Ground Temporal Logic: A Logic for Hardware Verification

(Slightly expanded from the 6th International Conference on Computer Aided Verification (CAV '94), LNCS #818, pp 247-259, Stanford, CA 1994).

David Cyrluk*
Dept. of Computer Science
Stanford University and
Computer Science Laboratory
SRI International
cyrluk@cs.stanford.edu

Paliath Narendran †
SUNY-ALBANY, Albany NY
dran@cs.albany.edu

Abstract

We present a new temporal logic, *GTL*, appropriate for specifying properties of hardware at the register transfer level. We argue that this logic represents an improvement over model checking for some natural hardware verification problems. We show that the validity problem for this logic is Π^1_1 complete. We then identify a fragment of the logic that is decidable. We show that in this fragment we are still able to encode many interesting problems, including the correctness of pipelined microprocessors.

1 Introduction

Temporal logic is a natural logic for hardware verification. Specifically model checking for various propositional temporal logics has proven to be a very practical tool for the fully automatic verification of many hardware circuits and finite state protocols. However these approaches suffer from various drawbacks.

One such drawback is the requirement that hardware implementations be carried out to the bit-level. This can lead to the state explosion problem as the number of states can increase exponentially with the number of bits in the implementation. It also necessitates a bit-level description of alus and adders. To deal with this problem current research relies on tools such as BDD's to encode a large number of states into a small representation [3, 6, 5]. [8] makes use of abstractions to significantly reduce the state space that needs to be explored.

However, the correctness argument for many of these circuits does not depend on a bit-level description of the circuit but only on a RTL description of the circuit. In such cases the correct abstraction is to abstract away from the bit-level using uninterpreted function symbols. Thus, perhaps, a first order temporal logic might be more appropriate for this type of hardware verification. The main drawback with using a full first-order temporal logic is that the validity problem now becomes incomplete, thus making automatic verification impossible.

*This research was partially supported by SRI International, DARPA contract NAG2-703, and NSF grants CCR-8917606, CCR-8915663.

†Much of this research was done while a visiting scientist at SRI International.

We thus propose a Ground Temporal Logic (GTL) that falls in between first-order and propositional temporal logic. To make this logic more expressive than propositional temporal logic we need to add a new *Next* temporal operator that operates on terms instead of on formulas. This allows us to relate successive states using uninterpreted functions without having to use quantified variables, and thus a first-order logic. For example, a typical statement one would want to encode is: *In the next state the value of C becomes $f(C)$.* In a first order temporal logic this would be encoded as $\forall x : C = x \supset \bigcirc(C = f(x))$, but in GTL, this could be more naturally stated without quantifiers: $\circ(C) = f(C)$.

Unfortunately this ground temporal logic is as undecidable as the full first-order logic. We identify a fragment of this logic that is straightforwardly decidable and yet still suitable for hardware verification. This fragment is expressive enough to express the correctness of the RSRE counter [9] verified using the interactive theorem prover, HOL. This example first motivated the definition of our language. On the one hand, we believed that the model-checking techniques associated with propositional temporal logic were more appropriate in verifying the correctness of the counter than theorem proving. On the other hand, by using a theorem prover we were able to abstract away from a bit-level description of the counter and thus obtain a more concise proof of correctness, that is independent of the size of the counter.

Our fragment is also expressive enough to express the correctness of the pipelined ALU circuit that has become a benchmark in the model checking community [8, 6, 5]. A goal of the model checking community is to find techniques that allow them to effectively verify the pipelined ALU with increasingly larger datapaths and register file. Our logic lets us verify the pipelined ALU once and for all for arbitrarily large datapaths and register file and for an arbitrary number of alu instructions. The cost we incur is that our fragment is much less temporally expressive than the decidable propositional temporal logics.

Using theorem proving techniques we have in the past verified several microprocessors such as Saxe's pipeline [20, 10]. We are currently verifying a more realistic microprocessor—a Verilog model of a much simplified MIPS R3000 processor. The correctness of these circuits is also expressible in our decidable fragment. In the future we can make use of this fragment by either implementing the fragment independently or by integrating it into a theorem proving environment.

The paper is organized as follows. Section 2 presents GTL. Section 3 gives a proof of the incompleteness of the logic. In section 4 we present a decidable fragment. In section 5 we extend this fragment so that various interesting verification problems can be described in it. Section 6 shows how to enode aspects of microprocessor verification in our logic. The final section presents conclusions and future work.

