generally do not exist in general frames. Therefore, one has to work with subsets of the powerset of the set of worlds to restrict the possible valuations. Section 5 showed how atoms may be used to give a more implicit definition of these restrictions. An application was the interpretation of atoms as states of program executions.

References

[1] Robert Goldblatt. *Logics of Time and Computation*, volume 7 of *CSLI Lecture Notes*. Stanford, 1987.

[2] Yonit Kesten, Zohar Manna, and Amir Pnueli. Temporal verification of simulation and refinement. *Lecture Notes of Computer Science 803*, 1994.

[3] Leslie Lamport. The temporal logic of actions. *ACM Trans. on Prog. Languages and Systems 16(3)*, pages 872–923, May 1994.

[4] Roman Sikorski. *Boolean Algebras*. Springer, 1969.

[5] Johan van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, Napoli, 1983.

An Introduction to Java

Thomas Feeß
Department of Computer Science, University of Munich
Oettingenstr. 67, 80538 Munich, FRG
feess@informatik.uni-muenchen.de

Abstract

The introduction of Java applets has taken the World Wide Web by storm. Information servers can customize the presentation of their content with server–supplied code which executes inside the Web browser. The first part of this paper contains a survey of the Java programming language, Java applets and the bytecode generated by the Java compiler. Two examples will demonstrate the use of Java: Implementing a bounded buffer and embedding a clock into a HTML page.
The second part of this paper deals with the security system of Java: We discuss the components responsible for the safe execution of Java applets and present some examples of known security bugs of Java. We will finally point out some shortcomings of Java which increase the danger of security holes.

1 Introduction

A brief introduction to the Java language and the programming of Java applets can be found in [4]. The historical survey of the development of Java is taken from [8].

1.1 History of Java

The genesis of Java can be traced back to December 1990 when Sun launched a project called "Green" which was initiated by Patrick Naughton. The aim of this project was to develop new software technology for chip–based consumer electronics.

Part of this project was the design of a simple, robust and portable programming language. James Goslin, the father of Java, started with C^{++}. But soon he realized that C^{++} was not suited for this task and Goslin began to develop a new programming language called "Oak". When searching for a trademark, the name "Oak" was later dropped in favour of "Java" (note that "Java" is not an abbreviation but has been adopted from slang).

In 1994 Sun realized that the main design goals of Java made this language suited for exchanging executable code over the Internet and the project was redirected at the Web. Patrick Naughton wrote "WebRunner", a prototype of a browser capable of running Java code. This browser was later re–implemented in Java and formally announced as "HotJava" browser by Sun in May 1995.

Shortly after the release of Java and HotJava in mid–1995 Netscape licensed Java and incorporated it into version 2.0 of their Web browser. Meanwhile other browsers (e.g. Mosaic) have become Java–compatible too.

1.2 Portability of Java

Java programs are portable at the binary level. This means that code generated by a Java compiler (the so–called *bytecode*) can be executed on every platform where the system libraries and the run–time environment of Java have been installed. Java bytecode is portable because

- it is not written in an existing machine language but it is aimed to run on a *virtual* machine (the Java virtual machine JVM).
- all system libraries are linked dynamically on execution.

The introduction of a virtual machine means that, in order to execute a Java program, the bytecode either has to be interpreted or has to be compiled into the native machine language. In most cases, the bytecode will be interpreted.

1.3 Enhancing Browser Capabilities through Java Applets

Substantial effort has been spent to make the HTML language more expressive (e.g. by adding tables or frames). But for many applications, like animated graphics or interactive user input, HTML is still insufficient.

Java grew popular because it provides a simple, yet powerful way to enhance the capabilities of Web browsers. This is achieved by embedding executable Java bytecode (so–called *Java applets*) into a HTML page. When viewing the Web page, the browser loads the applet from a (possibly remote) host, using a specified URL, and executes the bytecode on the client host. (Note that you need not install the Java system classes nor the Java interpreter as they are integrated into a Java–compatible browser.) Therefore Java programs take on two slightly different forms:

- A *Java application* is a stand–alone Java program.
- *Java applet* denotes a Java program embedded in a HTML page.

Besides having different calling conventions, applications and applets differ in their security restrictions: As Java applets loaded over the Internet are considered untrusted, their execution is monitored and their access to system resources, like files or network connections, is restricted. In contrast to this, Java applications are executed like trusted native user code.

2 Programming in Java

A detailed specification of the Java programming language can be found in [6]. [3] and [7] contain a complete reference manual of the Java API. [1] provides many examples of Java programs and explains how to write Java applets. Details of the bytecode and the Java virtual machine (JVM) are described in [5].

2.1 Java Programming Language

Java is an object–oriented programming language which syntactically resembles C^{++}. However, some insecure constructs (like memory access through pointers) have been removed. Furthermore, Java supports parallel programming through multiple threads. The multithreading facilities of Java can be used by a browser to instantiate a separate thread for each applet on a HTML page.

2.1.1 Java Type System

Java is a strongly–typed object–oriented language i.e. each expression is assigned a type at compile–time. The run–time type of an expression is guaranteed to be a subtype of its compile–time type. Besides primitive types (like `boolean`, `int` and `float`) and array types, the type hierarchy of Java consists of

- *classes*
- *interfaces* (see below)

In contrast to C^{++}, Java only supports *single inheritance* in order to avoid semantic problems associated with multiple inheritance. Thus, the class hierarchy of Java constitutes a tree having the class `Object` as its root where `Object` supplies basic methods e.g. for cloning objects or transforming objects into a string.

Classes may be *abstract* which means that they declare methods whose implementation is delegated to subclasses. However, it is not possible to declare generic classes like `Stack(T)` where `T` denotes a type parameter (in C^{++} generic classes are built using the `template` construct).

Interfaces were introduced to provide a means for constructing a rich type hierarchy forming an arbitrary acyclic graph. An interface resembles a completely abstract class i.e. it contains only method declarations and no method definitions. Therefore no inheritance is associated with interfaces and so the interface hierarchy may form an arbitrary acyclic graph. Moreover a class may be declared to implement one or more interfaces in which case the class must provide all methods listed in the interfaces.

As an example for the use of interfaces, consider the constructor method `Thread(Runnable r)` instantiating a thread over an object `r` of the interface type `Runnable`. The interface `Runnable` in turn declares a single method `run` (the body of the thread to be called at start–up). In this way a thread can be constructed on objects `r` belonging to otherwise unrelated classes provided they implement `Runnable` (and thus supply the method `run`).

2.1.2 Insecure Constructs removed from C^{++}

Because of security aspects, the Java programming language does not contain any construct which allows uncontrolled memory access. Thus, pointer types and dereferencing operators have been removed from C^{++}. In fact, the only way to access memory locations in Java is through objects which are treated as pointers i.e. objects are always passed per reference. Moreover accessing an array element automatically enforces an index check.

The task of memory management has been shifted from the programmer to the Java run–time environment. For example, a garbage collector (normally running as a low–priority thread) periodically cleans up unused objects using a mark–sweep algorithm. Java provides no means for the programmer to explicitly dispose the memory used by an object.

2.1.3 Multithreading

Java supports *multithreading* i.e. different flows of control sharing the same address space may be started by a program. The multithreading facilities of Java are used by the Java run–time environment to run the garbage collector and by Java–compatible browsers to start several applets in parallel. Moreover a Java applet may delegate a time–consuming task (like loading an image) to a separate thread.

To create a thread, it is necessary to instantiate a subclass of the Java system class `Thread`. Invoking the inherited method `start` on the thread object will start a separate flow of control thereby calling the method `run` which implements the body of the thread. Threads may be suspended or stopped from outside by invoking the corresponding methods. It is possible to organize threads into arbitrarily nested *thread groups* whose members can be accessed as a whole.

Threads synchronize themselves through *monitors* i.e. some methods of a class may be declared as monitor operations (using the keyword `synchronized`) which means that these methods run under mutual exclusion. Each instance of the class is treated as a separate monitor and a call of a monitor method acquires a lock on this object. Note that monitors in Java are *re–entrant* so that a thread does not block when entering a monitor he already holds (thus decreasing the danger of deadlocks).

A thread may temporarily leave a monitor using the method `wait` but it is not possible to wait on different boolean conditions (the notion of a "condition variable" is unknown to Java). Therefore for any monitor object there exist only two waiting queues: One queue of threads waiting to enter the monitor and another queue of threads having temporarily left the monitor using the `wait` method. Invoking the `notify` method wakes up some thread in the second queue which will then compete for the monitor.

2.1.4 Example: Bounded Buffer

The following example (the implementation of a bounded FIFO buffer by an array) demonstrates the synchronization facilities of Java. It declares a single class `Buffer` exporting the methods `put` and `get`. The implementation of the buffer is thread–safe so that multiple threads may simultaneously try to add and remove elements. Therefore `put` and `get` are declared to be monitor operations using the keyword `synchronized`.

Both methods, `put` and `get`, will block when the appropriate precondition (not `is_full` resp. not `is_empty`) is not satisfied in which case the thread leaves the monitor and suspends itself using the `wait` method. Note that after resuming `put` and `get` have to test their preconditions again because of the lack of condition variables. Before leaving the monitor, `put` and `get` invoke the method `notify` to wake up some suspended thread.

```
/* Bounded FIFO Buffer (thread-safe) */
class Buffer
{
  /* Instance variables (partially initialized) */
  private int capacity;             /* Buffer capacity */
  private Object[] queue;           /* Array queue[0,...,capacity - 1] */
  private int nr_elements = 0;      /* Nr of elements currently in "queue" */
  private int first = 0, last = -1; /* Index of first and last element */

  /* Constructor */
  public Buffer(int capacity)
  {
    this.capacity = capacity;
    this.queue = new Object[capacity];
  }

  private boolean is_empty()
  {
    return(nr_elements == 0);
  }

  private boolean is_full()
  {
    return(nr_elements == capacity);
```

```
  }

  /* Monitor method: Put an object into the buffer */
  public synchronized Buffer put(Object element)
  {
    /* Leave monitor and wait if the buffer is full */
    while (is_full())
      try wait(); catch (InterruptedException e);
    /* Append object to buffer */
    queue[++last] = element;
    nr_elements++;
    if (last >= capacity - 1) last -= capacity;
    /* Wake up some thread who has left the monitor */
    notify();
    return(this);
  }

  /* Monitor method: Get next object from buffer */
  public synchronized Object get()
  {
    Object result;

    /* Leave monitor and wait if the buffer is empty */
    while (is_empty())
      try wait(); catch (InterruptedException e);
    /* Remove object from buffer */
    result = queue[first++];
    nr_elements--;
    if (first >= capacity) first -= capacity;
    /* Wake up some thread who has left the monitor */
    notify();
    return(result);
  }
}
```

2.2 Java Applets

The popularity of Java stems from the possibility to incorporate executable Java bytecode (Java applets) into HTML pages thus using the GUI facilities of Java to enhance the appearance and the interactivity of the Web page.

2.2.1 Embedding Applets in a HTML page

A HTML page may contain one or more URLs pointing to applets which will be loaded by the browser and executed on the client host when viewing the Web page. An applet may use the Java system classes provided by the browser or it may load applet–specific classes from the host it originated from.

Applets in HTML are marked by the `APPLET` tag where the attribute `CODE` contains the name of the corresponding bytecode and `CODEBASE` describes the location of the applet (i.e. `CODEBASE` contains an absolute or relative URL). For each applet a specified area on the HTML page is reserved as a graphical interface of the applet. The size of this area is described by the attributes `WIDTH` and `HEIGHT`. The position of this area may vary as the browser is responsible for the layout of the Web page.

For each applet a HTML page may contain additional parameters which are passed as strings. Moreover an applet may communicate with other applets on the *same* Web page by invoking methods on them. Note, however, that there is no way to access applets on *other* HTML pages.

The bytecode of an applet must provide a subclass of the Java system class `Applet` from which it inherits the methods `init`, `start`, `stop` and `destroy` which mark the life–cycle of an applet. After loading the bytecode, the browser creates a single instance of the class of the applet and invokes the methods `init` and `start` on it. The method `stop` is called when the user temporarily leaves the Web page. Normally the applet will then suspend execution although it is not forced to. After returning to this Web page, `start` is called again. Finally the browser invokes the `destroy` method on the applet when removing the HTML page from the stack of the browser.

