Combining Model Checking and Theorem Proving to Verify Parallel Processes[*]

Hardi Hungar

Dept. of Computer Science, University of Oldenburg, D-2900 Oldenburg, Germany

Abstract. To overcome the limitations of pure model checking, this verification technique is combined with theorem proving. Large processes are split into components whose correctness w.r.t. local specifications is checked via model checking. The correctness of the composition w.r.t. the global specification is then established by constructing a formal proof in a derivation system with the help of a theorem prover.

1 Introduction

Techniques for automatic verification of finite state systems are still not powerful enough to cope with real life systems. The reason for this is the complexity of the verification task. The state space of a finite system grows exponentially with the number of its parallel components whereas the size of its description only grows linearly. This phenomenon is usually called "state explosion". The state space is also exponential in the number of (boolean) data variables. Both of the above severely limit explicit enumeration techniques - they have an exponential space complexity. Symbolic techniques seem in many cases to reduce this to a quadratic or cubic complexity [BCMDH90, Fil91]. This suffices for many interesting examples, including abstractions covering relevant aspects of real world systems. But still too many systems can not be verified rigorously.

Our approach to overcome these difficulties is to apply semiautomatic or interactive proof techniques for tasks which can not be done automatically, but to use model checking as much as possible. Roughly, the idea is as follows. A variant of CTL is used to specify processes. This gives rise to a correctness logic: Its basic statements are of the form

$$p \text{ sat } \phi \ ,$$

where p is a program (or process) and ϕ is a formula. Axioms and rules for this logic are implemented in a theorem prover environment. If the process is small enough to permit model checking, this may be done. A successful run of the model checker counts as an axiom of the logic. If it is too big, proof rules have to be applied to reduce the correctness of p to the correctness of smaller processes

[*] This research has been funded by the German Ministery for Research and Technology (BMFT) as part of the project KORSO (Korrekte Software) under grant No. 01 IS 203 P. The responsibility for the contents lies with the author.

(w.r.t other specifications, of course). There are also be further "external" axioms of the proof systems — e.g. temporal logic tautologies which are established by calling a tautology checker. Proofs made up from rules applications may be constructed completely interactively, but the theorem prover environment also offers the possibility to automate common patterns of proofs (via so called *tactics*). Thus, even the scope of fully automatic verification might be increased significantly in this way.

The use of a general purpose theorem prover has further benefits. Model checking is only possible for finite systems. If a system to be checked is infinite, the user first has to generate a finite *abstraction* of the system, then have the abstraction checked and prove that correctness of the abstraction implies correctness of the original system. In a theorem prover framework, one can *formalize* the two steps the user has to do. Since all steps are performed (or at least checked) by the machine, no errors are left in the proof.

At present, a verification system as described above is under implementation. The proof environment is based on the LAMBDA system, which serves as a general purpose, interactive theorem prover. A short description follows which explains what processes and their specifications look like.

A subset of OCCAM is used to describe processes. Basically, a process is a parallel composition of while-programs which are allowed to communicate via synchronous channels. Allowed data types are subranges of the natural numbers or the natural numbers itself.

In the temporal logic, we have atoms to talk about the basics of synchronous communication: whether a (sub-)process is *ready to send* on some channel, whether some data are *transmitted* over a channel, and so on. Also, processes may have global variables whose values a specification might refer to (but which can not be read by other processes to prevent shared variable parallelism). Those atoms are added to Josko's logic MCTL (*modular* CTL, see [DDGJ89, Jos89]). This logic is tailored to meet three requirements:

- allow efficient model checking,
- support the assume/guarantee paradigm, and
- allow powerful proof rules dealing with the parallel composition of processes.

Thus, a specification of a process is a pair $\langle A, C \rangle$ where C describes the behavior of the process under the assumption that A is guaranteed (usually by the environment of the process).

The use of MCTL is of course essential to the approach: We need a proof system dealing with parallel composition of processes which are specified in temporal logic. Less essential is the choice of the programming language. Indeed, within the ESPRIT project FORMAT (Formal Methods in Hardware Verification) it is planned to develop a similar proof environment for a substantial subset of the hardware description language VHDL.[2]

[2] What is described here covers only part of the system to be developed within FORMAT.

In the rest of the paper I will describe in more detail both the theoretical foundations of the verification method (in Section 2) and its implementation (in Section 3).