2 Ground Temporal Logic

In this section we give the syntax and semantics of the first-order temporal language we will be using. We follow the presentation of Kröger [14].

2.1 Syntax

Given a first order language, FOL, consisting of function symbols, constants (0-ary function symbols), predicate symbols, equality, but no variables, we define the language GTL as follows.

The alphabet of GTL is the alphabet of FOL along with $\bigcirc$, $\circ$, $\Box$, and a set of state variables, V_s, whose values can change with time. There are no global variables.

The *terms* of GTL are defined inductively. Every state variable is a term. If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term. If t is a term then so is $\circ t$.

The *atomic formulas* of GTL are defined as follows:

If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is an atomic formula.

Formulas of GTL are defined inductively. Every atomic formula is a formula. If A and B are formulas then so are $\neg A, A \wedge B, \bigcirc A$, and $\Box A$. The language of GTL is the smallest language generated by these rules.

Note that this language is the quantifier free, ground version of the language of First Order Temporal Logic as presented in [14] with the addition of our $\circ$ operator.

2.2 Semantics

Closely following the presentation in [14] we define the semantics of GTL.

We define a *model* $\mathbf{K} = (\mathbf{S}, \mathbf{W})$ for GTL as follows.

$\mathbf{S}$ is a model for the first-order language FOL. $\mathbf{S}$ consists of a non-empty universe $|\mathbf{S}|$, an n-ary function $\mathbf{S}(f) : |\mathbf{S}|^n \rightarrow |\mathbf{S}|$ for every n-ary function symbol f, and an n-ary relation $\mathbf{S}(p) \subset |\mathbf{S}|^n$ for every n-ary predicate symbol p other than $=$.

$\mathbf{W} = \{\eta_0, \eta_1, \ldots\}$ is an infinite sequence of state variable valuations (*states*): $\eta_i : V_s \rightarrow |\mathbf{S}|$.

We define two evaluation functions:

$\mathbf{S}_t^{(\eta_i)}$: *terms* $\rightarrow |\mathbf{S}|$, and

$\mathbf{S}_a^{(\eta_i)}$: *atomic formulas* $\rightarrow \{\mathbf{f}, t\}$.

- $\mathbf{S}_t^{(\eta_i)}(a) = \eta_i(a)$ for state variable a.
- $\mathbf{S}_t^{(\eta_i)}(f(t_1, \ldots, t_n)) = \mathbf{S}(f)(\mathbf{S}_t^{(\eta_i)}(t_1), \ldots, \mathbf{S}_t^{(\eta_i)}(t_n))$, for f other than $\circ$.
- $\mathbf{S}_t^{(\eta_i)}(\circ t) = \mathbf{S}_t^{(\eta_{i+1})}(t)$
- $\mathbf{S}_a^{(\eta_i)}(p(t_1, \ldots, t_n)) = \mathbf{t}$ iff $(\mathbf{S}_t^{(\eta_i)}(t_1), \ldots, \mathbf{S}_t^{(\eta_i)}(t_n)) \in \mathbf{S}(p)$ for p other than $=$.
- $\mathbf{S}_a^{(\eta_i)}(t_1 = t_2) = \mathbf{t}$ iff $\mathbf{S}_t^{(\eta_i)}(t_1) \underset{|\mathbf{S}|}{=} \mathbf{S}_t^{(\eta_i)}(t_2)$.

We now define the truth value function $\mathbf{K}_i$: formulas $\rightarrow \{\mathbf{f}, t\}$ for every $i \geq 0$.

- $\mathbf{K}_i(A) = \mathbf{S}_a^{(\eta_i)}(A)$ for atomic formula A.
- $\mathbf{K}_i(\neg A) = \mathbf{t}$ iff $\mathbf{K}_i(A) = \mathbf{f}$.
- $\mathbf{K}_i(A \wedge B) = \mathbf{t}$ iff $\mathbf{K}_i(A) = \mathbf{t}$ and $\mathbf{K}_i(B) = \mathbf{t}$.
- $\mathbf{K}_i(\bigcirc A) = \mathbf{t}$ iff $\mathbf{K}_{i+1}(A) = \mathbf{t}$.
- $\mathbf{K}_i(\Box A) = \mathbf{t}$ iff $\mathbf{K}_j(A) = \mathbf{t}$ for every $j \geq i$.