2.2.2 Example: Displaying the Time of the Day

The following HTML page contains an applet displaying the time of the day:

```
<HTML>
<BODY>

The following applet displays the time of the day:
<APPLET CODE=Time_Applet.class WIDTH=80 HEIGHT=30>
</APPLET>
It starts a thread which updates the time every 500 ms.

</BODY>
</HTML>
```

The file `Time_Applet.class` (having the same location as the Web page) contains the bytecode of the applet. When initializing, the applet creates and starts a thread `time_thread` which periodically (every 500 ms) repaints the applet by invoking the `repaint` method which in turn calls `paint`. When the user temporarily leaves the HTML page, the applet suspends `time_thread`. Below is the source code containing the classes `Time_Applet` and `Time_Thread`:

```
/* Import some classes */
import java.applet.Applet;
import java.util.Date;
import java.awt.Graphics;
import java.lang.*;

/* Applet displaying the time of the day */
public class Time_Applet extends Applet
{
  /* Constant: Time in ms between successive updates of time */
  static final int INTERVAL = 500;

  /* Thread which periodically repaints the applet */
  private Time_Thread time_thread;

  /* Initialization of the applet */
  public void init()
  {
    time_thread = new Time_Thread(INTERVAL, this);
    time_thread.start();
  }

  /* Called when (re)entering the Web page of the applet */
  public void start( )
  {
    time_thread.resume();
  }

  /* Called when leaving the Web page of the applet */
  public void stop()
  {
    time_thread.suspend();
  }

  /* Called when cleaning up the applet */
  public void destroy()
  {
    time_thread.stop();
```

```
  }

  /* Called when (re)painting the applet */
  public void paint(Graphics g)
  {
    Date date = new Date();
    String date_string =
      date.getHours() + ":" + date.getMinutes() + ":" + date.getSeconds();
    g.drawString(date_string, 0, 30);
  }
}

/* Thread which periodically repaints an applet */
class Time_Thread extends Thread
{

  /* Time in ms after which the applet is repainted */
  private int interval;

  /* Applet to repaint */
  private Applet applet;

  /* Constructor */
  public Time_Thread(int interval, Applet applet)
  {
    this.interval = interval;
    this.applet = applet;
  }

  /* Called when starting the thread */
  public void run()
  {
    while (true)
    {
      try sleep(interval); catch (InterruptedException e);
      applet.repaint();
    }
  }
}
```

2.2.3 Applet Restrictions

As applets loaded from a (possibly untrusted) site are executed on the client host, it is necessary to restrict their capabilities. In fact, an applet is prevented from

- reading and writing arbitrary files
- starting programs on the client host
- opening network connections except to the host that it came from
- reading arbitrary system properties (like username or operating system)

The restriction of network connections to the host from which the applet originated prevents applets from by–passing a firewall. Applets which have been loaded into a LAN may only communicate to a host outside the firewall.

2.3 Java Bytecode

The secret of the portability of Java programs is the Java bytecode generated by the Java compiler as the bytecode has to be executed on different platforms. To achieve portability, the Java virtual machine (JVM) has been introduced. The executable part of the bytecode consists of instructions for the JVM. To execute the bytecode on a specific platform, it is either compiled into the native machine language or (more likely) it is interpreted. In many aspects the JVM resembles a real CPU:

- It is stack–based i.e. all parameters, local variables and intermediate results are placed onto the stack.
- It supports fixed– and floating–point arithmetic using standard internal representations of numbers.
- Control flow is driven by conditional and unconditional jumps or by subroutine calls.

However, some design issues concerning the layout of objects, the management of the heap and the way how to implement dynamic linking of methods have not been specified for the JVM. These elements remain abstract and are left to the implementor. For example:

- Objects are processed as a primitive data type "handle" which may be thought of a pointer to the heap.
- The JVM accepts a statement `new` which places a handle to a newly allocated uninitialized object on the stack.

- Methods are called *symbolically* using a pointer to the name of the method. (However, this only applies to the first call. For efficiency reasons the corresponding bytecode statement is replaced by a "quick" variant as soon as the method name has been resolved.)

To securely run Java applets, a static type check has to be performed on the bytecode. Therefore the bytecode contains more type information than necessary to execute:

- The bytecode contains the complete signature (including access rights) of every variable and method used.

- The `checkcast` statement can be used to perform type checking at run–time

As an example, consider the bytecode generated from the method `is_empty` in the bounded buffer example of section 2.1.4:

Source Code	Bytecode
`private boolean is_empty()`	`Method boolean is_empty()`
`{`	`0 aload_0`
`return(nr_elements == 0);`	`1 getfield <Field Buffer.nr_elements I>`
`}`	`4 ifeq 9`
	`7 iconst_0`
	`8 ireturn`
	`9 iconst_1`
	`10 ireturn`

`aload_0` places the zeroth parameter on the stack which (by convention) is a handle to the current object `this`. `getfield` uses this handle to access the instance variable `nr_elements` whose value replaces `this` on the stack. Note that this operation takes as parameter the class and name of the variable (`Buffer.nr_elements`). Furthermore, the bytecode contains the signature of the variable (`Field` of type `I` = `integer`). Dependent on the value of the variable (`ifeq`) the value 1 = `true` or 0 = `false` is pushed on the stack (`iconst_1` resp. `iconst_0`). Finally `ireturn` leaves the method by returning an integer on the stack.

3 Applet Security

As Java applets loaded from an untrusted site are executed on the client host (potentially possessing all access rights of the user), it is vital to restrict their capabilities. The security system of Java must guarantee that running applets are not able to damage the client host e.g. by removing files or accessing secret information.

We give a survey of the security system of Java (thereby describing its components) and present some examples of known security holes of Java. Finally we discuss some shortcomings of Java which will probably cause further security bugs. Much of the material presented in this section can be found in [2]. [9] provides details of the implementation of the bytecode verifier.

3.1 Java Security System

The security system of Java consists of three components:

- the *bytecode verifier*
- the *class loader*
- the *security manager*

The bytecode verifier and the class loader constitute the lower level of the security system. Together they assure that executing the bytecode never causes the JVM to fall into an unsafe state (e.g. by passing parameters of the wrong type or accessing private variables from outside a class). On a higher level the security manager monitors all accesses to security–relevant resources (files, threads or system properties) possibly denying the access.

The bytecode verifier is part of the Java run–time environment and is automatically invoked when transforming a raw byte stream into a Java class. In contrast to this, class loader and security manager have to be implemented by the browser thus permitting a browser–specific security policy. For this reason the Java system classes `ClassLoader` and `SecurityManager` have been provided. The browser is expected to subclass and instantiate these classes.

3.1.1 Bytecode Verifier

Besides checking the format of the bytecode, the main task of the bytecode verifier is to perform a static type check on the bytecode before it is executed. This is necessary because we can not presume that the bytecode loaded from an untrusted site has been generated by a compiler. In detail the verifier assures that

- methods and JVM instructions are called with arguments of an appropriate type on the stack.
- access restrictions (e.g. private and protected variables) are not violated.
- the operand stack (i.e. that part of the stack frame reserved for intermediate results) does not over– or underflow.

To achieve this goal, a data–flow analysis is performed on the bytecode using the information about the signature of methods and variables contained in the bytecode. For each code position the verifier determines the layout of the current stack frame (i.e. the number and static type of parameters, local variables and intermediate results). The layout of the stack frame must be independent of the execution path by which the bytecode position is reached. Note that this places some restrictions on the bytecode. For instance, the verifier will not accept a bytecode program pushing elements on the stack in a loop.

3.1.2 Class Loader

The class loader is responsible for dynamically loading classes needed by an applet. Furthermore, it is the task of the class loader to cache classes already loaded. The class loader distinguishes between

- system classes of Java which are loaded from the client host.
- applet–specific classes which are loaded from the host the applet originated from.

When loading an applet, the browser creates an instance of a subclass of `ClassLoader` which contains the address of the host the applet was loaded from. The applet will then be marked with this class loader (which is done automatically when the class loader transforms the byte stream of the applet into a class). The run–time environment will call the class loader of the applet when it encounters a class name which has not yet been resolved. Note that classes loaded from the client host are marked with a NULL class loader. This provides the criterion by which Java distinguishes between trusted (class loader = NULL) and untrusted (class loader $\neq$ NULL) classes.

The class loader has to assure that a Java system class may not be overridden by an applet–specific class sharing the same name. Therefore, when resolving a class name, the class loader must search on the client host *before* consulting the applet host. In a similar way the class loader separates the name spaces of applets loaded from different hosts. As the class loader is responsible for the name space of an applet, an applet is not allowed to create an instance of `ClassLoader`.

3.1.3 Security Manager

Every time an applet tries to access a security–relevant resource, the associated Java system class invokes the security manager. Examples of such accesses are

- reading and writing a file
- opening a network connection
- suspending a thread

- reading a system property (username, operating system etc.)
- opening a top–level window

In each of these cases the security manager has to decide whether to grant the access or not. If the access is denied, the security manager throws a security exception. Normally the security manager will base its decision on the content of the call stack.

By creating an instance of a `SecurityManager` subclass, the browser has the ability to implement its own security policy. For instance, the security policy of the Netscape browser is very restrictive: An applet may not access any file on the client host. Furthermore, an applet can read only a limited number of system properties. In contrast to this, the HotJava browser uses user–definable access–control lists.

3.2 Known Security Bugs

Since Java was released, a number of security flaws have been discovered. Although being mere implementation errors, we present two examples of known security bugs in order to give a flavour of the attacks possible and to shed some light on shortcomings of Java.

3.2.1 Domain Name Service (DNS) Attack (January 1996)

As stated in section 2.2.3, an applet may only open a TCP/IP connection back to the server it was loaded from. While this policy is sound, it was not uniformly enforced by the security manager. The policy was enforced as follows:

1. Get all the IP addresses of the hostname that the applet came from.
2. Get all the IP addresses of the hostname that the applet is attempting to connect to.
3. Allow the connection iff any address in the first set matches any address in the second set.

The problem occurs in the second step: An applet can ask to connect to any host on the Internet so it can control which DNS server supplies the second list. By installing his own DNS server, an attacker is able to supply any list of IP addresses for step 2 thus allowing the applet to connect to any desired host. In this way a malicious applet running behind a firewall can attack any machine on the LAN.

This bug has been fixed by now: The security manager now strictly takes care that an applet may only open a connection to the same IP address from which it was loaded.

3.2.2 Verifier Bug (March 1996)

As noted in section 3.1.2, an applet may not create an instance of `ClassLoader`. This is checked by the constructor of `ClassLoader`. According to the specification of Java, this constructor has to be called directly or indirectly when instantiating a subclass of `ClassLoader`. Because the bytecode verifier did not strictly enforce this specification, the constructor call could be by–passed thus allowing an applet to create its own class loader.

Creating a malicious class loader gives an attacker the ability to defeat the type system of Java. Assume that classes `A` and `B` both refer to a class named `C`. A class loader could resolve `A` against class `C` and `B` against class `C'`. If an object of class `C` is allocated in `A`, then passed as an argument to a method of `B`, the method in `B` will treat the object as having a different type `C'`. If the fields of `C'` have different type or different access modifiers (`public`, `private` or `protected`), then the type safety of Java is defeated.

As a consequence, this bug allows attackers to get and set the value of *any* non–static variable and call *any* method (including arbitrary native machine code). Sun has promised to fix this bug in the next release of Java.

3.3 Security–relevant Shortcomings of Java

Although all known security bugs of Java are caused merely by implementation errors, the design of Java also possesses some shortcomings which increase the risk of further security holes. We will discuss some of these shortcomings concerning the programming language, the bytecode and the implementation of the security system of Java.

3.3.1 Shortcomings of the Programming Language

The name space of Java is actually flat as the package system of Java provides only basic modules and these modules cannot be nested (although the name space superficially appears to be hierarchical). This means, for example, that access to an internal variable of the security manager is only denied because this variable is declared `protected`. Properly nested modules would provide a better means for the programmer to limit the visibility of security–critical components.