2 Theory

2.1 Programs

```
<prc> ::= x:= t | c?x | c!t | stop | skip |
          var x init t <prc> | <prc>;<prc> |
          if b → <prc>, ..., b → <prc> fi |
          alt b & c?x→ <prc>, ... tla |
          while b do <prc> od

<prg>::=  <prc> || ... || <prc>
```

Fig. 1. Syntax of Programs

The syntax of processes and programs is given in Figure 2.1. In this table, no mention is made of data types. Allowed types are subranges of natural numbers and the natural numbers itself. On those types, usual arithmetic operations and relations are available. The type information is kept implicit in the names of variables and of channels. When necessary, conversions are done automatically.

A more essential point are the restrictions the static semantics imposes on free variables and channels. In a process, one channel may occur as an *inchannel* ($c?x$) or as an *outchannel* ($c!t$), but not in both forms. A channel may occur in one or two processes of one program: As an inchannel of one process and as an outchannel of another, but neither twice as an inchannel nor outchannel. If a channel occurs twice, it is considered as an internal channel of the program, otherwise it is external. Global variables within processes are allowed, but no process may write to a variable (in a statement $x{:=}\ t$ or $c?x$) which is read by another (in $y{:=}\ t$ or $c!t$ with $x \in \text{free}(t)$ or a condition b with $x \in \text{free}(b)$). This is to prevent shared-variable parallelism. As a result, the only communication between processes which is allowed is via directed channels which connect one process with one other.

Although these restrictions are not strictly necessary for our approach to work, they influence substantially atoms and rules for the specification logic.

Now to the semantics of programs. The intuitive meaning of all the constructs should be clear. Perhaps the only thing which needs some explanation are the clauses in an **alt**-statement. A branch b & $c?x$ $\rightarrow$ <prc> may be chosen if b is true for the current (internal) state of the process and the environment offers an output on c. The first step of its execution is the communication, then the control passes to the process behind the arrow.

Viewed in isolation, the semantics of a process or program is given by an automaton or Kripke structure whose states determine the current position in the program and the values of all variables. Whenever an input statement is to be executed, a suitable value is guessed. To make this semantics compositional, steps of the environment are included. An interleaving semantics is chosen here to keep the structures small. Since the only thing which is visible of the environment is its willingness of the environment to engage in communications, just this has to be added. In this way, we get a modular semantics for our processes and programs.

The generated Kripke structures are finite if all occurring data types are finite. The size of the structure for one process is exponential in the number of channels linking it to the outside and exponential in the number of (bits for) its variables.

Figure 5 gives an example for the generated Kripke structures.

For CTL model checking, a Kripke structure is viewed as a representation of an infinite computation tree. Here, we can not simply take the unwinding of the Kripke structure. Since the environment may make a random step at any time, on some paths no internal step of the process would occur, which is not the intended semantics. To guarantee progress both internally and externally, *fairness constraints* are added. The notion of fairness adopted here is the following. If the control reaches some point infinitely often and one step could always be taken at that point, then this step will infinitely often be executed. I do not want to motivate the choice of this particular notion of fairness. It may suffice to say that it is between weak and strong fairness and that any other sensible notion could be implemented without problems.

2.2 Specification logic

We use an instance Josko's MCTL (*modular* CTL) as specification logic. Its main properties — which motivated its development — are:

- efficient model checking (therefore it is a branching time logic),
- assume/guarantee style of specification: so there is no need to specify the behavior for situations which will not occur. A specification $\langle A , C \rangle$ means that the commitment C may be false if the assumption A is violated.
- Validity of the following rules
  - $$\frac{p \textbf{ sat } \langle A , C \rangle,\ q \textbf{ sat } \langle true , A \rangle}{p \parallel q \textbf{ sat } \langle true , C \rangle}$$
  - $$\frac{p \textbf{ sat } \langle A , C \rangle,\ q \textbf{ sat } \langle B , D \rangle}{p \parallel q \textbf{ sat } \langle A \wedge B , C \wedge D \rangle}$$
  - $$\frac{p \textbf{ sat } \langle A , C \rangle,\ B \rightarrow A,\ C \rightarrow D}{p \textbf{ sat } \langle B , D \rangle}$$