Definition: A formula A of GTL is valid in the model $\mathbf{K}$ if $\mathbf{K}_i(A) = \mathbf{t}$ for every $i \geq 0$. A is *valid* iff A is valid in every model $\mathbf{K}$. A is *satisfiable* iff A is valid in some model $\mathbf{K}$.

An alternate method for giving the semantics of the language is to divide the set of function symbols and constants into rigid and flexible sets. The approach we have taken uses V_s as the set of flexible constants. Allowing a set of flexible non-constant function symbols does not make the language more expressive.

3 Undecidability of Ground Temporal Logic

The logic presented in Section 2 is the simplest and in some way the smallest extension to propositional temporal logic that makes the problem of determining validity undecidable. We prove this result by encoding the recurrence problem for two-counter machines in the logic.

3.1 Two-counter machines

A *two-counter machine* is a sextuple $M = (Q, C_1, C_2, \Delta, q_0, q_f)$, where Q is a finite set of states. C_1 and C_2 are natural numbers. The operations zero?, add1, and dec1 defined on the counters. $q_0 \in Q$ is the initial state. $q_f \in Q$ is the final state. Δ is the transition relation. Δ is a finite subset of $((Q \times \{\mathbf{f}, t\} \times \{\mathbf{f}, t\}) \times (Q \times \{\mathrm{noop}_1, \mathrm{inc}_1, \mathrm{dec}_1\} \times \{\mathrm{noop}_2, \mathrm{inc}_2, \mathrm{dec}_2\}))$

Definition: A *configuration* of the machine is a quadruple: (q, c_1, c_2), where q is a state, and c_1, c_2 are the values of the two counters.

Informally, if the machine is in state (q_1, c_1, c_2), and if $((q, f, t), (q', op_1, op_2)) \in \Delta$ then if c_1 is non-zero and c_2 is zero the machine can transition to configuration (q', c_1', c_2') where c_i' is obtained from c_i by applying the appropriate operation op_i.

We say $(q, c_1, c_2) \vdash_M (q', c_1', c_2')$ if such a transition can occur.

A *computation* of M is a possibly infinite sequence of configurations starting with $(q_0, 0, 0)$, and where config_{i+1} follows config_i just in case $\mathrm{config}_i \vdash_M \mathrm{config}_{i+1}$.

A computation is *recurring* iff it contains infinitely many configurations whose first component is q_0.

Lemma: [2] The problem of deciding whether a two-counter machine has a recurring computation is Σ_1^1-hard.

3.2 Encoding Natural Numbers in *GTL*

The first step in encoding two counter machines is to enocode nautral numbers. The following formula defines the successor (s) and predeccesor (p) functions:

nat:

$$\begin{array}{ll} C = 0 & \wedge \\ \Box(\circ C = s(C)) & \wedge \\ \Box(0 \neq s(C)) & \wedge \\ \Box(p(s(C)) = C) & \end{array}$$

The following theorem then establishes that s is the successor function for natural numbers and that $s^n(0)$ encodes the natural number n.

Theorem: $s^m(0) = s^n(0) \supset m = n$

3.3 Validity of *GTL* is Π_1^1-complete

We now prove the main complexity result concerning *GTL*.

Theorem: Validity of *GTL* is Π_1^1-hard

Proof: We show how to encode the computations of a two-counter machine in *GTL*.

We use local variables, *state,* C_1, C_2, a constant symbol, 0, the unary function symbol s and p defined above, and constant symbols for each state in Q. C_1 and C_2 encode the values of the two counters.

The *initial condition* of the machine is encoded as:

initial: $state = q_0 \wedge C_1 = 0 \wedge C_2 = 0$

The transitions can also be easily encoded by GTL formulas. For example the transition: $(q, 0, C_2) \vdash_M (q', 0, C_2 + 1)$ would be encoded as:

$$\begin{array}{l}\Box[(state = q \wedge C_1 = 0\ ) \supset \\ \quad (\circ\, state = q' \qquad \wedge \\ \quad\ \circ\, C_1 = C_1 \qquad \wedge \\ \quad\ \circ\, C_2 = s(C_2))]\end{array}$$

Transitions for decrementing a counter are also straight forward. The transition for: $(q, 0, C_2) \vdash_M (q', 0, C_2 - 1)$ would be encoded as:

$$\begin{array}{l}\Box[(state = q \wedge C_1 = 0\ ) \supset \\ \quad (\circ\, state = q' \qquad \wedge \\ \quad\ \circ\, C_1 = C_1 \qquad \wedge \\ \quad\ \circ\, C_2 = p(C_2))]\end{array}$$

We denote the conjunction of such formulas describing the transitions of a two-counter machine as: **transitions**.