Despite of having removed most insecure constructs from C^{++}, Java still offers some dangerous features to the programmer. For example, methods may be invoked on objects which have not yet been fully initialized. As large parts of the system classes of Java are written in Java, such dangerous constructs should be forbidden.

3.3.2 Shortcomings of the Bytecode

As Java applets are distributed as bytecode programs, the security of Java heavily depends on the bytecode verifier. Unfortunately the bytecode and the JVM lack a formal semantics. Therefore we can neither formally specify the properties of "secure" bytecode nor can we prove that the verifier only accepts secure bytecode programs.

As stated in section 3.1.1, the main task of the bytecode verifier is to perform a static type check on the bytecode which is normally part of the front end of a compiler. The linear form of the bytecode makes type checking a difficult task requiring a data–flow analysis which is further complicated by the possible raise of exceptions. Imposing a tree–like structure on the bytecode together with compositional typing rules would greatly simplify the bytecode verifier.

3.3.3 Shortcomings of the Implementation of the Security System

Although the security system of Java seems to be sound, it suffers from the fact that the security–critical components of Java are spread among the system classes and cannot be localized in a small, simple and verifiable security kernel. For instance, the constructor of `ClassLoader` takes care that an applet may not create an instance of this class. However, this task should be delegated to the security manager.

Furthermore, the security policy of Java is not strictly enforced. For example, the programmers of the system classes of Java are responsible for calling the security manager at the right moment. A syntactical construct which enforces the call of the security manager seems to be more appropriate.

Finally the Java system does not define any auditing capability. If we wish to trust a Java implementation that runs bytecode downloaded across a network, a reliable audit trail is a necessity. The level of auditing should be selectable by the user or system administrator. As a minimum, files read and written from the local file system should be logged, along with network usage.

4 Conclusion

Much effort has been spent into the design of the security system of Java. However, some shortcomings still remain. Therefore substantial changes of the language, the bytecode and the run–time environment are necessary to build a higher–assurance system.

The presence of flaws in Java does not imply that competing systems are more secure. If the same level of scrutiny had been applied to competing systems, the results would have been similar. Execution of remotely–loaded code is a relatively new phenomenon and more work is required to make it safe.

References

[1] M. Campione and K. Walrath. The Java Tutorial. To be pulished by Addison–Wesley in summer 1996. Draft available via *ftp://ftp.javasoft.com/-docs/tutorial.ps.tar.Z*, March 1996.

[2] D. Dean, E.W. Felten, and D.S. Wallach. Java Security: From HotJava to Netscape and Beyond. To appear in IEEE Symposium on Security and Privacy. Available via *http://www.cs.princeton.edu/sip/pub/secure96.html*, May 1996.

[3] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates Inc., 1996.

[4] U. Schneider. Applets, schöne Applets. *iX*, pages 62–68, May 1996.

[5] Sun Microsystems. *The Java Virtual Machine.* Available via *ftp://-ftp.javasoft.com/docs/vmspec.ps.gz.*

[6] Sun Microsystems. *The Java Language Specification*, December 1995. Available via *ftp://ftp.javasoft.com/docs/javaspec.ps.tar.Z.*

[7] Sun Microsystems. *Java API Documentation*, May 1996. Available via *ftp://-ftp.javasoft.com/docs/JDK-1_0_2–apidocs.tar.Z.*

[8] Java: The inside story. Available via *http://www.sun.com/sunworldonline/-swol-07-1995/swol–07–java.html*, July 1995.

[9] F. Yellin. Low Level Security in Java. Available via *http://www.javasoft.com/-java.sun.com/sfaq/verifier.html.*

Transformational Design of Distributed System Services

Marjeta Pučko
Jožef Stefan Institute, University of Ljubljana
Jamova 39, SI-1001 Ljubljana, Slovenia
marjeta.pucko@ijs.si

Abstract

A formal transformational approach to design of services in distributed systems based on Intelligent Network architecture is suggested in the paper. The approach offers composition transformations to formally and systematically compose atomic actions into partial and complete service specifications, and behaviour-preserving transformations for distribution and redundancy handling. We define a nondeterminism-reducing transformation in detail and illustrate it by an example. Developing the approach, we were motivated by transformational design methods for different types of architectures, particularly for hardware architectures.

Keywords: distributed systems, Intelligent Network architecture, formal specification, behaviour-preserving transformations, nondeterminism-reducing transformation

1 Introduction

Design of services for distributed systems is becoming an increasingly complex and time-consuming task. Furthermore, rapidly growing needs for new services on the market, as for example telecommunication services, often allow not to validate a new service carefully. An automated approach to derive implementations of services from their high level specifications by transformations leads to highly reliable software and may significantly reduce development costs.

This paper focuses on the design of Intelligent Network (IN) services which are typically composed of standardized service-independent building blocks (SIBs) [12], [4]. Using this service creation concept, users designing IN services need not to be familiar with all details of an IN network. The design process can be effectively automated by using formal service specifications at various levels of abstraction and behaviour-preserving transformations between them.

The idea of describing SIBs in the formal language LOTOS has originally been offered by Cheng and Jackson [3]. In their approach, however, LOTOS is used only in the initial service specification phase to formally describe SIBs (called functional components in [3]) and their compositions in SLPs (Service

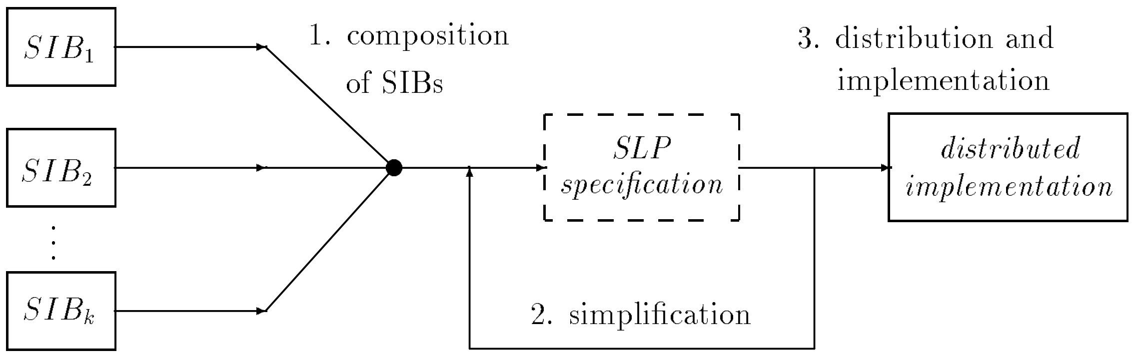


Figure 21: Main phases of the suggested approach

Logic Programs). In the service design phase, an SLP specifying an IN service is described by Cheng and Jackson in the less abstract SDL language.

Motivated by the work of [3] and formal transformational approaches in other types of architectures where correctness of an implementation is achieved by construction (approaches of [5], [10] and [1] for hardware system synthesis, and [11] and [2] for transformational design of software systems), we have developed a transformational approach for specification and design of IN services. It consists of two composition transformations to formally and systematically compose atomic actions into SIB and SLP specifications, and of several behaviour-preserving transformations to eventually simplify SLP specifications, to distribute and implement them. The main phases of our approach are shown in Figure 21.

We describe the formal background of the approach in Section 2 and define a set of composition, redundancy-handling and distribution transformations in Section 3. In Section 4 we present in detail a nondeterminism-reducing transformation and illustrate it by an example. Finally, we give some conclusions and directions for future work in Section 5.

2 Formal description of system behaviour

2.1 Syntax of the SSL specification language

To represent an IN network as a distributed system offering an IN service, we adopt the well-known model of the “black box” service provider (for LOTOS described in [7]), which interacts with its environment through interaction points, in the following also called places. Interaction points are supported by underlying entities which execute atomic actions and, to provide the execution ordering specified by a service definition, exchange synchronization messages over reliable FIFO channels. A service is defined by an abstract algebraic service specification in the LOTOS-like language SSL (Service Specification Language). We constructed a language which covers basic composition modes for SIBs and is simple enough to efficiently observe causal relations between atomic actions in service expressions.

Let ";", "|||" and "[]" denote, in that order, operators of sequential, parallel and alternative composition. Let Act^{Place} specify a service primitive of type Act from a set of actions $Acts$ occurring at $Place$. Then, the concrete prefix service syntax is defined by the following production rules where $B, B_1, ..., B_N$ represent behaviour expressions:

pr1. $B \longrightarrow Act^{Place}$
pr2. $B \longrightarrow ;(B_1, B_2)$
pr3. $B \longrightarrow |||(B_1, ..., B_N)$
pr4. $B \longrightarrow [](B_1, ..., B_N)$

Rule *pr1* defines an atomic action, specifying an interaction between a service provider and its environment at particular place. Rules *pr2*, *pr3* and *pr4* define sequential, parallel and alternative composition.

Assigning to SSL an operational semantics, we generalize in [9] the rules for systematical derivation of actions from the structure of the given behaviour expression which have been defined in [7] for standard basic LOTOS using labelled transition systems. The generalized rules may be used for parallel and alternative composition operators of arity N, $N \geq 2$. We also assume that the parallel composition operator specifies only independent parallel behaviour (i.e. interleaving).

2.2 Acceptance tree semantics

Using the alternative composition operator, we enter non-determinism in SSL expressions. Therefore another type of SSL semantics is required that can accurately describe the behaviour of non-deterministic processes, interpreted over term algebras. Since execution of the service described by an SLP specification can be viewed as a testing process, we suggest to use an adapted version of acceptance trees, originally used in Hennessy's theory of testing [6]. An acceptance tree of a process models its readiness to interact with its environment.

Acceptance trees in general base on action trees where, besides edges labelled by actions from $Acts$, nodes are labelled by acceptance sets. For every action $a \in Acts$, every node in an acceptance tree A has at most one successor branch labelled by a. Every node in the tree A is therefore uniquely determined by a string str in $Acts$. Let all the strings of A compose a set $L(A)$, and the set of the successor branches of a node n be called a successor set [6] and denoted by $D(n)$. We then restrict the number of successors of a node, so that for every str in $L(A)$, $D(L(A), str)$ has to be always finite (as required also in [6]).

The acceptance sets labelling the nodes must also satisfy several criteria. For $D \subseteq Acts$, an acceptance set $\mathcal{S}$ is a nonempty set of subsets $\mathcal{S}_i$ of $Acts$, which satisfies

1. for every $\mathcal{S}_i$ in $\mathcal{S}$, $\mathcal{S}_i \subseteq \mathcal{S}$,
2. for every a in D, there is some $\mathcal{S}_i$ in $\mathcal{S}$ such that a is in $\mathcal{S}_i$.

Acceptance sets of [6] also satisfy the criteria of $\cup$-closure and convex-closure which are in our definition disregarded, since we consider this saturation requirement to be too abstract for our use in IN services design transformations. An algorithm for construction of an acceptance tree from a given service expression we give in Section 4.

A subtree sA of an acceptance tree A is defined by a node n in A and all its successors $D(n)$, including the corresponding acceptance sets. By deletion of all acceptance sets in A, we get the skeleton of A, denoted by $SK(A)$.

2.3 Event structure semantics

Since the acceptance tree semantics does not evidently show causal dependencies between service primitives, we need another type of SSL semantics which makes dependencies related to causality obvious. We suggest to use elementary event structures, as they are defined in [13].

Event structures in general consist of events modelling occurrence of actions, and relations between events. Let $\mathcal{X}$ specify a set of occurrences of atomic actions from a given service specification B where different occurrences of the same action can be distinguished by different internal labels. Let R specify a binary relation with property of partial order over the set $\mathcal{X}$, defined as follows: an event e_i causes e_j if the execution of e_i has to be completed before execution of e_j starts. We define an event structure by a partially ordered set $(\mathcal{X}, R)$. The event pairs (e_i, e_j), which are causally related because of sequential composition operators ";" in the service expression B, are elements of a set $\mathcal{R}$.

