Journal of Universal Computer Science, vol. 9, no. 2 (2003), 120-137
submitted: 14/10/02, accepted: 14/2/03, appeared: 28/2/03 © J.UCS

Optimized Temporal Logic Compilation

Andreas Krebs
(University of Tübingen, Germany
wsi@krebs-net.de)

Jürgen Ruf
(University of Tübingen, Germany
ruf@informatik.uni-tuebingen.de)

Abstract: Verification and validation are the major tasks during the design of digital hardware/software systems. Often more than 70% of the development time is spent for locating and correcting errors in the design. Therefore, many techniques have been developed to support the debugging process. Recently, simulation and test methods have been accompanied by formal methods such as equivalence checking and property checking. However, their industrial applicability is currently restricted to small or medium sized designs or to a specific phase in the design process. Therefore, simulation is still the most commonly applied verification technique.

In this paper, we present a method for asserting temporal properties during simulation and also during emulation of hardware prototypes. The properties under verification are efficiently translated into an intermediate language (of a virtual machine). This intermediate representation can then be interpreted during simulation. We may also produce executable checkers running in parallel to the simulation. Furthermore, we are able to translate the properties into synthesizable hardware modules which can then be used during system emulation on FPGA-based emulators or as self test components checking the functionality during the lifetime of the system.

Key Words: Verification, Simulation, System-Level, Temporal Logic, Emulation

Category: I.6.6, B.8.1

1 Introduction

Assuring correctness of digital designs is one of the major tasks in the system design flow. Systems in our context are embedded hardware/software systems such as bus arbiters, protocol controllers or microprocessors. These systems are reactive, i.e., they are embedded in an interactive environment and have to react within certain time bounds.

The system design starts with an abstract model describing the main functionality. The extracted system components are partitioned into hardware and software modules. These modules are then further refined until they are implementable on the given target architecture respective in the given hardware technology.

The elimination of design errors can become very expensive, especially if errors are encountered in later design stages. Hence, it is extremely important to find errors as early as possible.

State of the art validation techniques are simulation and test methods. Recently, formal methods such as equivalence checking [Brand 1993] and model checking [Clarke et al. 1990, Biere et al. 1999] found entrance into design laboratories. Model checking is an automated method working well on small or medium sized designs on block level and is usually applied very early in the design process. In contrast to this approach, equivalence checking has successfully been used for verifying large designs and is the matter of choice once a design is brought down to the gate level.

From a validation point of view, formal methods have the desired property that one single verification run implicitly covers 100% of all test cases. The major drawback of formal verification methods, however, is the limited size of verifiable systems. In contrast to formal verification methods, simulation based approaches only provide a partial test case coverage (defined by the simulated test-cases), but do not suffer from combinational explosion and can therefore be applied to very large systems.

Our approach aims at the verification of large systems on high levels of abstraction. Therefore, we have chosen a simulation based approach for validating temporal properties. Our method checks simulation-runs during the validation phase against one or more temporal specifications. We call this approach on-the-fly since the property checking algorithm is directly linked to the simulator and works like an observer during simulation.

Our approach works in two phases: In the first phase we translate the properties into an intermediate language. In the second phase, we use this intermediate representation for different verification techniques:

1. Interpretation of the intermediate language during simulation. This approach allows the validation of the temporal properties during simulation. We can dynamically add and remove specifications.

2. Compilation of the intermediate language. If the properties are translated to executable checker programs, they may be linked to the simulatable design and may be executed very efficiently in parallel to simulation. The interpretational and the compilational approach both directly reports the violation of properties to the designer. Furthermore, both techniques may directly affect the simulation, e.g. they may stop the simulation in case of a property failure and return the simulation trace as counter example.

3. Translation of the intermediate language to hardware monitors. This technique allows the designer to generate synthesizable hardware blocks which may be placed next to the digital hardware components of the system. These hardware monitors may be used for checking the temporal properties during the emulation of system prototypes on hardware emulators. This technique

speeds-up the simulation in software by a factor of 100 and more. Furthermore, the hardware checker components my be used as self-checking modules acting in parallel to the final system running in its real world settings. These modules may indicate functional failures and may switch the system into a fail-safe state.

This paper is organized as follows: Section 2 discusses the state-of-the-art in simulation-based verification techniques. Afterwards we present in Section 3 the formal specification language LTL and the intermediate language which we use for further processing. The translation of LTL formulas into the intermediate language is subject of the Section 4. Section 6 addresses the interpretation and the translation of the intermediate format to software or hardware monitors. We conclude the paper with the presentation of some experimental results in Section 7 and a summary in Section 8.