The formula that says that the initial state is reached infinitely often is:

recurring: $\Box\Diamond(state = q_0)$

For a given two-counter machine the formula stating that the counter has a recurring computation is:

$$\textbf{nat} \wedge \textbf{initial} \wedge \textbf{transitions} \wedge \textbf{recurring}$$

This formula is satisfiable iff the counter machine it encodes has a recurring computation. Thus the satisfiability problem for GTL is Σ^1_1-hard, and the validity problem is Π^1_1-hard.

Lemma: Validity of GTL is Π^1_1.

Proof: The validity of a formula, A in GTL can be stated as: $\forall \mathbf{K} \forall i (\mathbf{K}_i(A) = \mathbf{t})$, which is a Π^1_1 sentence.

Theorem: Validity of GTL is Π^1_1-complete.

Proof: Follows immediately from the previous lemma and theorem.

Although this is a discouraging result, it provides useful guidance in developing logics that are both expressive and can be automated; we have an upper bound on how expressive such a logic can be. The rest of this paper presents a fragment of GTL that can both express interesting properties of hardware and be decided.

4 A Decidable Fragment

We now present a fragment of GTL that is decidable. While this fragment is very simple, we illustrate its usefulness by encoding the correctness of a simple counter in it. In later sections we will extend this fragment to make it more useful until we eventually are able to express the correctness of a simple microprocessor.

GTL was motivated by a desire to combine the best features of model-checking and theorem proving into one logic. The algorithms from model-checking yields relatively efficient decision procedures that allow the automatic verification of hardware circuits. Theorem proving allows the verifier to abstract away from some of the bit-level details that are irrelevant for the proof of correctness, but is in general undecidable and requires a large amount of human effort.

By adding uninterpreted function symbols to a propositional temporal logic, we are trying to make use of the most natural type of abstraction, but still do something similar to model-checking.

The model checking problem is, given a model $\mathcal{M}$ and a specification S, to determine whether:

$$\mathcal{M} \models S$$

is true.

In propositional temporal logic $\mathcal{M}$ is a finite Kripke structure, and S is a propositional temporal formula.

In *GTL* $\mathcal{M}$ may or may not be a finite Kripke structure. It would be if the interpretations of the uninterpreted function symbols were functions with a finite domain and range. For example, in Figure 1, a possible interpretation for the *inc* function would be *plus one mod four*, yielding a 4-bit counter. In this case the model-checking problem for *GTL* would reduce to that of a propositional temporal logic.

However, in some cases there are no finite models or, more importantly, we want to verify that all models satisfy the specification S. Thus, instead of model checking we want to determine the validity of formulas of the form

$$I \supset S \tag{1}$$

where I is a *GTL* formula defining an implementation, and S is as before, a *GTL* formula.

GTL has been designed so as to be able to encode transition systems. The use of the $\circ$ operator allows the encoding of a next-state relation. If we restrict I to only those formulas that encode transition systems, and S to properties of the system that can be checked by exploring fixed length paths, then the validity problem for the formulas of the form in equation 1 is decidable. Note that I is not capable of stating an initial state for the transition system. This is crucial for our decidability argument. Note also that the restrictions that S refer only to fixed length paths is similar to the restrictions explored in a propositional temporal logic in [4].

Before giving details we illustrate this with an example. The transition from the Inc1 state to the Fetch state in Figure 1 would be encoded in *GTL* as:

$$\Box[\mathrm{state} = \mathrm{Inc1} \wedge \mathrm{double} = \mathrm{false} \supset \circ\, \mathrm{state} = \mathrm{Fetch} \wedge \circ\, \mathrm{Cnt} = \mathrm{inc}(\mathrm{Cnt})] \tag{2}$$

Similar formulas express the remaining transitions in Figure 1. The conjunction of these formulas make up the implementation I. A finite model of I, perhaps by instantiating Inc with addition by 1 mod 4, corresponds to the traditional Kripke models of a propositional temporal logic.