3 Transformations

We formalize the approach by a set of transformations based on the defined SSL syntax and semantics. Corresponding to the main phases of the approach, we define the three types of transformations, abstractly described by mappings from input to output semantic representations:

- *Composition transformations (mappings)* are needed to construct a SIB specification (i.e. a partial service specification) S_P from a given set of atomic actions and a set of operators, and to compose a set of selected SIB specifications into an SLP specification (i.e. a complete service specification) S_C. Let $ACT(SIB_m) = \{Act_i^{Place_y} | 1 \leq i \leq I, 1 \leq y \leq Y\}$ represent a set of atomic actions, required to describe the standardized actions of SIB_m, and $O = \{;, |||, [], (,)\}$ represent the set of SSL operators and parentheses.

Let $SIBS = \{S_{P_n} | n \geq 1\}$, $S_{P_m} \in SIBS$ specify a set of SIB specifications which have been selected to compose an SLP specification S_C. We then define composition transformations as follows:

$$T_{cp} \; : \; [ACT(SIB_m), O] \longrightarrow S_{P_m} \tag{1}$$
$$T_{cc} \; : \; [SIBS, O] \longrightarrow S_C \tag{2}$$

- *Redundancy-handling transformations* of a given SLP specification S_C are needed to reduce or add redundancy of a particular type (internal or external, as we defined in [8]). While redundancy reduction results in a simpler and easily implementable SLP specification, redundancy addition can be used for later optimizations of eventual parallel implementations of the SLP (as in the example of implementation of a hardware architecture in [1]).

  Let A represent an acceptance tree of S_C, $maxp$ a set of actions sequences which denote edges in the paths from the root of A to its leaves, $\mathcal{R}$ a set of causally related event pairs, red_type the selected type of redundancy, and re/ad the selection of removal/addition of redundancy. The redundancy-handling transformation

$$T_r : [S_C, A, maxp, \mathcal{R}, red\_type, re/ad] \longrightarrow [S_C^T, A^T, \mathcal{R}^T] \tag{3}$$

  maps S_C into a transformed SLP specification S_C^T, A into a transformed acceptance tree A^T and $\mathcal{R}$ into a transformed set $\mathcal{R}^T$. The skeleton $SK(A^T)$ of the transformed tree has to be bisimulationally equivalent to the skeleton $SK(A)$ of the original tree. A typical redundancy-handling (nondeterminism-reducing) transformation is defined more precisely in Section 4.

- *Distribution transformations* are required for logically correct distributed implementation of a given SLP specification S_C. An SIB is usually not implemented by a single physical unit [12], but distributed over a set of physical units. To distribute it, we introduce synchronization messages to provide the execution ordering of actions specified by S_C. Using the transformation

$$T_{sm} : [S_C, \mathcal{R}, \mathcal{M}] \longrightarrow S_C^T \tag{4}$$

  we introduce into S_C synchronization messages, specified by events of transmission s_j and reception r_i, $s_j, r_i \in \mathcal{M}$, where i identifies the sender of a message and j its recipient. Necessary synchronization messages are already determined by causal dependencies between events in pairs (e_i, e_j) of the set $\mathcal{R}$.

After the introduction of synchronization messages, the actual distribution of an SLP specification is performed by projecting S_C to its participating places $Place_\ell \in \mathcal{PP}$, $1 \leq \ell \leq |\mathcal{PP}|$. In $S_C^T(Place_\ell)$ all atomic actions $Act_i^{Place_y}$ are replaced with empty events ε, except those happening at $Place_\ell$, $y = \ell$:

$$T_p : [S_C, \mathcal{PP}, \varepsilon] \longrightarrow [S_C^T(Place_1), ..., S_C^T(Place_\ell), ..., S_C^T(Place_{|\mathcal{PP}|})] \quad (5)$$

Projections $S_C^T(Place_1), ..., S_C^T(Place_\ell), ..., S_C^T(Place_{|\mathcal{PP}|})$ are after that simplified by deletion of ε together with the accompanied operators and parentheses.

Both distribution transformations, defined in detail in [9], logically implement a given SLP specification and preserve the external service behaviour which is specified by the original S_C before its distribution.

4 Nondeterminism-reducing transformation

The nondeterminism-reducing transformation has been chosen to demonstrate how an SLP specification, if described in an abstract algebraic specification language, can be simplified, still specifying a service with unchanged external behaviour. Using the transformation, we remove from the SLP specification the unnecessary alternative subexpressions which are internally redundant, i.e. their external behaviour is due to the SSL semantics already properly specified by other service subexpressions [8]. The resulting SLP specification is more deterministic.

Internal redundancy can enter into an SLP specification in two ways:

- It may be already included in a particular SIB specification.
- It can be inserted into the SLP specification by composition of SIB specifications which not necessarily contain nondeterminism.

4.1 Redundancy identification and reduction

Let $A(B)$ represent an acceptance tree of a service expression (a SLP specification) B, $B = S_C$, and sB_j, $sB_j \subseteq B$, $1 \leq j \leq J$ a service subexpression. Let $\mathcal{A}_j$ denote a set of subtrees $sA_k(sB_j)$, $1 \leq k \leq K$ in the acceptance tree A. Then, we define the function $\Phi : sB_j \longrightarrow \mathcal{A}_j$, which maps the subexpression sB_j into the set $\mathcal{A}_j$ of subtrees in A, representing sB_j in A. We also define a set $maxp_j$ whose elements are sequences of atomic actions Act^{Place}, denoting edges in the paths from the roots of all $sA_k(sB_j)$ in $\mathcal{A}_j$ to their leaves. As red_type we select internal redundancy which has to be removed (re).

Let $sB_1, sB_2, ..., sB_m$, represent alternative subexpressions in B. The behaviour of an alternative subexpression sB_j is already properly covered by alternative subexpressions $sB_k, ..., sB_m$, $j \neq k, ..., m$, if $maxp_j$ equals to or is a subset of $\cup_r maxp_r$, $k \leq r \leq m$. We describe this property of sB_j with the relation "$\subseteq_s$": $sB_j \subseteq_s (sB_k, ..., sB_m) \iff maxp_j(\Phi(sB_j)) \subseteq maxp_k(\Phi(sB_k)) \cup ... \cup maxp_m(\Phi(sB_m))$. Due to their external irrelevance, internal labels of atomic actions are here disregarded. If $sB_j \subseteq_s (sB_k, ..., sB_m)$, then we define sB_j as an unnecessary, internally redundant subexpression of B.

Nondeterminism-reducing transformation proceeds in four steps:

1. construction of A from a given SLP specification,
2. calculation of values of Φ and $maxp_j$,
3. identification of unnecessary alternative subexpressions, and
4. simplification of the SLP specification.

Step 1. An acceptance tree A is recursively constructed from a given SLP specification S_C:

1a. Create the root r.

1b. Identify elements of the acceptance set $\mathcal{S}$ assigned to the root. For each action a in $\mathcal{S}$, generate a successor branch and add it to the successor set $D(r)$ if it is not already contained in $D(r)$.

1c. If there exist successors of r in $D(r)$, assign the successors to be the new roots r_j, $1 \leq j \leq |D(r)|$ and execute the substep 1a.

Elements of $\mathcal{S}$ are identified for each node of A using the following construction rules:

cr1.	$\mathcal{S}(\delta) = \{\emptyset\}$
cr2.	$\mathcal{S}(Act^{Place}) = \{\{Act^{Place}\}\}$
cr3.	$\mathcal{S}(;(B_1, B_2)) = \mathcal{S}(B_1)$
cr4.	$\mathcal{S}(\vert\vert\vert(B_1, ..., B_N)) = \{\mathcal{S}_i \mid \mathcal{S}_i = \cup \mathcal{S}_{i\ell}, \mathcal{S}_{i\ell} \subseteq \mathcal{S}(B_\ell), 1 \leq \ell \leq N\}$
cr5.	$\mathcal{S}([](B_1, ..., B_N)) = \mathcal{S}(\vert\vert\vert(B_1, ..., B_N))$

Rule *cr1* defines $\mathcal{S}(\delta)$ where δ represents an observable event of successful termination of a process execution. Rule *cr2* specifies an acceptance set for a service expression, containing a single atomic action Act^{Place}. By Rule *cr3*, an acceptance set for sequentially composed service expressions B_1 and B_2 is defined which equals to the acceptance set of B_1. The acceptance set for parallel composition of service expressions $B_1, ..., B_N$, defined by Rule *cr4*, equals to the union of acceptance sets for particular expressions. In the case of alternatively composed expressions $B_1, ..., B_N$ the same rule may be applied (*cr4* = *cr5*).

Step 2. Values of Φ and $maxp_j$ are calculated in two substeps:

2a. For each sB_j, $1 \leq j \leq J$ first find the roots r_k of the corresponding $sA_k(sB_j)$, $1 \leq k \leq K$. The roots r_k are identified regarding the contents of sB_j (the actions which may execute first) and the successor sets $D(r_k)$.

2b. Determine the subsets of $D(r_k)$, belonging to $sA_k(sB_j)$, and calculate $maxp_j$.

Step 3. Unnecessary alternative subexpressions are identified by comparison of $maxp_j(\Phi(sB_j))$:

3a. To perform the comparison in an appropriate order, first construct a syntax tree of the behaviour expression B.

3b. Find the leftmost alternative operator at the highest level of the syntax tree as possible. Assign identifiers $1 \leq m \leq M$ to the corresponding subexpressions sB_m. For each sB_j, $1 \leq j \leq m$ test the relation $\subseteq$ between $maxp_j$ and $\cup_r maxp_r$, $k \leq r \leq m$, $j \neq k, ..., m$. If an sB_j is unnecessary, assign to the atomic actions Act_{Place} involved the values of boolean variables $UN(Act^{Place}) = true$. Recursively execute this substep in the depth of B for all its alternative subexpressions, which contain at least one alternative operator and no actions with the value of $UN = true$.

3c. If there were in the first substep several alternative operators at the same level of the syntax tree of B, continue the comparison in the width of B.

3d. All atomic actions Act^{Place} with values of boolean variables $UN(Act^{Place}) = true$ finally subsequently replace with empty events ε.

Step 4. Finally, the service expression B is simplified. For all empty events ε in B:

4a. Delete an ε.

4b. Delete the accompanied operators and parentheses.

The simplification is carried out using the following simplification rules:

sr1. $op(a, \varepsilon) \longrightarrow a,\ op \in \{;, |||, []\}$
sr2. $op(\varepsilon, a) \longrightarrow a,\ op \in \{|||, []\}$
sr3. $op(a, \varepsilon, b, ...) \longrightarrow op(a, b, ...),\ op \in \{|||, []\}$
sr4. $op(\varepsilon, ..., \varepsilon) \longrightarrow \varepsilon,\ op \in \{|||, []\}$
sr5. $op(\varepsilon, \varepsilon) \longrightarrow \varepsilon,\ op \in \{;\}$

Rule *sr3* may be used for any execution ordering of events "a", "b", "ε" etc. The expression "$op(\varepsilon, ..., \varepsilon)$" in Rule *sr4* may contain any number of empty events ε.

Since $SK(A(S_C))$ is strongly bisimulationally equivalent to $SK(A)$, the transformation preserves the external behaviour, specified by the S_C. Describing the transformation, we omitted the algorithms for construction of the set $\mathcal{R}$ and updating of its contents to get $\mathcal{R}^T$. Observation of causal dependencies is in this particular case of nondeterminism-reducing transformation not necessarily needed. If the simplified SLP specification is used as input of another behaviour-preserving transformation where the contents of $\mathcal{R}$ has to be known, the $\mathcal{R}$-handling algorithms have to be used. We give a detailed description of these algorithms and other algorithms in the scope of the transformation in [9].

The worst-case time complexity of the nondeterminism-reducing transformation is $O(N!)$ where N represents a number of internally labelled atomic actions in the original SLP specification. As the worst case we consider an SLP specification with an alternative operator in the root of its syntax tree and parallel composition operators in the remaining nodes. However, in SLP specifications of real IN services such worst cases can hardly be found. That results in a significantly lower average time complexity and acceptable applicability of the transformation.