2 Related work

Several approaches have been proposed in the literature for checking temporal specifications during simulation. The method presented in [Augustin et al. 1988] is utilized for checking event-patterns in VHDL descriptions. The patterns are directly annotated as special comments inside the hardware description language (VHDL). The patterns may be clustered hierarchically, i.e., a pattern may contain sub patterns which have to appear in the specified order. As the patterns can only be linearly chained, the supported logic is less expressive than the logic introduced in this paper.

Nelson and Jones describe in [Nelson and Jones 1994] a simulation-based checking algorithm based on a translation of the properties into finite state machines. The composition of the finite state machines is restricted to sequential chaining of two state machines. Hence, temporal operators can only be combined sequentially restricting the supported logic to temporal formulas that are not nested. Sequential chaining is expressed by a newly introduced "THEN" operator.

Canfield et. al. proposed in [Canfield et al. 1997] a method based on formula manipulation. This approach checks the boolean fraction in the current simulation cycle. If no violation is detected in the current cycle, the temporal operators are unrolled by their fix-point definition and the algorithm is repeated. The approach requires less memory than approaches based on finite-state-machines. However, the algorithm requires considerably more computation overhead in each simulation cycle. In contrast to compilation based approaches which consume the same computation time independent of the formula size, the computation overhead of the algorithm in [Canfield et al. 1997] increases with the size of the formula to check. The approach is, however, well suited in scenarios where memory consumption is more important than simulation time.

The approach presented in [Ruf et al. 2001] uses also a translation of LTL formulas into finite state machines. But this approach does not support neither the translation to executable monitors nor the translation to hardware monitors. Due to the intermediate representation used in the new approach we are able to apply more powerful optimization techniques than used by our old approach.

Verification techniques have also been combined with test-bench generation methods and realized in commercially available tools (e.g., Specman Elite [Verisity], TestBuilder [Cadence] or Vera [Synopsys]). All tools provide object-oriented languages enriched with special constructs for specifying temporal behavior. Temporal specifications are therefore part of the test-bench in contrast to our approach where temporal formulas can be placed inside the system description itself.

3 LTL and the intermediate language

In this section we present the language LTL used for property specification. We also explain the intermediate language (IL) which serves as common base for the following validation techniques.

3.1 LTL

LTL (linear time temporal logic [Emerson 1990]) is a temporal logic which is often used for property specification (e.g. for model checking). The logic consists of atomic propositions. These propositions are the signals of the system under verification. For the rest of this paper, assume $Props = \{a, b, c, \ldots\}$ is a finite set of distinct symbols, called the *atomic propositions*. Furthermore, LTL formulas contain boolean operators (conjunction, disjunction or negation) and temporal operators. The syntax of an LTL formula ϕ is recursively defined by:

$$\phi := \begin{array}{l} prop \mid \neg\phi \mid \phi \wedge \phi \\ \mid X_{[m]}\,\phi \mid F_{[m,n]}\,\phi \mid G_{[m,n]}\,\phi \mid \phi\, U_{[m,n]}\,\phi \end{array}$$

with $prop \in Props$, $m \in \mathbb{N}$ and $n \in \mathbb{N} \cup \{\infty\}$. The X-operator assumes the correctness of the formula ϕ in exactly m time steps. The F-operator assumes the correctness of ϕ within m to n time steps. The G-operator guarantees that the formula ϕ is true at all time steps t with $m \leq t \leq n$. $\phi\, U_{[m,n]}\psi$ expresses that ψ has to become true at t with $t \in [m, n]$ and at all times $v < t$ the formula ϕ is true.

To define the formal semantics of LTL formulas over finite simulation runs we first have to formally fix the notion of "simulation run". We do this by introducing traces.

Definition 1. A trace $T[m..n]$ $(n \geq m)$ is a mapping $T : \{m, \ldots, n\} \rightarrow 2^{Props}$. If m and n are clear from the context, we often simply write T instead of $T[m..n]$. The set of all traces is denoted by $\mathcal{T}$. The set of all traces $T[0..n]$ with $n = \infty$ is denoted by $\mathcal{T}^\infty$.