$$
\begin{array}{lll}
<\texttt{atom}> & ::= & \mathbf{rtr}(c) \mid \mathbf{rts}(c) \mid \mathbf{rtx}(c) \mid @c = t \\
\\
<\texttt{assm}> & ::= & \Box(<\texttt{form}> \rightarrow (<\texttt{form}> \mathbf{until} <\texttt{form}>)) \\
& \mid & \Box(<\texttt{form}> \rightarrow (<\texttt{form}> \mathbf{unless} <\texttt{form}>)) \\
& \mid & (<\texttt{form}> \mathbf{until} <\texttt{form}>) \mid (<\texttt{form}> \mathbf{unless} <\texttt{form}>) \\
& \mid & <\texttt{assm}> \wedge <\texttt{assm}> \mid \forall\, x.\, <\texttt{assm}> \\
\\
<\texttt{comm}> & ::= & <\texttt{form}> \mid <\texttt{comm}> \forall\_\mathbf{until} <\texttt{comm}> \\
& \mid & <\texttt{comm}> \forall\_\mathbf{unless} <\texttt{comm}> \\
& \mid & <\texttt{comm}> \wedge <\texttt{comm}> \mid <\texttt{comm}> \vee <\texttt{comm}> \\
& \mid & \forall\, x.\, <\texttt{comm}> \mid \exists\, x.\, <\texttt{comm}>
\end{array}
$$

where `<form>` is a first-order formula with the special atoms of the form `<atom>`.

Fig. 2. Syntax of Formulae.

It turned out that the assumptions should be formulated in linear time logic — otherwise it would be hard to formalize (semantically) the intuitive concept of an assumption. The soundness of the second and third rule above requires that commitments are not existential. This resulted in the choice of the monotonic, universal fragment of CTL as the logic for commitments. The necessity to incorporate assumptions into the model checking procedure imposed further restrictions on the assumptions. The syntax of both logics is given in Figure 2.

The special atoms listed there correspond to the visible effects of the communication statements. Their meaning is:

- **rts**(c): the process with outchannel c is ready to send on c.
- **rtr**(c): the process with inchannel c is ready to receive on c.
- **rtx**(c): A value is being communicated on c at the moment.
- $@c = t$: The value of t is visible on c (which implies that either **rts**(c) or **rtx**(c) is true).

Before a communication can take place, both **rts** and **rtr** have to be true. When the communication starts, both signals become false and **rtx** goes up instead. **rtx** stays high exactly for one step.

The values of those signals are included in the Kripke structures which give meanings to processes. For boolean connectives and the temporal operators the semantics is obvious. $\forall$ and $\exists$ are viewed as abbreviations for $\wedge$ and $\vee$: The value of a variable which is bound by a quantifier is supposed not to change over time. What has to be explained is the role of assumptions.

Similar to fairness constraints, they restrict the set of paths of the computation tree. When evaluating a CTL path quantifier for one particular node in the tree, only those paths are taken into account which obey the restrictions imposed by assumptions and fairness constraints.

Since all path quantifiers in commitments are universal, they are omitted, and *every* subformula is considered to be universally path quantified. As a consequence we get that $\langle \mathit{false}, C\rangle$ is a tautology of MCTL.[3]

The logic and its model checking procedure are described in more detail in [Jos89].

2.3 Proof system

The proof system deals mainly with parallel composition of processes and includes weakening rules. It is based on the rules from [DDGJ89]. We exhibit the main principles and explain their soundness. The formulation of the rules is a little different from the perhaps more intuitive one in the motivation of MCTL. It more closely reflects the form in which they are implemented.

Embedding. Any specification which is satisfied by one process is also true for any parallel composition involving this process.

$$\frac{p \textbf{ sat } \langle A, C\rangle}{p \parallel q \textbf{ sat } \langle A, C\rangle}$$

This is true because of the monotonicity of the commitment. Adding processes restricts the executions.

Modus Ponens. Assumptions which are guaranteed (by some component) can be eliminated.

$$\frac{p \textbf{ sat } \langle B, C\rangle,\ p \textbf{ sat } \langle A, B\rangle}{p \textbf{ sat } \langle A, C\rangle}$$

The soundness of this rule relies on the fact that if B is guaranteed under branching time interpretation, then all computation paths satisfy B in the linear time interpretation, too.

Conjunction. The third rule simply states that under combined assumption the combined commitments do hold.

$$\frac{p \textbf{ sat } \langle A, C\rangle,\ p \textbf{ sat } \langle B, D\rangle}{p \textbf{ sat } \langle A \wedge B, C \wedge D\rangle}$$

[3] Here, it becomes apparent that adding implicitly universal path quantifiers makes a little difference: If we did not do this, $\langle \mathit{false}, \mathit{false}\rangle$ would not be a tautology, since the commitment false is not in the scope of a path quantifier.