The correctness statement for this counter states that, depending on the input, the value of Cnt is eventually updated correctly when the counter reaches the Fetch state again. We cannot state this most naturally in *GTL*, but rather have to give the exact number of transitions required for the counter to reach the fetch state again.

Thus, for example, the statement of correctness for the case when the input is Inc2 is:

$$\Box[\mathrm{state} = \mathrm{Fetch} \wedge \mathrm{input} = \mathrm{Inc2} \supset \circ\circ\circ\, \mathrm{state} = \mathrm{Fetch} \wedge \circ\circ\circ\, \mathrm{Cnt} = \mathrm{inc}(\mathrm{inc}(\mathrm{Cnt}))] \tag{3}$$

The statement of correctness for the counter as a whole would consist of the conjunction of similar formulas for the case of each of the different inputs.

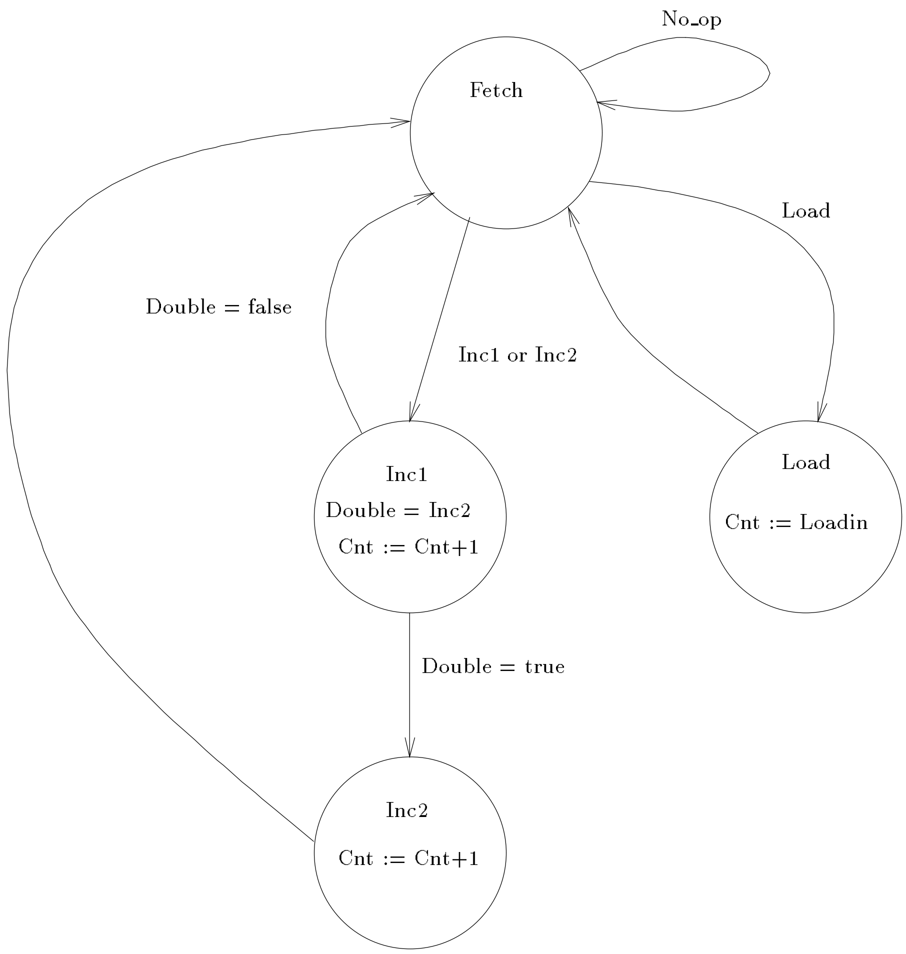


Figure 1: A Counter

This example provides the prototype for our first decidable fragment. I is of the form:

$$\bigwedge_i \square[\text{condition}_i \supset \bigwedge_j \circ\, \text{state\_var}_j = \mathcal{T}_j] \tag{4}$$

where condition_i is a GTL formula that contains no temporal operators and $\mathcal{T}_j$ is a GTL term that contains no instance of $\circ$. Furthermore, the conditions are mutually exclusive, i.e., $i \neq j \supset \neg(\text{condition}_i \wedge \text{condition}_j)$

Mutual exclusivity of the conditions can either be implicitly assumed or guaranteed explicitly by allowing I to include constraints of the form:

$$\bigwedge_i \square[\mathcal{T}_i \neq \mathcal{T}'_i] \tag{5}$$

where $\mathcal{T}_i$ and $\mathcal{T}'_i$ are GTL terms that contain no instance of the $\circ$ operator.