Definition 2. Let $T[0..m], T'[0..n]$ be two traces with $n > m$. T' is called a *trace extension* of T iff

$$\text{for all } j \text{ with } 0 \leq j \leq m : T(j) = T'(j) \tag{1}$$

LTL formulas are interpreted over traces and evaluated at certain time instances. We first define the semantics of LTL for infinite traces.

Definition 3. Let $T \in \mathcal{T}^\infty$ be an infinite trace and f, g are two LTL formulas. The satisfiability relation $\models_i$ for a time instance $i \geq 0$, is defined recursively over the structure of the LTL formulas:

$$
\begin{array}{ll}
T \models_i a & \text{if } a \in T(i) \\
T \models_i \neg f & \text{if } T \not\models_i f \\
T \models_i f \wedge g & \text{if } T \models_i f \text{ and } T \models_i g \\
T \models_i \mathrm{X}_{[m]} f & \text{if } T \models_{i+m} f \\
T \models_i \mathrm{G}_{[m,n]} f & \text{if for all } j \text{ with } i+m \leq j \leq i+n \text{holds that } T \models_j f \\
T \models_i \mathrm{F}_{[m,n]} f & \text{if there ex. a } j \text{ with } i+m \leq j \leq i+n \text{ such that } T \models_j f \\
T \models_i f \mathrm{U}_{[m,n]} g & \text{if there ex. a } j \text{ with } i+m \leq j \leq i+n \\
 & \quad \text{such that } T \models_j g \text{ and for all } i \leq k < j . T \models_k f
\end{array}
$$

The standard temporal operators (F,G,U) are special cases of the timed operators by instantiating m, n with 0 and ∞, respectively. We now define the semantics of LTL in terms of a satisfiability relation.

Definition 4. Let f be an LTL formula and $T \in \mathcal{T}^\infty$ be a trace. T is called to *satisfy* f (written as $T \models f$) iff

$$T \models_0 f. \tag{2}$$

We now interpret LTL formulas over finite traces.

Definition 5. Let $T[0..n]$ be a trace and f be an LTL formula. f is called *true* with respect to T (denoted by $T \models f$) if for all trace extensions $T'[0..\infty]$ of T holds that $T' \models f$. f is called *false* with respect to T if there exists no trace extension $T'[0..\infty]$ of T such that $T' \models f$. Otherwise f is called *pending*.

The abstract formulas in LTL have the disadvantage that they cannot be checked directly in a linear way. An easy example would be "(X a) ∨ b", where "b" has to be checked first and "a" in the following time step. Considering a formula like "((F a) ∧ b) ∨ ((F c) ∧ ¬ b)" we have to check different expressions in future time steps depending on the values of "previous" ones. So we have to convert an LTL formula into a "linear" language.

3.2 The intermediate language

We use an intermediate language to capture all possible behaviors of the property in an efficient way and to minimize the representation of the formal specification before starting the validation process. Figure 1 shows the overall idea behind our approach.

The advantage of this process is that we can interpret or convert the intermediate language very fast, while the cost of transforming the LTL Formula into this language has to be done only once and does not depend on the simulation. The process is split into two parts "Stage 1" transforming the LTL formula into the intermediate language and "Stage 2" the interpretation or conversion.

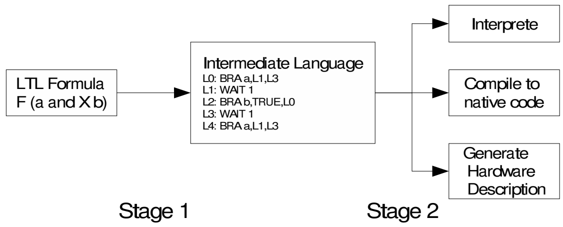

Figure 1: The two phase translation process

The basic commands of our intermediate language are:

WAIT	n	Wait n time steps
BRA	t	Branch to target t
RET	[TRUE|FALSE]	Return true|false.
CMP	c	Compare c to 0
BEQ	t	If the last comparision 'CMP c' succeeded branch to target t, continue otherwise.
BNE	t	If the last comparision 'CMP c' failed branch to target t, continue otherwise.
REQ	[TRUE|FALSE]	If the last comparision 'CMP c' succeeded return true|false, continue otherwise.
RNE	[TRUE|FALSE]	If the last comparision 'CMP c' failed return true|false, continue otherwise.

The conditional branching statements are always accompanied by a compare statement. For a fast access, the signals used for conditions are encoded by an integer index.