Weakening.

$$\frac{p \textbf{ sat } \langle A\,,C\rangle,\; B \rightarrow_{\text{linear time}} A,\; C \rightarrow_{\text{branching time}} D}{p \textbf{ sat } \langle B\,,D\rangle}$$

Strengthening of assumption is sound because of (again) the monotonicity of the commitment logic. Unrestricted use of CTL implication for weakening of commitments needs implicit universal path quantifiers, compare the remarks above. Otherwise, conclusions of e.g. $(\forall\Box C) \rightarrow C$ would not be justified.

Applying the Rules. With the above set of rules, the standard proof proceeds as follows. First, all necessary atomic correctness statements for single processes have to be established (e.g. via model checking). Those are lifted to statements about the whole system under investigation via the embedding rule. Now the conclusions have to be drawn. This is mainly done by applying Modus Ponens. Indeed, a stronger version of this rule is needed where the eliminated assumption is only one conjunct of the assumptions. The other composition rule, which is called conjunction, caters for the situation where two services can be proven independently. This includes the (rare) case of two processes running independently (disjoint parallelism). Note that the rules should not be applied blindly. Whenever possible, assumptions should be eliminated. Otherwise, one might end up with a trivial statement like "assuming that the system is correct, correct behavior can be guaranteed".

There is no completeness result for the proof system. Most probably, it can be complete for a severely restricted class of processes only. One reason is that suitable *modular* specifications for the components of a program will not be expressible in MCTL. Nevertheless, a lot of systems can be handled. A small example illustrating the verification process follows in the next section.

2.4 Example

```
producer                 distributor                          consumer1               consumer2

var $x_{0:0}$ init 0;    var $x_{0:0}$; var $y_{0:1}$;
while true do         ||  while true do                    ||  var $x_{0:0}$ ;       ||  var $x_{0:0}$ ;
  if true→ $c$!0      ||    $c$?$y_{0:1}$; $d$?$x_{0:0}$;   ||  while true do         ||  while true do
    true→ $c$!1 fi;   ||    if $y_{0:1}$ = 0 → $e$!$x_{0:0}$ ||    $e$?$x_{0:0}$       ||    $f$?$x_{0:0}$
  $d$!$x_{0:0}$       ||       $y_{0:1}$ = 1 → $f$!$x_{0:0}$ fi || od                  ||  od
od                       od
```

Fig. 3. An abstract producer/consumer scenario

Figures 3 and 4 present two views of a simple system. A producer process generates addresses and data, sends these to a distributor, which forwards the

```
producer               distributor                    consumer1            consumer2

var x_int init 0;      var x_int; var y_0:1;          var x_int ;          var x_int ;
while true do       || while true do               || while true do     || while true do
  x_int:= random;   ||   c ? y_0:1; d ? x_int;     ||   e ? x_int       ||   f ? x_int
  if true→ c ! 0    ||   if y_0:1 = 0 → e ! x_int  || od                || od
    true→ c ! 1 fi; ||     y_0:1 = 1 → f ! x_int fi
  d ! x_int            od
od
```

Fig. 4. An more concrete producer/consumer scenario

data to the addressed consumer. One view abstracts completely from the data and just keeps the communication structure. In the second, the data may be arbitrary integers. The system has no external channels. Therefore, no assumptions should be used when specifying it. Three aspects of its behavior are formalized in monotonic CTL below.

$$\Box(\mathbf{rtx}(c) \wedge @c = 0 \;\rightarrow [\neg\mathbf{rtx}(f)\;\mathbf{until}\;(\mathbf{rtx}(e) \wedge \neg\mathbf{rtx}(f))]) \qquad (1)$$

This says that if the address 0 is transmitted, *consumer0* is the next to get a value.

$$\Box\forall x.\,(\mathbf{rtx}(d) \wedge @d = x \;\rightarrow$$
$$[(\neg\mathbf{rtx}(e) \wedge \neg\mathbf{rtx}(f))\;\mathbf{until}\;(\mathbf{rtx}(e) \wedge @e = x \;\vee\; \mathbf{rtx}(f) \wedge @f = x)]) \qquad (2)$$

This formula means that a value transmitted on d is the next one which will be delivered via e or f.

$$\Box(\mathbf{rtx}(c) \wedge @c = 0 \;\rightarrow$$
$$\forall x.\,([\neg\mathbf{rtx}(f)\;\mathbf{unless}\;(\mathbf{rtx}(f) \wedge @f \neq x)] \;\vee\; \Diamond[\mathbf{rtx}(e) \wedge @e = x])) \qquad (3)$$

Formula (3) expresses that, after transmission of the address 0, the next value sent on d will eventually be delivered to *consumer0*.