4.2 Example

Let an SLP specification S_C be composed of SIB specifications S_{Pi}, $1 \leq i \leq 4$, specified in SSL as $S_{P1} = S(SIB_1) = (a^1; b^2); c^3$, $S_{P2} = S(SIB_2) = d^4$, $S_{P3} = S(SIB_3) = b^2|||c^3$ and $S_{P4} = S(SIB_4) = b^1; e^5$. Let S_C have the following form (for the reason of more readable specifications we use in this example an infix notation of SSL expressions):

$$S_C = (((a^1; b^2); c^3); d^4) \; [] \; (a^1; ((b^2|||c^3)|||d^4)) \; [] \; ((b^1; e^5); d^4) \tag{6}$$

S_C is a behaviour expression of type $[](sB_1, sB_2, sB_3)$ where sB_j, $1 \leq j \leq 3$ represent alternative subexpressions. Performing the nondeterminism-reducing transformation, we first translate the expression S_C into the corresponding acceptance tree A. The tree is shown in Figure 22. Different occurrences of equal atomic actions are indicated in the figure by subscripts ("a_1^1" and "a_2^1", for example). After that we identify a set $\mathcal{A}_j$ of subtrees $sA_k(sB_j)$, $k = K = 1$, which are values of Φ for subexpressions sB_j, and maximal path sets $maxp_j$ for the sets of subtrees $\mathcal{A}_j$. $maxp_j$ is constructed by observing paths from the root of each $sA_k(sB_j)$ in $\mathcal{A}_j$ to its leaves.

In our example, each sB_j is assigned an $\mathcal{A}_j$ with $sA_1(sB_j)$, where the roots of all $sA_1(sB_j)$ equal to the root of A. The subtree $sA_1(sB_1)$ starting with the edge specified as "a_1^1"corresponds to sB_1, the subtree $sA_1(sB_2)$ starting with the edge "a_2^1" to sB_2, and the subtree $sA_1(sB_3)$ starting with the edge "b_1^1" to sB_3. Maximal path sets $maxp_j$ contain the following elements:

$$maxp_1(\Phi(sB_1)) \;\; = \;\; \{a_1^1 b_1^2 c_1^3 d_1^4\}$$

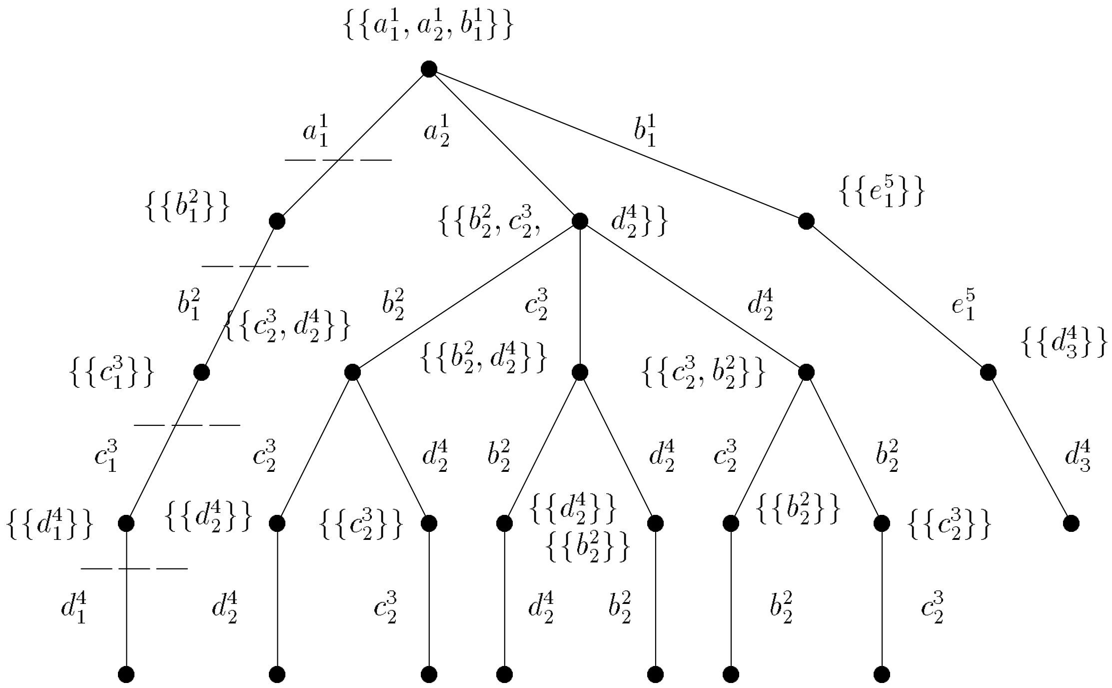


Figure 22: Identification of internal redundancy

$$
\begin{aligned}
maxp_2(\Phi(sB_2)) &= \{a_2^1 b_2^2 c_2^3 d_2^4,\ a_2^1 b_2^2 d_2^4 c_2^3,\ a_2^1 c_2^3 b_2^2 d_2^4, a_2^1 c_2^3 d_2^4 b_2^2,\ a_2^1 d_2^4 c_2^3 b_2^2,\ a_2^1 d_2^4 b_2^2 c_2^3\} \\
maxp_3(\Phi(sB_3)) &= \{b_1^1 e_1^5 d_3^4\}
\end{aligned}
$$

After comparison of the sets contents we may conclude that the set $maxp_1$ satisfies $maxp_1(\Phi(sB_1)) \subseteq maxp_2(\Phi(sB_2)) \cup maxp_3(\Phi(sB_3))$. Since $sB_1 \subseteq_s (sB_2, sB_3)$, the alternative subexpression sB_1 is internally redundant. All atomic actions of sB_1 may be assigned the value of the boolean variable $UN = true$ and S_C may be simplified by removing the complete sB_1. Edges corresponding to sB_1 are in Figure 22 crossed out. The internal redundancy of sB_1 has been caused by composition of SIB specifications.

Considering the contents of $maxp_j$, $1 \leq j \leq 3$, we may also conclude that S_C does not contain other unnecessary alternative subexpressions. The result is the more deterministic SLP specification

$$S_C^T = (a^1; ((b^2|||c^3)|||d^4)) \; [] \; ((b^1; e^5); d^4) \tag{7}$$

which is easier to implement than the original one (implementation of SIB_1 is not needed), still offering users a service with equivalent external behaviour.

If S_C^T is to be used as an input to another behaviour-preserving transformation where information on causal dependencies between atomic actions is required, we also construct the set $\mathcal{R}$ for S_C and update its contents regarding S_C^T. Here, we delete all causally related pairs of service primitives which are implied by the unnecessary alternative subexpression sB_1.

5 Conclusion

We suggested a formal transformational approach to design of IN (Intelligent Network) services. Developing the approach, we were motivated by design methods based on behaviour-preserving transformations for other types of architectures, particularly for hardware architectures. Its main advantages are the formal systematical generation of SIB (Service-Indepenendent Building Block) and SLP (Service Logic Program) specifications by composition transformations, the preservation of syntactical and semantical correctness at execution of redundancy-handling and distribution transformations, and the reduction of the nondeterminism degree by the nondeterminism-reducing transformation. The proposed approach, when implemented as a tool, can effectively support an IN service designer to develop correct SLP specifications and their efficient implementations. The defined transformations can also be adapted for the use on system behaviour descriptions with other formal description techniques, which may be assigned the same or similar types of semantics.

References

[1] A.J.W.M. Ten Berg and T. Krol. Formal transformational design of hardware architectures. In *Proc. of the 21st Euromicro Conference*, pages 110–117, Como, Italy, 1995. IEEE Computer Society Press.

[2] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution of the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995.

[3] K.E. Cheng and L.N. Jackson. An intelligent network service creation environment. In O. Færgemand and R. Reed, editors, *SDL'91: Evolving methods*, pages 207–220. Elsevier Science Publishers B.V. (North-Holland), 1991.

[4] O.B. Clarisse, E.A. Kidwell, W.F. Opdyke, R.E. Pitt, and B.A. Westergren. Service creation using application building blocks. In *Proc. of the 3rd Int. Conference on Intelligence in Networks*, pages 61–66, Bordeaux, France, 1994.

[5] D. Gajski, N.D. Dutt, A.C-H. Wu, and S.Y-L. Lin. *High-level Synthesis – Introduction to Chip and System Design.* Kluwer Academic Publishers, Norwell, Massachusetts, 1992.

[6] M. Hennessy. *Algebraic Theory of Processes.* The MIT Press, Cambridge, Massachusetts, 1988.

[7] ISO. *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, IS 8807.*

[8] M. Pučko. Automated development of Intelligent Network services. In *Proc. of the 21st Euromicro Conference*, pages 387–394, Como, Italy, 1995. IEEE Computer Society Press.

[9] M. Pučko. *Automatic service-driven protocol synthesis for specific users.* PhD thesis, University of Ljubljana, Slovenia, 1995.

[10] K. Rath, M. Esen Tuna, and S.D. Johnson. Behaviour tables: A basis for system representation and transformational system synthesis. In *Proc. of the 1993 IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, pages 736–740, Santa Clara, California, 1993. IEEE Computer Society Press.

[11] M. Schenke, H. Langmaack, W.-P. de Roever, and J. Vytopil. Specification and transformation of reactive systems with time restrictions and concurrency. In H. Langmaack and W.-P. de Roever, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 605–620, Berlin, Germany, 1994. Springer-Verlag.

[12] ITU: Telecommunication Standardization Sector. *Intelligent Network Recommendations, Q.12xx Series.*

[13] G. Winskel. An introduction to event structures. *Lecture Notes in Computer Science*, (354):364–397, 1989.

Primitive Subtyping $\wedge$ Implicit Polymorphism $\models$ Object-orientation

François Bourdoncle
Centre de Mathématiques Appliquées, Ecole des Mines de Paris
60, boulevard Saint-Michel, F-75014 Paris, France
bourdoncle@cma.ensmp.fr

Stephan Merz
Institut für Informatik, Technische Universität München
Arcisstraße 21, D-80290 München, Germany
merz@informatik.uni-muenchen.de

Abstract

We present a new predicative and decidable type system, called $\mathrm{ML}_{\leq}$, suitable for languages that integrate functional programming and parametric polymorphism in the tradition of ML [12, 16], and class-based object-oriented programming and higher-order multi-methods in the tradition of CLOS [7]. Instead of using extensible records as a foundation for object-oriented extensions of functional languages, we propose to reinterpret ML datatype declarations as abstract and concrete class declarations, and to replace pattern matching on run-time values by dynamic dispatch on run-time types. $\mathrm{ML}_{\leq}$ is based on universally quantified polymorphic constrained types. Constraints are conjunctions of inequalities between monotypes built from type constructors organized into extensible and partially ordered classes. We give type checking rules for a small, explicitly typed functional language à la XML [11] with multi-methods, show that the resulting system has decidable minimal types, and discuss subject reduction.

1 Overview

We present a new predicative and decidable type system called $\mathrm{ML}_{\leq}$ that adds primitive subtyping to the ML type system [16]. Our goal is to devise a type discipline for a strongly typed, higher-order object-oriented language, building on well-understood concepts from type systems for functional languages, rather than on special calculi [1] or resorting to "ad-hoc" polymorphism [19, 23], second-order systems [6, 21], or recursive types [4]. Instead, our system is based on the following ideas. We insist on a separation of interface and implementation. Interfaces define types and method signatures, where subtyping provides for a hierarchical, user-extensible modelling. Implementation modules may define data

```
interface Point is
 class Point[];
 type point : Point;
  meth dist : point → real;
  meth move : ∀α : α ≤ point. α → α;
  meth leq : (point, point) → bool
end Point;
 module Point is
  meth leq(p : point, q : point) =
   (dist p) ≤ (dist q)
end Point;
```

```
module CPoint is
 open Point;
data cpt[] : Point is
  x, y : real end;
 order cpt ⊏ point;
 meth dist(p : cpt) =
  sqrt((cpt.x p)^2 + (cpt.y p)^2);
 meth move(p : cpt) =
  cpt (1 + (cpt.x p))
      (1 + (cpt.y p));
end CPoint;
```

```
module PPoint is
 open Point;
data ppt[] : Point is
    d, a : real end;
 order ppt ⊏ point;
 meth dist(p : ppt) =
  (ppt.d p);

 meth move(p : ppt) =
  ppt (1 + (ppt.d p))
       (ppt.a p);
end PPoint;
```

Figure 23: Points in the plane

types (in the form of tagged records), which form the leaves of the type hierarchy, as well as method implementations for specific argument types. We do not model the inheritance of implementations (which we believe to be a purely syntactic notion) in the type system. We think of methods as sets of functions, similar in spirit to languages like CLOS [7] and Cecil [5], that dispatch on the run-time types of all their arguments.