We introduce compound commands which are a combination of multiple basic commands. The program shown in Algorithm 1 is internally represented by a single command with 2 arguments "4" and "a", but even if you disassemble a program like this, the command is expanded. There exist up to 128 different compound statements which can be statically defined before the compilation Unless it is needed for optimization purposes, we will not make a difference between basic commands and compound commands.

Algorithm 1 Check X[4] a

```
0000: 6e 04 0000
WAIT 4
CMP v
RNE TRUE
REQ FALSE
```

4 Transformation of LTL into IL

First the formula is parsed and a complete top-to-bottom flow analysis is done. We will construct the IL program bottom to top according to the parse tree. We already have to optimize the code at conversion time in order to reduce the space need and to be more efficient.

4.1 Single variable

A variable "v" with index 0000 can be converted to the statement:

Algorithm 2 Check a single Variable

```
0000: 6e 00 0000
CMP v
RNE TRUE
REQ FALSE
```

4.2 Not operator

The not-Operator will transform the program by exchanging the branch targets "true" and "false".

4.3 Wait operator

The implementation of "X" is also very easy, we simply prepend a wait command to the program.

Algorithm 3 X-Operator

```
WAIT 1
... // remaining program
```

4.4 And operator

The following operation is more complex, but the basic algorithm will be the same for all the missing operators. This algorithm merges several programs to one new program by explicit unrolling parallel executions of the different programs. This operation is used to handle the binary and the other temporal operators of LTL.

We use multiple PCs to traverse the multiple programs or the same program at different positions to construct the new program (see Figure 2). We demonstrate the translation through an example by means of the "and" operator.

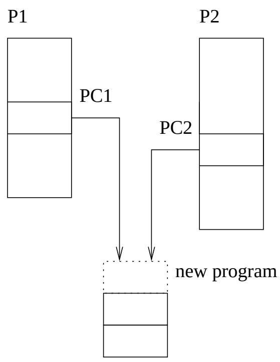

Figure 2: Tracing two programs at the same time

Assume we have the LTL formula "X (a $\vee$ b)" (P1, Algorithm 4) and "a $\wedge$ X b" (P2, Algorithm 5) already converted to the programs (the variable a has index 0000 and b has index 0001).
We simulate both programs P1 and P2 to be executed in parallel. For that we form the set of execution environments (ENVs) consisting only of the PC (program counter) and the W (wait) register. The set of environments to start with is {(P1:0000,0), (P2:0000,0)}. Now we merge the programs by executing the commands of P1 and P2 in turns.

In order to decide which command goes first we introduce an ordered relation upon the set of all possible ENVs of P1 and P2. This order basically sorts the commands in a way that this statement comes first which would be executed

Algorithm 4 P1

```
0000: 20 01
WAIT 1
0002: 6d 00 0000
CMP 0
RNE TRUE
0008: 6e 00 0001
CMP 1
RNE TRUE
REQ FALSE
```

Algorithm 5 P2

```
0000: 66 00 0000
CMP 0
REQ FALSE
0006: 7e 01 0001
WAIT 1
CMP 1
RNE TRUE
REQ FALSE
```

next if the programs were executed in parallel. We discuss this relation later in detail.

We choose the smallest command from our ENVs: because the command at P1:0000 is a wait and P2:0000 has to be executed at this time step, we first append P2:0000 (P2:0000¡P1:0000).

Algorithm 6 P1 and P2 (Step 1)

```
0000: 66 00 0000 (P1:0000,0)(P2:0000,0)
CMP 0
REQ FALSE
```

The "REQ FALSE" statement is kept because if one expression is false the whole expression is false (its a conjunction). The new ENVs are {(P1:0000,0), (P2:0006,0)}.

Now we have to wait for both commands since each wait register is smaller than the steps we have to wait. We simulate one time step by incrementing all wait registers and proceed with the ENVs {(P1:0000,1), (P2:0006,1)}.

The register W is now large enough to execute P1:0000 or P2:0006 next, but according to our order relation we choose P1:0000 (because the variable that has to be compared next in this thread is smaller).