I present a derivation of (1) to illustrate the proof process.

The atomic assertions (established by model checking) are:

$$\begin{aligned}
&\textit{producer}\ \ \mathbf{sat}\ \ \langle \textit{true}, \Box(\mathbf{rtx}(c) \rightarrow \Diamond\mathbf{rts}(d)) \wedge \\
&\qquad\qquad \Box[\mathbf{rts}(d) \rightarrow (\mathbf{rts}(d)\;\mathbf{unless}\;\mathbf{rtx}(d))]\rangle && (4)\\
&\textit{consumer0}\ \ \mathbf{sat}\ \ \langle \textit{true}, \Box\Diamond\mathbf{rtr}(e) \wedge \Box[\mathbf{rtr}(e) \rightarrow (\mathbf{rtr}(e)\;\mathbf{unless}\;\mathbf{rtx}(e))]\rangle && (5)\\
&\textit{distributor}\ \ \mathbf{sat}\ \ \langle \Box[\mathbf{rtx}(c) \rightarrow \Diamond\mathbf{rts}(d)] \wedge \Box[\mathbf{rts}(d) \rightarrow (\mathbf{rts}(d)\;\mathbf{unless}\;\mathbf{rtx}(d))] \\
&\qquad\qquad \wedge \Box\Diamond\mathbf{rtr}(e) \wedge \Box[\mathbf{rtr}(e) \rightarrow (\mathbf{rtr}(e)\;\mathbf{unless}\;\mathbf{rtx}(e))], \\
&\qquad\qquad \Box[\mathbf{rtx}(c) \wedge @c = 0 \rightarrow (\neg\mathbf{rtx}(f)\;\mathbf{until}\;(\mathbf{rtx}(e) \wedge \neg\mathbf{rtx}(f)))]\rangle && (6)
\end{aligned}$$

(4) describes part of the behavior of *producer* which is independent of the environment: After a communication on c, the process will come to a point where

it constantly offers to engage in a communication on d. (5) is a similar statement for *consumer0*, only that this process will without any restrictions arrive at the state where it wants to communicate. Both assertions together provide the assumptions for (6). Applying the embedding rule for all three assertions and then two times Modus Ponens gives the desired result.

The Kripke structures belonging to the abstract system are very small. The one for *producer* is given in Figure 5. So the proof above can completely be carried out on the machine. For the concrete system, this is impossible, because the Kripke structures get too big even when integers are restricted to 32 bit. In the next section it is described how the concrete system can nevertheless be verified.

2.5 Abstracting Out Data

The key to automatically verify (model check) processes with large data domains is to apply *abstractions* first. In [Wol86], Wolper has introduced the notion of a *data independent* process. Intuitively, a process is data independent if its behavior does not depend on the values which it gets as input. This applies to queues or similar devices. The process *distributor* of our example is data independent on the channels d, e and f, though its behavior does depend on the values received on c. To capture processes like this one too, I use a slight generalization of Wolper's original definition.

Definition 1. Let C be a subset of the channels of a process p, and let type(c) for $c \in C$ denote the set of values which may be sent on c. Then p is *data independent on* C if for any function f whose domain is $\bigcup_{c \in C}$ type(c) and any computation tree of p there exists another computation tree of p where in every node the value of $@c$ for $c \in C$ is replaced by $f(@c)$.

A simple syntactic check can show that a process is data independent. What makes this property important is that to verify suitable specifications, the types of the channels can be replaced by very small data types. For example, the commitment of the specification of *distributor* in the proof example above does not mention any data on d, e or f. According to [Wol86], it suffices to verify the specification for a process where the type of those channels has only one element, i.e. it is save to consider the *abstract* version of *distributor*.

Also, in proving the formulae (2) and (3) the data independence of *distributor* can favorably be used. In the commitments to be established for *distributor*, infinite conjunctions appear. Each of the conjuncts mentions the transmission of only one value on d, e or f. The truth of such specifications is left invariant if the integers are replaced by a domain with two elements. Again, the resulting Kripke structures are small enough to be explicitly enumerated.