We enrich the original Hindley-Milner type system [12, 16] with *polymorphic constrained types* of the form $\forall\vartheta : \kappa.\ \theta$ where ϑ is a set of variables, κ is a constraint, and θ is a (quantifier-free) monotype. Before we define these notions in detail in section 2, we would like to illustrate the use of our type system at the hand of the classical example of points in the two-dimensional plane, expressed in a fictitious language with full type annotations (see figure 23). We do not study type inference in this paper, although we will come back to this issue in the conclusion.

The interface *Point* declares a zero-ary *type constructor class* Point as well as a type constructor point. Type constructors are used to form complex type expressions; we identify a zero-ary type constructor such as point with the type $\mathsf{point}[\,]$. Type constructor classes partition the universe of type constructors into disjoint subuniverses. In particular, we consider two types incomparable if their outermost type constructors belong to different classes. In OO jargon, one would call the type point an "abstract class", because the interface does not define an implementation for points. The interface *Point* specifies three methods *dist*, *move*, and *leq* that operate on objects of type point. Method *dist* has type $\mathsf{point} \to \mathsf{real}$, that is, it expects an argument of type point and returns a real number[11]. As we shall see, this type is equivalent to the polymorphic constrained

[11] In this example, we assume predefined types real and bool.

type $\forall\alpha\colon \alpha \le \mathsf{point}.\ \alpha \to \mathsf{real}$, that is, method *dist* may be applied to any subtype of type point, as it should be when object hierarchies are extensible.

Method *move* similarly expects some type below point, but returns a result of the argument type. In some object-oriented languages, a similar type can be specified using type specifiers such as like self. Note that recursive types are necessary to give a satisfactory typing for the method *move* in the "methods-as-records" paradigm. We shall see in section 2 that the type of *move* is a strict subtype of the type $\mathsf{point} \to \mathsf{point}$ in the system $\mathrm{ML}_{\le}$. Moreover, we can also give precise typings for methods that expect several arguments, because our methods dispatch on all their arguments. Finally, the method *leq* compares two point objects—even if they belong to different subtypes of type point. In fact, the type given to *leq* is equivalent to

$$\forall\alpha,\beta\colon \alpha \le \mathsf{point} \wedge \beta \le \mathsf{point}.\ (\alpha,\beta) \to \mathsf{bool}$$

Along with the interface, we also define an implementation module *Point* that provides a default implementation of the method *leq* based on the method *dist* whose existence is specified in the interface.

Modules *CPoint* and *PPoint* provide two different implementations of the interface *Point*, one using Cartesian and the other polar coordinates. The implementations first define zero-ary data type constructors cpt and ppt below type constructor point. Data types not only define tagged records, which can be instantiated at run-time (as in ML), but also type constructors (as opposed to ML). The only relationship that exists between the data types cpt and ppt is the fact that they share a common supertype. In fact, we require data type constructors to be minimal elements in the hierarchy of type constructors. For notational conciseness, we assume that the definition of a data type implicitly defines corresponding constructor and accessor functions for the record components.

The modules *CPoint* and *PPoint* go on to provide implementations of the methods *dist* and *move* specified in the interface. Notice that we need not give new implementations for method *leq*, because its implementation in module *Point* is effectively defined on the entire domain of *leq*. Dynamic binding of methods ensures that the correct implementations of *dist* are used whenever *leq* is evaluated.

The requirement that data type constructors are minimal ensures that methods like *move* are well-typed. For assume we were allowed to define a subconstructor $\mathsf{scpt} \sqsubset \mathsf{cpt}$ of cpt in some further module. Because we insist that subtypes should be acceptable whenever their supertypes are, we should be able to pass an object of type scpt to the function *move* defined in module *CPoint*—which would then return an object of type cpt, violating the type specification of method *move* in the interface *Point*.

Finally, we would like to emphasize that methods are first-order in our system. Specifically, they can be passed as function arguments. For instance, suppose that we have a polymorphic sorting method *sort* of type

$$sort : \forall\alpha.\ \mathsf{list}[\alpha] \to ((\alpha,\alpha) \to \mathsf{bool}) \to \mathsf{list}[\alpha]$$

that expects a list and a comparison function and returns a list of the same element type. It is then perfectly acceptable to apply *sort* to a list of points. The capability of defining polymorphic higher-order functions has proven to be a powerful tool in functional programming; we inherit it for free from the ML type system, which is a special case of our type system. In contrast to ML, we allow methods to have different implementations for different argument types.

A complete exposition, including all proofs that had to be omitted due to space restrictions, can be found in [3].

2 Constrained types

A *type constructor class* $C = (N, T, D, \sqsubseteq, \partial)$ is given by a name N, a finite and non-empty set T of *type constructors*, a set $D \subseteq T$ of *data type constructors*, a partial order $\sqsubseteq$ on T such that the data type constructors are minimal with respect to $\sqsubseteq$, and a *variance* ∂, given as a tuple of elements of the set $\{\oplus, \ominus, \otimes\}$ that denote, respectively, positive variance, negative variance, and non-variance. The arity of a class is the length of its variance. A *type structure* is a finite set $\mathcal{T}$ of type constructor classes with distinct names and pairwise disjoint sets of type constructors. We assume that every type structure contains a ()-variant class Unit with at least one data type constructor unit, and a $(\ominus, \oplus)$-variant class Arrow with at least one data type constructor $\to$. The set $\mathcal{G}_{\mathcal{T}}$ of *ground monotypes over* $\mathcal{T}$ is the least set such that when t_C is a type constructor of a class C in $\mathcal{T}$ with arity n and $\theta_1, \ldots, \theta_n$ are ground monotypes over $\mathcal{T}$, then $t_C[\theta_1, \ldots, \theta_n]$ is a ground monotype over $\mathcal{T}$. The *standard ordering* $\leq$ is the smallest relation on $\mathcal{G}_{\mathcal{T}}$ such that $t_C \sqsubseteq_C t'_C$ and $\Theta_C \leq_C \Theta'_C$ imply $t_C[\Theta_C] \leq t'_C[\Theta'_C]$, where the relation $\leq_C$ on monotype lists is defined as the componentwise ordering induced by the variance of C. For example, if C has the variance $(\oplus, \ominus, \otimes)$, then $(\theta_1, \theta_2, \theta_3) \leq_C (\theta'_1, \theta'_2, \theta'_3)$ holds if and only if $\theta_1 \leq \theta'_1$, $\theta'_2 \leq \theta_2$, and $\theta_3 = \theta'_3$ all hold, where we take the latter identity to mean that both $\theta_3 \leq \theta'_3$ and $\theta'_3 \leq \theta_3$ hold. The definition of the standard ordering ensures that $\leq$ is strictly structural: in particular, the outermost type constructors of the monotypes θ and θ' are of the same class whenever $\theta \leq \theta'$ holds.

In programming terms, a type structure is determined by class, type, and ordering declarations in modules. Our goal is to model object-orientation; we must therefore provide for the extension of type structures. Adding new type constructor classes is never a problem (assuming that there are no name clashes), because type constructors of different classes are completely unrelated to each other. When new type constructors are added to existing classes, one has to

$$
\begin{array}{lll}
\textit{C-constructors} & \phi_C & ::= t_C \mid v_C \\
\textit{Monotypes} & \theta & ::= \phi_C[\Theta_C] \mid v \\
\textit{Constraints} & \kappa & ::= \phi_C \sqsubseteq \phi_C \mid \theta \leq \theta \mid \kappa \wedge \kappa
\end{array}
$$

Figure 24: Monotypes and constraints

be careful to preserve the ordering on existing type constructors. We thus define *admissible extensions* of type structures much as conservative extensions are defined for models of algebraic specifications [3].

Ground monotypes are not enough to model polymorphism. We therefore assume given countable and pairwise disjoint sets of *type variables* $v, v', \ldots$, and, for each class C, *type constructor variables* $v_C, v'_C, \ldots$. In informal exposition, we do not distinguish between type and type constructor variables, but assume a universal set of variables $\alpha, \beta, \gamma, \ldots$. We define arbitrary (not necessarily ground) *monotypes* to be either a single type variable v, or recursively be built from type constructors, type constructor variables, and monotypes. The formation rules are formally given in figure 24, where we let Θ_C denote a list of monotypes whose length agrees with the arity of class C.

Figure 24 defines a *constraint* κ to be a conjunction of inequations $\phi_C \sqsubseteq \phi'_C$ between type constructors (or type constructor variables) and $\theta \leq \theta'$ between monotypes. We identify a constraint with the set of its conjuncts and use *true* to denote the trivial constraint $\mathsf{unit} \sqsubseteq \mathsf{unit}$.

A *type* τ is of the form $\forall\vartheta\colon \kappa.\ \theta$ where ϑ is a variable set, κ is a constraint, and θ is a monotype. Constrained types provide a flexible and expressive notation for types. For example, a type like $\forall\alpha\colon \mathsf{int} \leq \alpha \wedge \alpha \leq \mathsf{real}.\ (\alpha, \alpha) \to \alpha$ may be a very precise typing of the subtraction method; it subsumes both types $(\mathsf{int}, \mathsf{int}) \to \mathsf{int}$ and $(\mathsf{real}, \mathsf{real}) \to \mathsf{real}$, neither of which is a subtype of the other. However, many syntactically different constrained types can have the same meaning. For example, we would expect to identify the types $\forall\alpha\colon \mathsf{cpt} \leq \alpha \wedge \alpha \leq \mathsf{cpt}.\ \alpha$ and $\forall\emptyset\colon \mathit{true}.\ \mathsf{cpt}$ (which we also write as $\forall\,\emptyset.\ \mathsf{cpt}$). It is therefore important to formally identify equivalent types in a type system. We propose to do this by defining a subtyping relation between types. In most predicative type systems, subtyping is implicitly defined in terms of instantiations: a polytype is "below" all its ground instances. In particular, two ML types are equivalent if and only if they are equal up to renaming of bound variables.

For constrained types, we want a similar subtyping relation, which should moreover be compatible with the standard ordering on ground monotypes. For example, since $\tau \equiv \forall\alpha\colon \mathsf{cpt} \leq \alpha \wedge \alpha \leq \mathsf{cpt}.\ \alpha$ and $\forall\,\emptyset.\ \mathsf{cpt}$ should be equivalent, and $\forall\,\emptyset.\ \mathsf{cpt}$ should obviously be a subtype of $\forall\,\emptyset.\ \mathsf{point}$, τ itself should be a subtype of $\forall\,\emptyset.\ \mathsf{point}$, although point does not satisfy τ's constraint. We find it

$[Approx]$	$\forall\vartheta.\ \kappa\{\kappa'\} \models \kappa'$	
$[CRef]$	$\forall\vartheta.\ \kappa \models \kappa \wedge \phi_C \sqsubseteq \phi_C$	
$[CTrans]$	$\forall\vartheta.\ \kappa\{\phi_C \sqsubseteq \phi'_C \sqsubseteq \phi''_C\} \models \kappa \wedge \phi_C \sqsubseteq \phi''_C$	
$[CTriv]$	$\forall\vartheta.\ \kappa \models \kappa \wedge t_C \sqsubseteq t'_C$	$(t_C \sqsubseteq_C t'_C)$
$[CMin]$	$\forall\vartheta.\ \kappa\{\phi_C \sqsubseteq d_C\} \models \kappa \wedge d_C \sqsubseteq \phi_C$	
$[VIntro]$	$\forall\vartheta.\ \kappa[\sigma] \models \kappa$	$(\sigma \in \mathcal{S}(\vartheta))$
$[VElim]$	$\forall\vartheta.\ \kappa\{v \simeq \phi_C[\Theta_C]\} \models \kappa \wedge v = v'_C[\vartheta'_C]$	$(v'_C, \vartheta'_C$ fresh$)$
$[MRef]$	$\forall\vartheta.\ \kappa \models \kappa \wedge \theta \leq \theta$	
$[MTrans]$	$\forall\vartheta.\ \kappa\{\theta \leq \theta' \leq \theta''\} \models \kappa \wedge \theta \leq \theta''$	
$[MIntro]$	$\forall\vartheta.\ \kappa\{\Theta_C \leq_C \Theta'_C \wedge \phi_C \sqsubseteq \phi'_C\} \models \kappa \wedge \phi_C[\Theta_C] \leq \phi'_C[\Theta'_C]$	
$[MElim]$	$\forall\vartheta.\ \kappa\{\phi_C[\Theta_C] \leq \phi'_C[\Theta'_C]\} \models \kappa \wedge \phi_C \sqsubseteq \phi'_C \wedge \Theta_C \leq_C \Theta'_C$	

Figure 25: Constraint implication

helpful to intuitively think of a type $\forall\vartheta\colon \kappa.\ \theta$ as the *upper ideal closure* (w.r.t. the standard ordering on ground monotypes) of the set of ground instances of θ that satisfy the constraint κ. Hence, the type τ defined above would denote the set $\{\mathsf{cpt}, \mathsf{point}\}$ in the type structure defined by the program of figure 23, and so would the type $\forall\,\emptyset.\ \mathsf{cpt}$. Taking the upper closure of the ground solutions ensures that subtypes may be substituted for their supertypes. For example, we want method *leq* to be applicable to objects of type $\forall\,\emptyset.\ \mathsf{cpt}$.