Algorithm 7 P1 and P2 (Step 2)

```
0000: 66 00 0000 (P1:0000,0)(P2:0000,0)
CMP 0
REQ FALSE
0006: 74 01 0000 ???? (P1:0000,0)(P2:0006,0)
WAIT 1
CMP 0
BEQ (P1:0008,0)(P2:0006,1)
```

We cannot simply return true if the condition (P1:0002:CMP 0) is true (the main operation is a conjunction), so we will follow this thread, which forces us to add a jump point to where we check the case that the condition (P1:0002:CMP 0) is false. The wait statement has automatically integrated in the new statement. Notice that we only decrement the W for the ENV of P1 since we did not yet process the wait statement of P2.

Since we continue with the thread that (P1:0002:CMP 0) is true, we only have to follow (P2:0006,1). We can simply copy all commands that can be reached from (P2:0006,1). We have to drop the wait statement of P2:0006 because W is already 1 which means that P2 is already in the correct time step.

Algorithm 8 P1 and P2 (Step 3)

```
0000: 66 00 0000 (P1:0000,0)(P2:0000,0)
CMP 0
REQ FALSE
0006: 74 01 0000 ???? (P1:0000,0)(P2:0006,1)
WAIT 1
CMP 0
BEQ (P1:0008,0)(P2:0006,1)
0010: 6e 00 0001 (P2:0006,1)
CMP 1
RNE TRUE
REQ FALSE
```

We now have to follow the thread we did not follow yet. So we restore the ENVs to {(P1:0008,0), (P2:0006,1)}.

Both commands can be executed without a wait, and they compare the same variable b, so they can be executed in one command. We add a conditional branch statement that compares b. If b is true both statements would return true, so we will also return true (actually we drop all statements that return true as described before and get the empty set of ENVs and because of this we return true). If b is false both statements would return false, so we will also return false (also if only one of the statements would return false we return false).

Algorithm 9 P1 and P2 (Step 4)

```
0000: 66 00 0000 (P1:0000,0)(P2:0000,0)
CMP 0
REQ FALSE
0006: 74 01 0000 ???? (P1:0000,0)(P2:0006,1)
WAIT 1
CMP 0
BEQ (P1:0008,0)(P2:0006,1)
0010: 6e 00 0001 (P2:0006,1)
CMP 1
RNE TRUE
REQ FALSE
0016: 6e 00 0001 (P1:0008,0)(P2:0006,1)
CMP 1
RNE TRUE
REQ FALSE
```

Now we have followed all threads. There is one case that did not appear in this sample if we detect at any stage that the set of ENVs already exists in the resulting algorithm we insert a goto statement to that location. This operation might introduce loops.

In the final step we "link" the program by removing the symbolic targets and replace them with the absolute addresses.

Algorithm 10 P1 and P2 (Step 5)

```
0000: 66 00 0000
CMP 0
REQ FALSE
0006: 74 01 0000 0016
WAIT 1
CMP 0
BEQ 0016
0010: 6e 00 0001
CMP 1
RNE TRUE
REQ FALSE
0016: 6e 00 0001
CMP 1
RNE TRUE
REQ FALSE
```

4.5 Other operators

In a similar way we can implement the "or" operator. For the "G" and "F" operator we use a list of ENVs to traverse the program, starting with the list that contains only the initial ENVs. Whenever we wait one time step we add the initial PC to the list of ENVs. In this way we simulate "e $\wedge$ X e $\wedge$ X X e $\wedge$ ..." for "G e" ("e $\vee$ X e $\vee$ X X e $\vee$ ..." for "F e"). For the time bounded operator we interrupt this unrolling if the time bound is reached. Operators, including unbounded once, only need to be unwound finitely often, because the number of new instructions is limited by the power set of the old instructions.

With these commands we also implemented a limited "U" (until) Operator. One operand of the until operator has to be a boolean expression. There exists an experimental implementation allowing instead of a boolean expression a time limited temporal expression, i.e., unlimited "G", "F" or "U" are excluded as operands of an until operator.

5 Optimizations

Before we further operate on the intermediate language programs for validation, we apply some optimizations to reduce program size or execution speed. These optimizations are divided in two classes: optimizations to decrease compile time and optimizations to increase execution speed.

5.1 Compilation time optimizations

The most direct optimization during compilation time is to store already compiled expressions or subexpressions and reload them if they are used in other expressions. The recognition of expressions is modulo variable names, i.e., many small expressions may be found quickly. The cache can also be written to hard disk, in order to provide a precompiled set of often used specifications.