3 Implementation

The basis of the LAMBDA system [FPZ88, MF91] is an interactive theorem prover for higher-order predicate logic. It offers *tactics* and *rewriting* to automate

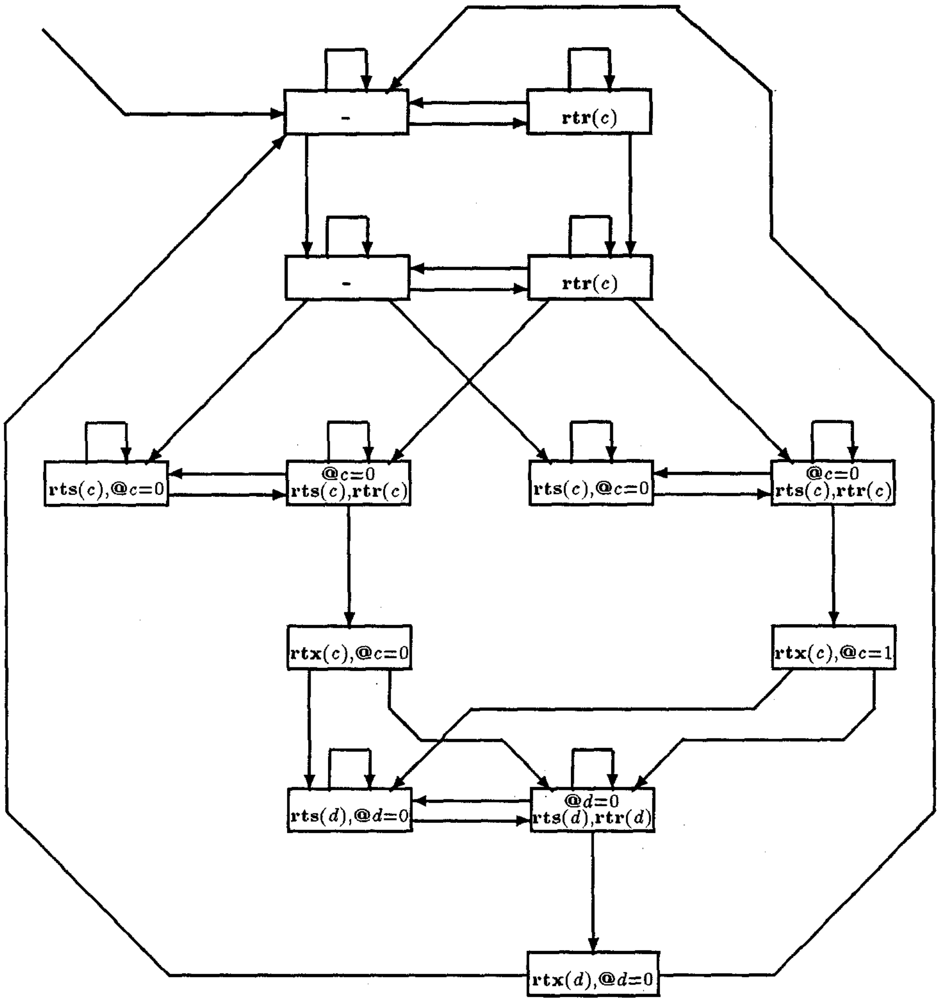

Fig. 5. The Kripke structure describing the behavior of *producer* within an arbitrary environment.

proving. In these respects, it is similar to other systems like HOL [Gor88] or ISABELLE [PN90].[4]

There are two ways in which with the help of such a system a prover for a given program logic can be built. The first one is to embed the program logic in higher-order logic. This involves defining a formal semantics for the program logic in higher-order logic. Any assertion would have to denote its semantics. Proof

[4] LAMBDA has some additional features to support specific styles of hardware verification and synthesis [MF91], which at the moment are not important to us.

rules for the program logic could be derived formally. This approach has the advantage of guaranteeing soundness of the proof system. But the disadvantage is that terms would sometimes be clumsy and that it would take a long time to establish the system. Indeed, in view of the complicated semantic domain for MCTL, one can not hope to do all this in a reasonable amount of time.