S is of the form:

$$\bigwedge_i \square[\text{condition}_i \supset \bigwedge_k \mathcal{T}_k = \mathcal{T}'_k] \tag{6}$$

where condition_i is a GTL formula that can contain arbitrary number of $\circ$s, but no other temporal operators and $\mathcal{T}_k$ and $\mathcal{T}'_k$ are GTL terms that can contain arbitrary number of $\circ$s.

We define the logic $GTL1$ to be the fragment of GTL that consists only of formulas of the form $I \supset S$ where I is restricted to be of the form described by equations 4 and 5 and S is restricted to be of the form described by equation 6.

The validity problem for $GTL1$ is obviously decidable. We can eliminate $\circ$ from the terms $\mathcal{T}_k$ and $\mathcal{T}'_k$ in 6 by conditional rewriting using the formulae from 4. This reduces the problem to an instance of the validity problem for ground conditional equational logic which is decidable.

Obviously this logic is not very expressive. There are many useful statements even about this simple counter that $GTL1$ cannot express. Without detailing these deficiencies we incrementally expand the expressiveness of $GTL1$ and show that even with very few additions we can achieve a logic that is useful and still decidable.

5 Useful Extensions

We now describe some extensions that make $GTL1$ more useful for hardware verification.

The motivation for the first extension is simply notational convenience. In describing hardware we want to be able to give names to wires in the circuit. To do this we allow I to additionally include formulas of the form:

$$\Box[\mathcal{T} = \mathcal{T}'] \tag{7}$$

where neither $\mathcal{T}$ nor $\mathcal{T}'$ contain any temporal operators.

However, to make the logic truly useful for hardware verification we also need to have a representation for some sort of memory or register file. Thus, we add to our logic the special interpreted symbols, *read* and *write*. These symbols are related by the following axioms:

$$\begin{aligned}&\forall \text{regfile}, \text{addr1}, \text{addr2}, \text{data} : \\ &\quad \text{addr1} = \text{addr2} \supset \text{read}(\text{write}(\text{regfile}, \text{addr1}, \text{data}), \text{addr2}) = \text{data}\end{aligned} \tag{8}$$

$$\begin{aligned}&\forall \text{regfile}, \text{addr1}, \text{addr2}, \text{data} : \\ &\quad \text{addr1} \neq \text{addr2} \supset \text{read}(\text{write}(\text{regfile}, \text{addr1}, \text{data}), \text{addr2}) = \text{read}(\text{regfile}, \text{addr2})\end{aligned} \tag{9}$$

We call this new logic $GTL2$. It is still decidable as methods such as those found in [18] can be used to decide combinations of ground equalities with other decidable theories. We can also add to $GTL2$ further interpreted functions that have decidable quantifier free theories [18].

Just these simple extensions let us state the implementation and correctness for the pipelined ALU in Figure 5. In GTL the correctness of the pipelined ALU becomes:

$$\Box[\neg\text{stall} \supset \text{read}(\circ\circ\circ\,\text{file}, \text{dstn}) = \text{alufun}(\text{op}, \text{read}(\circ\circ\ \text{file}, \text{src1}), \text{read}(\circ\circ\ \text{file}, \text{src2}))] \tag{10}$$

In comparing this to the work done in [8] we note that we can abstract away the size of the data paths and the specific alu function without resorting to logarithms or any other clever means, and yet we do not lose decidability.

This is still a toy example. We are, however, also able to state the correctness of microprocessors in this simple fragment.