One of the most important optimizations is that the commands in the programs are "ordered". If during one time step two variables are checked e.g. "a $\wedge$ b" we will always check "a" first. So the expressions "a $\wedge$ b" and "b $\wedge$ a" compile to the exact same IL code. This allows us to merge the conditional branches very effectively during the compilation. Another effect of this ordering is that every variable is checked at most once during a time step. The variables within one time step are ordered by their index. This allows the merge operation to generate small and fast programs.

The second optimization is used after the execution of "some" compilation steps. It searches for traces that return equal results and merges them. This optimizations takes very long. Therefore we apply it only after a heuristically determined number of compilation steps or if the operation that was last compiled usually creates a lot of equal traces e.g. "G", "F" or "U". The problem with this optimization is that we first have to compile the full expression and then reduce its size, i.e. for statements like: "G a $\rightarrow$ F[20] b", this optimization does not improve compilation time unless the expression is used as a subexpression in another formula. But in any case it reduces the program size. An experimental version of this optimization runs during compilation time. It uses additional information about traces, and makes it therefore possible to reduce the size of the result. In the example "G a → F[20] b" the subexpression "F[20] b" creates the information that it contains a trace of 21 consecutive statements, that it can only become true under the condition that "b" is true, and that otherwise it will fail.

Another way to speed up compilation time is to compile only parts of the expression. The remaining parts will be interpreted. This means that at execution time a dynamic set of ENVs has to be handled. Sticking to the example above the "a → F[20] b" operation would generate code that checks "F[20] b" each time "a" is true. And returns true/false/pending according to the results of "F[20] b". Of course this forces us to use multiple execution environments since multiple "F[20] b" expressions might be executed in the same turn. So this will reduce execution time by large if "a" is very often true. But if we know that "a" is not very often true and/or the expression is very long, the increase in compilation time is worth the reduction in execution time. For instance, to compile the formula "G(a → X[20] b)" needs $2^{20} = 1048576$ statements! The formula "a → X[20] b" needs 2 statements. In this situation it is worth to interpret the outermost G operator to avoid the blowup in the program size. Currently the user has to specify when interpretation is preferred over compilation. Therefore we have introduced some new operations forcing the compiler to use interpretation, e.g. the expressions above becomes "F[20] b when a".

5.2 Execution time optimizations

Multiple operations may be packed to one compound command. Especially the combination WAIT, CMP, BNE/RNE, BEQ/REQ occur very often and therefore we reduced these combinations to one compound statement.

Assume we have an LTL expression "exp" that always has to be true if a certain signal "a" is true. The LTL expression will be "G a → exp". Depending on the index of the signal "a" and the signals in exp, the left side of the implication might be checked before the right side or vice versa. To support the compiler by optimizing formulas which are often evaluated to true or to false, we added special hint operators. For instance "a |→ b" (if a is false most of the time) or "a →| b" (if b is true most of the time). Similar hints can be given for the other operators. The compiler uses these hints to order the evaluation of variables in the most efficient way. Another possibility is to count the frequency certain variables are true or false. For instance most interrupt signals of a CPU are normally false, where a clock signal is 50% of the time false and 50% of the time true.

In certain situations optimization may cut down execution time extremely.

6 Interpretation and conversion

In this section we describe different techniques which can be applied to the intermediate language programs. These techniques are used for the simulation- or emulation-based verification of hardware/software systems.

Interpretation of the intermediate language is very easy and straightforward. We need two variables the PC and a wait register W.

Depending on the interface we can either follow the control flow and actively wait when we reach a wait statement, or return with result pending. In most cases the interpreter will be called once every time step, so we have to return when we reach a wait statement. We will use W to wait multiple time steps at a wait statement with a parameter greater than 1.

The other way is to compile the intermediate language to native machine code. Since all CPUs, even simple ones, contain at least one register for W and can handle conditional branches, this again is very straightforward.

A third possibility is to convert the intermediate language into a hardware description. We will show a very simple way of doing this on the previous example Algorithm 5.

The final hardware monitor contains one "start" signal for launching the checking process and one "clock" for triggering all internal flip-flops. For each signal used in the formula, there is one input to the checker. The output signals "true" and false indicate the cuccess or the failure of the checking process. The main checker circuit is depicted in Figure 3 a).

Figure 3 b) shows the mapping of IL-commands to hardware elements. We need a 2-multiplexer for every conditional branch and a delay register (edge triggered flip-flop) for every wait statement. We can also use the first and the second hardware blocks for REQ, RNE and RET respectively. In this case we have to connect the targets with the true or the false output. We connect the "start" signal to the first component. If n different branch commands of the program have the same target it is necessary to introduce an or-gate with fan-in n to collect all branch activations.