Unfortunately, in general this intuition is not quite correct, because it does not account for possible extensions of the current type structure in object-oriented programming. For example, consider the types $\forall\,\emptyset.\ \mathsf{point}$ and $\forall\alpha\colon \alpha \leq \mathsf{point}.\ \alpha$ in the type structure defined by the interface *Point* of figure 23 alone. Although both types have the same solutions, namely just point, in this type structure, they should not be considered equivalent. In general, the denotation of the latter type will be larger than the denotation of the former one.

Formally, assuming that ϑ_1 and ϑ_2 are disjoint, we say that $\tau_1 \equiv \forall\vartheta_1\colon \kappa_1.\ \theta_1$ is a subtype of $\tau_2 \equiv \forall\vartheta_2\colon \kappa_2.\ \theta_2$ if for every admissible extension $\mathcal{T}^*$ of the given type structure $\mathcal{T}$ and every ground substitution σ_2 such that $\kappa_2[\sigma_2]$ holds in $\mathcal{T}^*$, there exists a ground substitution σ_1 that agrees with σ_2 on the variables that are free in τ_2 such that $(\kappa_1 \wedge \theta_1 \leq \theta_2)[\sigma_1]$ holds in $\mathcal{T}^*$. The requirement that this condition holds for any admissible extension ensures that well-typed programs remain well-typed when the type structure is extended.

In order to study the ordering on types, we define a more general notion of *constraint implication*. We will show below how the ordering on types may be defined in terms of constraint implication. We say that a constraint κ implies a constraint κ' for all ϑ, written $\forall\vartheta.\ \kappa \models \kappa'$ if for every admissible extension $\mathcal{T}^*$ of type structure $\mathcal{T}$ and every solution σ_1 of κ in $\mathcal{T}^*$ there exists a solution σ_2 of κ' in $\mathcal{T}^*$ that agrees with σ_1 on the variables in ϑ. We propose to axiomatize the implication of constraints by the rules of figure 25 plus the transitivity rule

$$\frac{\forall\vartheta.\ \kappa_1 \models \kappa_2 \qquad \forall\vartheta.\ \kappa_2 \models \kappa_3}{\forall\vartheta.\ \kappa_1 \models \kappa_3}\,[\mathit{Trans}]$$

In these rules, d_C denotes a data type constructor of class C and we let $\theta_1 \simeq \theta_2$ denote either of $\theta_1 \leq \theta_2$ or $\theta_2 \leq \theta_1$. $\mathcal{S}(\vartheta)$ denotes the set of *ϑ-substitutions*, which map type variables to monotypes and C-constructor variables to type constructors in T_C, and which are the identity on variables in ϑ. Finally, we write $\kappa\{\kappa'\}$ to denote that κ' is a subconstraint of κ, and $\theta = \theta'$ for the constraint $\theta \leq \theta' \wedge \theta' \leq \theta$.

In particular, we say that a constraint κ is *well-formed* if $\forall\emptyset.\ \mathit{true} \models \kappa$ holds, and that two constraints κ_1 and κ_2 are *ϑ-equivalent* if they imply each other for all ϑ.

The following theorem shows that the axiomatization of figure 25 captures our intentions:

Theorem 1 *Let ϑ be a variable set, and κ_1 and κ_2 be two well-formed constraints. Then $\forall\vartheta.\ \kappa_1 \models \kappa_2$ is derivable if and only if for every admissible extension $\mathcal{T}^*$ of $\mathcal{T}$ and every $\mathcal{T}^*$-ground substitution σ_1 such that $\kappa_1[\sigma_1]$ is satisfied in $\mathcal{T}^*$, there exists a $\mathcal{T}^*$-ground substitution σ_2 that agrees with σ_1 on the variables of ϑ such that $\kappa_2[\sigma_2]$ is a ground constraint satisfied in $\mathcal{T}^*$.*

For computational purposes, it is helpful to observe that rules *VElim* and *MElim* can be used to reduce any well-formed constraint to a constraint $\kappa \equiv \kappa_S \wedge \kappa_B$ in "normal form" where the "substitution part" κ_S contains only subconstraints of the form $v = \theta$ for type variables v in ϑ and monotypes θ (for a given variable set ϑ), and the "base part" κ_B contains only subconstraints of the form $v \leq v'$ or $\phi_C \sqsubseteq \phi'_C$ that do not contain complex monotypes. We show in [3] how this normal form helps to prove that constraint implication is decidable.

Theorem 2 *If κ_1 is a well-formed constraint and κ_2 is an arbitrary constraint, then it is decidable whether $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds. In particular, well-formedness of constraints is decidable.*

It follows from the results of [22] that deciding well-formedness of constraints (and therefore also constraint implication) is PSPACE-hard. In particular, conversion to the normal form described above may in general entail an exponential growth in the size of the constraint. Nevertheless, in [3] we give an algorithm (inspired by an incomplete algorithm given by Fuh and Mishra in [10]) that we expect to be polynomial for "real-world" programs.

We can now define the ordering on types in terms of constraint implication. In practice, we work relative to a *constraint context* $\Delta = (\vartheta : \kappa)$. The use of the constraint context will become clear when we explain type checking in section 3. It is used to store type information for symbols defined in the context of the current declaration; intuitively, it asserts the existence of variables ϑ that satisfy constraint κ. We say that a type $\tau_1 \equiv \forall \vartheta_1 \colon \kappa_1.\ \theta_1$ is *well-formed* w.r.t. Δ if ϑ and ϑ_1 are disjoint, all variables that appear in κ_1 or θ_1 are from $\vartheta \cup \vartheta_1$, and $\forall \vartheta.\ \kappa \models \kappa_1$ holds. For two types $\tau_1 \equiv \forall \vartheta_1 \colon \kappa_1.\ \theta_1$ and $\tau_2 \equiv \forall \vartheta_2 \colon \kappa_2.\ \theta_2$ that are well-formed w.r.t. Δ, we say that τ_1 is a subtype of τ_2 w.r.t. Δ if

$$\forall \vartheta, \vartheta_2.\ \kappa \wedge \kappa_2 \models \kappa_1 \wedge \theta_1 \leq \theta_2$$

holds, assuming that ϑ_1 and ϑ_2 are disjoint. This definition of type ordering generalizes the instance relation between typing statements defined by Mitchell [17], which requires equality on the right-hand side.

As an example, let us prove that the type $\forall \alpha \colon \alpha \leq \mathsf{point}.\ \alpha \to \alpha$ of the method *move* of figure 23 is a subtype of $\forall \emptyset.\ \mathsf{point} \to \mathsf{point}$ in the empty context. We have to prove

$$\forall \emptyset.\ \mathit{true} \models \alpha \leq \mathsf{point} \wedge \alpha \to \alpha \leq \mathsf{point} \to \mathsf{point}$$

which follows by rule *VIntro* from

$$\begin{aligned} \forall \emptyset.\ \mathit{true} \models {} & \mathsf{point} \leq \mathsf{point} \\ & \wedge\ \mathsf{point} \to \mathsf{point} \leq \mathsf{point} \to \mathsf{point} \end{aligned}$$

On the other hand, $\forall \emptyset.\ \mathsf{point} \to \mathsf{point}$ is not a subtype of $\forall \alpha \colon \alpha \leq \mathsf{point}.\ \alpha \to \alpha$, so the latter type is a strict subtype of the former one. For otherwise, we would have to show

$$\forall \alpha.\ \alpha \leq \mathsf{point} \models \mathsf{point} \to \mathsf{point} \leq \alpha \to \alpha$$

which (by rule *MElim*) amounts to proving

$$\forall \alpha.\ \alpha \leq \mathsf{point} \models \alpha = \mathsf{point}$$

which is obviously not derivable.

In the report [3] we prove that the definition of type ordering ensures that, as in ML, any well-formed type is a subtype of all its ground instances, and that the type ordering is compatible with the standard order on ground monotypes. Moreover, we obtain a lattice-like structure on types, as shown by the following theorem. This result is important to ensure minimal typing of, for instance, conditional expressions.

Theorem 3 *Assume that $\tau_1 \equiv \forall \vartheta_1 \colon \kappa_1.\ \theta_1$ and $\tau_2 \equiv \forall \vartheta_2 \colon \kappa_2.\ \theta_2$ are well-formed types that have a common supertype. Then τ_1 and τ_2 have a least upper bound $(\tau_1 \vee \tau_2)$ which is equivalent to*

$$\forall v, \vartheta_1, \vartheta_2 \colon (\kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq v \wedge \theta_2 \leq v).\ v$$

if ϑ_1 and ϑ_2 are disjoint and v is a fresh type variable.

Of particular interest are functional types of the form $\forall\vartheta\colon \kappa.\ \theta \to \theta'$. We show in [3] that it is possible to identify functional types with monotonic type transformers (w.r.t. the subtype relation) that provide an abstract representation of the input-output relation of a function. We also show that one can define the domain $dom_\Delta(\tau)$ of a functional type, written as $\exists\vartheta\colon \kappa.\ \theta$, with a theory dual to that of types. (We use existential quantification to emphasize the duality of types and domains, our notation should not be confused with existential types as proposed for example in [18].) We say that a type $\tau \equiv \forall\vartheta_1\colon \kappa_1.\ \theta_1$ belongs to a domain $\delta \equiv \exists\vartheta_2\colon \kappa_2.\ \theta_2$ in context $\vartheta\colon \kappa$ if for every admissible extension there exist instances of θ_1 and θ_2 that satisfy κ_1 and κ_2, respectively, such that θ_1 is below θ_2. Again, we use constraint implication for the formal definition of domain membership, and say that type τ belongs to domain δ in context $\vartheta\colon \kappa$ if

$$\forall\vartheta.\ \kappa \models \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq \theta_2$$

holds, assuming that ϑ_1 and ϑ_2 are disjoint. This definition ensures that every subtype of τ belongs to δ if τ does, and that τ belongs to every superdomain of δ. Hence, subtypes may be substituted for supertypes as function arguments, as required by the notion of substitutivity in object-oriented languages.

A functional type $\tau \equiv \forall\vartheta\colon \kappa.\ \theta \to \theta'$ can be interpreted as a type transformer if we define the application $app_\Delta(\tau, \tau_1)$ for type $\tau_1 \equiv \forall\vartheta_1\colon \kappa_1.\ \theta_1$ as

$$app_\Delta(\tau, \tau_1) = \forall\vartheta, \vartheta_1\colon \kappa \wedge \kappa_1 \wedge \theta_1 \leq \theta.\ \theta'$$

again assuming that ϑ and ϑ_1 are disjoint. For example,

$$\begin{aligned} & app_\Delta(\forall\alpha\colon \alpha \leq \mathsf{point}.\ \alpha \to \alpha, \forall\,\emptyset.\ \mathsf{cpt}) \\ = \ & \forall\alpha\colon \alpha \leq \mathsf{point} \wedge \mathsf{cpt} \leq \alpha.\ \alpha \end{aligned}$$

which is equivalent to $\forall\,\emptyset.\ \mathsf{cpt}$ w.r.t. the type ordering. The definition of the operator app_Δ ensures that it is covariant in both arguments, which is at the heart of a proof of the subject reduction theorem for the language defined in the following section.