Therefore, another way has been chosen: Just the syntax of the program logic has been formalized in higher-order logic. Its rules have simply been added as additional axioms. But also this way requires a lot of work. A formal syntax has to be much more detailed than what one usually writes down when talking about a formal system. Let me exemplify this by presenting the formal syntax of two of the rules.

```
                                    G//H |- P sat_prg (B , C)
G//H |- P sat_prg (A t_and B , C)   G//H |- l_taut(A t_impl B)
G//H |- P sat_prg (A' , A)          G//H |- is_assm A
----------------------------------  -----------------------------
G//H |- P sat_prg (A' t_and B , C)  G//H |- P sat_prg (A , C)
```

The part to the left side of the turnstile (|-) in these rules is LAMBDA-specific and can be ignored here. It simply means that hypotheses play no role in performing these proof steps. Important are the terms to the right. There, a lot of operators do appear which have the following (syntactic) types and informal meanings.

- `sat_prg` : program $\times$ MCTL $\longrightarrow$ truth value. Corresponds to sat.
- `l_taut` : temporal formula $\longrightarrow$ truth value. Tells whether a temporal formula is a linear time tautology.
- `t_and, t_impl` : Constructors for temporal formulae.
- `is_assm` : temporal formula $\longrightarrow$ truth value. Gives true for formulae which obey the restrictions imposed on assumptions (see Figure 2).

There are many further operators. Most of them are partially or completely axiomatized in LAMBDA. If an operator (like e.g. `is_assm`) can completely be axiomatized by a set of equations, the *rewriting* component of LAMBDA can automatically discharge any valid premises with this operator occurring in the proof. For some other operators external programs do exist which decide the validity. The most prominent is of course `sat_prc` for which the model checker can be called. But even for completely axiomatized operators it is favorable to have external programs. To check the static semantics constraints for the simple example from Section 2.4, rewriting within LAMBDA took several minutes whereas a ML program needed at most seconds.

The implementation is not yet complete. Some components are still sort of a prototype. Nevertheless, the abstract producer/consumer scenario from Section 2.4 has been verified, although some of the steps (e.g. the generation of internal fairness constraints) had to be done manually. The missing parts will be added in the near future.

4 Future Work

In addition to complete the system in order to provide the full functionality as described above, some extension are also planned. One major issue will be the development of powerful tactics to automate reasoning in the proof system. Some experience with more realistic examples will help to perform this task. It is planned to verify a communication processor for a direct connection network which guarantees reliable message passing.

On the theoretical side, further concepts are sought for to cope with large data domains. I.e. what is needed is a more powerful way to abstract out data. Also, a *symbolic* model checker would help in this respect, but unfortunately none is available at the moment.

Acknowledgement

I sincerely thank Werner Damm and Bernhard Josko, on whose ideas and previous work the above is based. Further, I would like to mention the students Roland Baumann and Gunnar Wittich who have done most of the programming which was necessary to implement the system.

References

[BCMDH90] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., and Hwang, J. *Symbolic model checking:* 10^{20} *states and beyond*, in: Proc. 5th IEEE Symp. Logic in Computer Science (1990).

[DDGJ89] Damm, W., Döhmen, G., Gerstner, V., and Josko, B. *Modular verification of Petri nets: the temporal logic approach*, in: deBakker, deRoever, Rozenberg (eds) *Stepwise refinement of distributed systems: models, formalisms, correctness*, Springer LNCS 430 (1990), 180–207.

[Fil91] Filkorn, T. *Functional extension of symbolic model checking*, in: Proc. CAV'91.

[FPZ88] Fourman, M., Palmer, W., and Zimmer, R. *Proof and synthesis*, in Proc. ICCD'88, Rye Brook, NY, 1988.

[Gor88] Gordon, M.J.C. *HOL: A proof generating system for higher-order logic*, in: Birtwistle, Subrahmanyam (eds) *VLSI specification, verification and synthesis*, Kluwer (1988), 73–128.

[Jos89] Josko, B. *Verifying the correctness of AADL modules using model checking*, in: deBakker, deRoever, Rozenberg (eds) *Stepwise refinement of distributed systems: models, formalisms, correctness*, Springer LNCS 430 (1990), 386–400.

[MF91] Mayger, E., and Fourman, M. *Integration of formal methods with system design*, in Proc. VLSI'91, Edinburgh.

[PN90] Paulson, L., and Nipkow, T. *ISABELLE tutorial and user's manual*, Tech. Rep. No. 189, Univ. Cambridge Comp. Lab. (1990).

[Wol86] Wolper, P. *Expressing interesting properties of programs in propositional temporal logic*, POPL '86, 184–193.