We connect the components according to the control flow. The resulting circuit of Algorithm 5 is shown in Figure 4.

Similar to the interpretation option used to reduce the compile time, we can apply this option in the hardware circuit. The idea is to start the checking process in several consecutive clock cycles. Since the signals propagate in a pipeline manner through the circuit we do not get any interference between the checking signals started in different clock cycles. This technique is best demonstrated by means of an example. consider a formula with an outermost G operator. This formula has to hold in each clock cycle. So we do not compile the G-operator into the hardware but we assert the start signal in each clock cycle. If once the

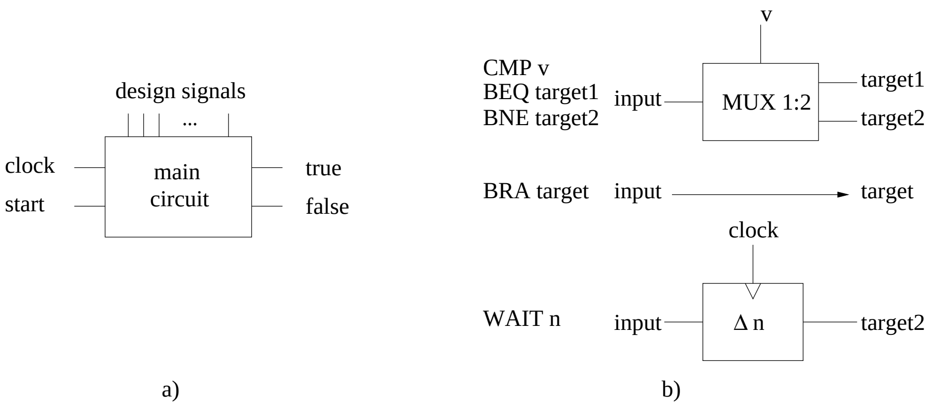

Figure 3: Basic blocks for hardware translation

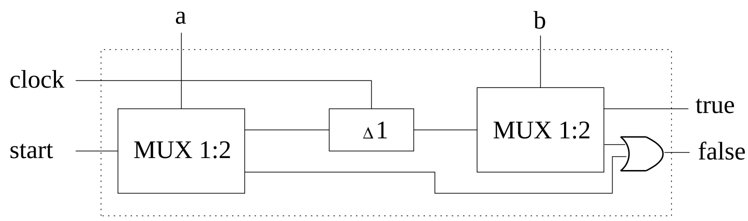

Figure 4: Final checker circuit

false output becomes active, we know that there were a violation of the formula under consideration.

7 Experimental results

For performing some experimental results we have implemented our approach in C++. We have coupled the IL-compiler with the SystemC simulation kernel [Grötker et al. 2002]. The checker interface is accessed via a set of library functions. LTL specifications may directly be placed in the system description.

We have executed some experiments by means of a scalable arbiter circuit specified in SystemC. This circuit is described in [Ruf et al. 2001]. The circuit controls the mutual exclusive access of multiple components to a shared resource (e.g. a bus). The arbiter combines a priority access control with a round robin schedule for guaranteing the fairness. The circuit consists of one arbitration cell for each accessing component. These cells are connected in a regular way.

For each arbiter cell we checked: "G ($\mathrm{req}_i \rightarrow$ (F[$2n$] ack_i)) $\wedge$ ($\mathrm{ack}_i \rightarrow \mathrm{req}_i$)". In order to see any compile times at all, the compiler cache was disabled. With more arbiter cells the compilation time is actually increasing linear, but the circuit simulation time increases over proportional. The execution time is zero because it is too small for measurements. Our results do not show measurable run-time difference between interpretation of the IL programs or compilation into native C++-code. The total run time was about 20-300 seconds and the execution time of all runs was less then 1/100 second. The results are shown in Table 1.