3 Type-checking

We consider an explicitly typed functional language based on the type system introduced in section 2 and give rules to type-check expressions of this language. A program (interface or module) consists of a set of declarations of type constructor classes and types, together with the implementation of every declared data type. These declarations serve to define the type structure $\mathcal{T}$; they may be mutually recursive. We will not consider them in detail, but see figure 23 for an example. The second part of a program is an expression e which must type

$$\Delta; \Gamma\{x\colon \tau\} \vdash x\colon \tau \quad [\mathit{Var}]$$

$$\Delta; \Gamma \vdash \rho(d_C; \vartheta_C; x_1, \ldots, x_n)\colon (\forall\, \emptyset.\ d_C[\vartheta_C]) \quad [\mathit{Rec}]$$

$$\Delta; \Gamma \vdash d_C.i\colon (\forall \vartheta_C.\ d_C[\vartheta_C] \to d_C^i\langle\vartheta_C\rangle) \quad [\mathit{Prj}]$$

$$\frac{\Delta; \Gamma \vdash e\colon \tau \qquad \Delta \vdash \tau \leq \tau'}{\Delta; \Gamma \vdash e\colon \tau'}\,[\mathit{Sub}]$$

$$\frac{\Delta; \Gamma \vdash e_1\colon \tau_1 \qquad \Delta; \Gamma[x_1\colon \tau_1] \vdash e_0\colon \tau_0}{\Delta; \Gamma \vdash (\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_0\ \mathsf{end})\colon \tau_0}\,[\mathit{Let}]$$

$$\frac{\Delta; \Gamma[x_1\colon \tau_1, \ldots, x_n\colon \tau_n] \vdash e_i\colon \tau_i \qquad (0 \leq i \leq n)}{\Delta; \Gamma \vdash (\mathsf{letrec}\ x_1\colon \tau_1 = e_1; \ldots; x_n\colon \tau_n = e_n\ \mathsf{in}\ e_0\ \mathsf{end})\colon \tau_0}\,[\mathit{LetRec}]$$

$$\frac{\Delta; \Gamma \vdash e\colon \tau \qquad \Delta; \Gamma \vdash e'\colon \tau' \qquad \Delta \vdash \tau' \in \mathit{dom}_\Delta(\tau)}{\Delta; \Gamma \vdash (e\ e')\colon \mathit{app}_\Delta(\tau, \tau')}\,[\mathit{App}]$$

$$\frac{\Delta[\upsilon, \vartheta\colon \kappa \wedge \upsilon \leq \theta]; \Gamma[x\colon \forall\, \emptyset.\ \upsilon] \vdash e\colon (\forall \vartheta'\colon \kappa'.\ \theta') \quad (\upsilon\ \mathit{fresh})}{\Delta; \Gamma \vdash (\mathsf{fun}\ \{\vartheta \mid \kappa\}\, (x\colon \theta) \Rightarrow e)\colon (\forall \upsilon, \vartheta, \vartheta'\colon \kappa \wedge \kappa' \wedge \upsilon \leq \theta.\ \upsilon \to \theta')}\,[\mathit{Fun}]$$

$$\frac{\begin{array}{c}\delta = \exists \vartheta\colon \kappa.\ \theta \qquad \pi_i = \exists \vartheta_i.\ \theta_i \qquad \pi_1, \ldots, \pi_n \text{ is a partition of } \delta \text{ w.r.t. } \Delta \\ \Delta[\upsilon, \vartheta, \vartheta_i\colon \kappa \wedge \upsilon \leq \theta, \theta_i]; \Gamma[x\colon \forall\, \emptyset.\ \upsilon] \vdash e_i\colon (\forall\, \emptyset.\ \theta') \quad (1 \leq i \leq n,\ \upsilon\ \mathit{fresh})\end{array}}{\Delta; \Gamma \vdash (\mathsf{meth}\ \{\vartheta \mid \kappa\}\, (x\colon \theta)\colon \theta' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n])\colon (\forall \vartheta\colon \kappa.\ \theta \to \theta')}\,[\mathit{Meth}]$$

Figure 26: Typing rules

check w.r.t. $\mathcal{T}$. Figure 26 introduces the constructs of the language together with the corresponding typing rules. A typing context is a pair $(\Delta; \Gamma)$ where Δ is a constraint context and Γ is a list of bindings of the form $x : \tau$ for expression variables. An expression of the form $\rho(d_C; \vartheta_C; x_1, \ldots, x_n)$ is a record expression; it can (essentially) only appear in the implementation of a data constructor

$$\mathsf{data}\ d_C[\vartheta_C]\ \mathsf{is}\ 1\colon d_C^1\langle\vartheta_C\rangle; \ldots;\ n\colon d_C^n\langle\vartheta_C\rangle\ \mathsf{end}$$

As in section 1, we assume that the declaration of any data constructor also defines corresponding extractors $d_C.i$.

The subsumption rule *Sub* and the application rule *App* are reminiscent of rules in $F_{\leq}$; standard ML-like type systems leave subtyping implicit in the non-deterministic search for instantiations and do not define the notion of application of a (functional) type to another one. The difference is that type application is approximated in $\mathrm{ML}_{\leq}$ but exact in $\mathrm{F}_{\leq}$; exactness may be lost if type variables are shared between record fields in data type implementations [3].

The rules *Let* and *LetRec* are standard. In the rule *Fun* for function declarations, the function body e is type-checked in a context where the parameter x is known to have type $\forall\emptyset.\ \theta$ for some instantiation of ϑ such that κ holds. Note that only the domain, but not the result type of a function needs to be given explicitly. In fact, this is the reason for keeping function declarations in the language, which could otherwise be subsumed by the following rule for method declarations.

A method m is an expression of the form $\mathsf{meth}\ \{\vartheta \mid \kappa\}\,(x{:}\ \theta){:}\ \theta' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n]$, where each *pattern* π_i is a special kind of domain, either $\exists v.\ v$ or $\exists\vartheta_C.\ d_C[\vartheta_C]$ for a type constructor d_C. We think of a method as a set of functions, one for each pattern π_i, whose type is a subtype of the method type, restricted to the domain π_i. In contrast to ML patterns, which may be complex, the present definition of $\mathrm{ML}_{\leq}$ patterns allows for dynamic dispatch according to the outermost type constructor only. Rule *Meth* defines the type of method m as the explicitly given type $\forall\vartheta{:}\ \kappa.\ \theta \rightarrow \theta'$, provided that two conditions are met. First, the set of patterns must be a *partition* of the method's domain δ. This condition ensures the absence of "method not understood" errors at run-time, as well as the existence of a most specific pattern for every type in the domain of the method (see [3] for a precise definition of partition). In the presence of a module system as in figure 23, this condition must be checked at link time. Second, the body e_i of each alternative must have type $\forall\emptyset.\ \theta'$ in the context where x is known to have both type θ and type θ_i.

The following theorem shows that $\mathrm{ML}_{\leq}$ has decidable minimal typing.

Theorem 4 *Let $(\Delta;\Gamma)$ be a well-formed typing context and e be a well-formed expression w.r.t. $(\Delta;\Gamma)$. It is decidable whether e is well-typed in the context $(\Delta;\Gamma)$. If e is well-typed, it has a minimal type and this minimal type can be effectively determined.*

4 Conclusion

Our interest in this paper has been to enhance the standard Hindley-Milner type system for an explicitly typed version of ML so that it can be used as a basis for a higher-order object-oriented language. We believe that $\mathrm{ML}_{\leq}$ is a practical and natural extension of the Hindley-Milner type system, and that constraint implication is a unifying concept for such extensions. We have also defined a small programming language and given typing rules in the system $\mathrm{ML}_{\leq}$. In the report [3], we define a strict operational semantics for this language and prove a subject reduction theorem. This semantics performs dynamic dispatch on run-time types. We have not addressed type inference in this paper. Nevertheless, we conjecture that standard type inference techniques such as described in [2, 8, 10, 9, 13, 17, 20] can be used to infer the types of untyped $ML_{\leq}$ programs except for method declarations. We believe that inferring the type of methods will be much more challenging.

The report [3] discusses several extensions to the base language presented here. In particular, we propose a module system, show that multi-methods are more expressive than the single-argument methods we have presented in this paper, and discusses type classes. Although some of the uses of type classes can be achieved in the $ML_{\leq}$ type system, type classes are still desirable since our notion of subtyping is strictly structural: type constructors of different classes are incomparable. We also discuss some syntactic sugar, for instance, to obtain a form of implementation inheritance.

$ML_{\leq}$ has some similarity with $F_{\leq}$. However, $F_{\leq}$ is impredicative and relies on explicit polymorphism, that is, types are passed as arguments to functions. In contrast, $ML_{\leq}$ is a decidable and predicative system for implicit polymorphism where run-time types are used for dynamic binding[12].

$ML_{\leq}$ has strong connections with many other systems derived from the Hindley-Milner type system, in particular, systems of overloaded functions [14, 15, 23]. As indicated above, we believe that type classes are orthogonal to our system. They cannot be used to obtain typing of mixed operations like *leq*, unless multi-parameter type classes are used, but then type-checking becomes undecidable in general. Kaes' type system [15] is probably closest in spirit to $ML_{\leq}$; it offers even more expressive types, and provides type inference. On the other hand, Kaes does not address methods and dynamic dispatch, nor does he provide an operational semantics. The report [3] gives detailed comparisons with these and other type systems, and indicates connections that deserve further study.

Finally, we want to mention that a type checker for $ML_{\leq}$ has been implemented in Objective Caml. This prototype type checks approximately 400 lines per second on a low-cost workstation.

References

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In Donald Sannella, editor, *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 1–25, Berlin, 1994. Springer-Verlag.

[12] The operational semantics of $ML_{\leq}$ defined in [3] does not keep complete type information, but only tags run-time values with the outermost type constructor of their dynamic type, in order to perform dynamic dispatch.

[2] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming and Computer Architecture*, pages 31–41. ACM Press, 1993.

[3] François Bourdoncle and Stephan Merz. On the integration of functional programming, class-based object-oriented programming, and multi-methods. Research Report 26, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Paris, March 1996.

[4] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994.

[5] C. Chambers and Gary Leavens. Typechecking and modules for multi-methods. Technical Report UW-CS TR 95-08-05, University of Washington, 1995.

[6] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and the type checking of $F_{<}$. *Mathematical Structures in Computer Science*, 2(1), 1992.

[7] Linda G. DeMichiel and Richard P. Gabriel. Common LISP object system overview. In Jean Bézivin, editor, *European Conference on Object-Oriented Programming*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170, Berlin, 1987. Springer-Verlag.

[8] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 30 of *ACM / SIGPLAN notices*, pages 169–184, New York, 1995. ACM Press.

[9] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 2*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, March 1989. Springer-Verlag.

[10] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, June 1990. Special Issue: Second European Symposium on Programming (Nancy, France, March 1988).

[11] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 1993.

[12] James R. Hindely. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[13] Lalita Jategaonkar and John C. Mitchell. Type inference with extended pattern matching and subtypes. *Fundamenta Informaticae*, 19:127–166, 1993.

[14] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *ACM Conference on Functional Programming and Computer Architecture*. ACM Press, 1993.

[15] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *1992 ACM Conferenc on Lisp and Functional Programming*, pages 193–204. ACM Press, August 1992.

[16] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[17] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[18] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[19] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Functional Programming and Computer Architecture*, pages 135–146, San Diego, California, June 1995. ACM Press.

[20] Jens Palsberg. Efficient inference of object types. In *Logic in Computer Science*, pages 186–195, Los Alamitos, 1994. IEEE Computer Society.

[21] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[22] Jerzy Tiuryn. Subtype inequalities. In *Proceedings of the Seventh Symposium on Logic in Computer Science*, pages 308–315. IEEE, June 1992.

[23] Philip L. Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.