A second example was the micro controller interface for the I^2C-protocol [Philips 2000] used for on-chip communication. We verified an abstract system description consisting of two nodes connected to the I^2C-bus. We have checked the following LTL expressions:

- The I^2C-bus is a pull-down bus, i.e. a low input signal pulls down the bus signal to a low voltage value:
  "$G(\neg sclo0 \rightarrow \neg sclo)$", "$G(\neg sclo1 \rightarrow \neg sclo)$",
  "$G(\neg sdao0 \rightarrow \neg sdao)$", "$G(\neg sdao1 \rightarrow \neg sdao)$"
- Every starting frame on the bus will eventually be terminated.
  "$G((sdao \wedge sclo \wedge X (\neg sdao \wedge sclo)) \rightarrow F (sclo \wedge \neg sdao \wedge X (sclo \wedge sdao)))$"

The compilation time with enabled formula cache and the run-time overhead are zero.

We have investigated other real-world systems with our approach, e.g. a holonic material transport system, a radio-based railway crossing or a micro controller interface for sequential communication.

For examining our compiler, we checked the comile times for different formulas. The results are shown in Table 2.
The last three examples show a formula producing an exponential amount of commands. Since the only register is the program counter, it contains all information about the values of "a" in the last 8, 12, 16 time steps. This blow-up can be avoided by the interpretation optimization technique.

number of arbiter cells	10	20	30	40	50	60	100	160
compilation time overhead (%)	21.3	22.6	20.6	17.8	18.3	16.7	13.5	10.0
execution time overhead (%)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 1: Runtime overhead caused by the compilation (without caching) and checking (compared to pure simulation runtime)

formula	compilation time
G((a ∧ c ∧ X(¬a ∧ c))→F(c ∧ ¬a ∧ X(c ∧ a)))	0.01 seconds
G((a ∧ c ∧ X(¬a ∧ c))→F[8](c ∧ ¬a ∧ X(c ∧ a)))	0.01 seconds
G[10]((a ∧ c ∧ X(¬a ∧ c))→F[8](c ∧ ¬a ∧ X(c ∧ a)))	0.01 seconds
G((a → F b) ∧ (c → d ∨ X[5] e))	0.06 seconds
G ((e ∧ b) → ((a ∧ X[3] b) ∨ (c → d ∨ X[3] e)))	0.03 seconds
G (a → X[8] b)	0.01 seconds
G (a → X[12] b)	0.13 seconds
G (a → X[16] b)	3.63 seconds

Table 2: Compile times for different formulas

8 Conclusion

We have presented a simulation-based approach for checking temporal assertions (LTL formulas) in digital hardware/software systems. LTL formulas are translated to an intermediate language. This intermediate language may be interpreted or compiled for a simulation-based system validation. The intermediate language may also be translated to hardware observers for emulation-based verification or for built-in self testing units.

We have presented a number of optimization strategies to decrease the intermediate language program size and to speed up the validation process. By means of real-world case studies we have shown the practicability of our approach.

For future work we plan to integrate further optimization strategies and we will extend the input language to industrial standards like sugar [Sugar 2001]. We also plan to extend this approach to branching time logic by collecting the simulation results of different runs.

References

[Augustin et al. 1988] L.M. Augustin, B.A. Gennart, Y. Huh, D.C. Luckham, and A.G. Stanculescu. Verification of VHDL designs using VAL. In *Design Automation Conference (DAC)*. ACM/IEEE, 1988.

[Sugar 2001] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic sugar. *Lecture Notes in Computer Science*, 2001.

[Biere et al. 1999] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, 1999.

[Brand 1993] D. Brand. Verification of Large Synthesized Designs. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.

[Clarke et al. 1990] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)*, Washington, D.C., June 1990. IEEE Computer Society Press.

[Canfield et al. 1997] W. Canfield, E. Emerson, and A. Saha. Checking formal specifications under simulation. In *International Conference on Computer Design (ICCD '97)*. IEEE Computer Society Press, 1997.

[Emerson 1990] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, Amsterdam, 1990. Elsevier Science Publishers.

[Grötker et al. 2002] T. Grötker, S. Liao, G. Martin, and S. Swan. *System design with SystemC*. Kluwer Academic Publishers, 2002.

[Nelson and Jones 1994] B.E. Nelson and R.B. Jones. Simulation event pattern checking with proto. In *SHDL 1994*, 1994.

[Philips 2000] Philips Semiconductor. *The I^2C-Bus Specification Version 2.1*, January 2000. http://www.semiconductors.philips.com/buses/i2c/facts.

[Ruf et al. 2001] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued AR-automata. In *Design Automation and Test in Europe (DATE)*. IEEE Conmputer Society Press, 2001.

[Synopsys] www.synopsys.com.

[Cadence] www.testbuilder.net.

[Verisity] www.verisity.com.