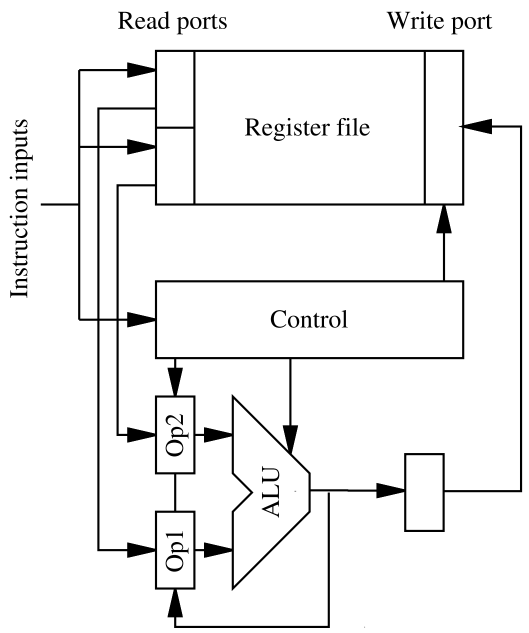


Figure 2: A pipelined ALU

6 Microprocessor Correctness

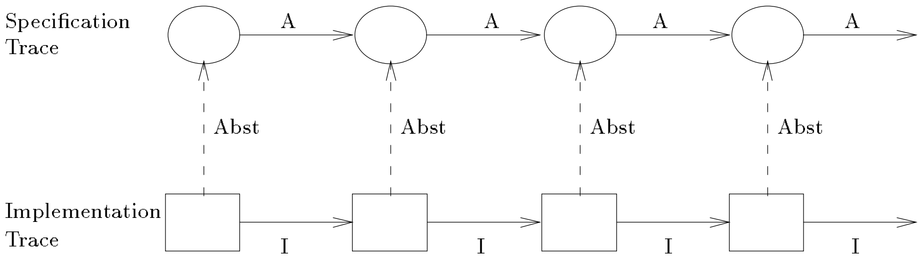
In [20] microprocessor correctness is stated in a form similar to equation 1, where I and S are conditional equations with a universally quantified time variable. In [21, 10] the microprocessor correctness is stated in a form that does not mention time, but rather uses explicit next-state relationships. We now summarize the approach to microprocessor correctness in [21, 10] and show that it can be encoded in a decidable fragment of *GTL*.

Microprocessors can be described as state transition systems. The state of the microprocessor consists of the state of the memory, register file, and internal registers of the processor (these would generally include the program counter, memory address register, and pipeline registers if the processor is pipelined, etc.). The approach taken in verifying microprocessors is to use a simple transition system as the specification of the microprocessor, and a more complex transition system as the implementation. In [20] the specification is actually a non-pipelined microprocessor. In [21] the specification is a transition system corresponding to the instruction set architecture. In both [21, 20] the implementation is a pipelined microprocessor. Early work [12, 13] used the instruction set architecture as the specification and a non-pipelined machine as the implementation.

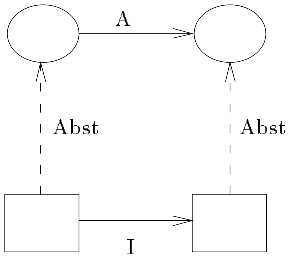
The microprocessor verification problem is to show that the traces induced by the implementation transition system are a *subset* of the traces induced by the specification transition system, where *subset* has to be carefully defined by use of an abstraction mapping. The verification problem is illustrated in Figure 3(a), where I represents the implementation next-state function and A represents the specification next-state function. The details and complications of this approach are beyond the scope of this paper, and the reader is referred to [20, 21, 10, 1] for them.

Following the approach in [1, 21, 20, 16][1] the proof of correctness makes use of an *abstraction* function that maps an implementation state into a *corresponding* specification state. Correctness can then be reduced to showing that for any execution trace of the implementation machine there

[1]The precise details followed in these papers are somewhat different.



(a)



(b)

Figure 3: `Commutes`

exists a *corresponding* execution trace of the specification machine. This is captured in Figure 3(a). The trace equivalence expressed in Figure 3 can be reduced to the commutativity of the diagram in Figure 3(b).

As discussed in [10, 21] the implementation machine may run at a different rate than the specification machine. For example, in the microprocessor described in [20] the specification machine takes one state transition to execute each instruction, but the implementation machine might take five cycles to execute branch instructions, but only one cycle for non-branch instructions. In the following we assume that the specification machine always takes one cycle to execute an instruction. We also assume that the number of cycles that the implementation machine takes to execute an instruction can be given as a function of the current state and current input. (This restriction can be slightly relaxed to deal with interrupts which might arrive a bounded number of cycles into the future.)

Let us denote the function that determines the number of cycles that the implementation machine takes to complete an instruction as `num_cycles`. We assume that it is provided by the hardware designer or verifier.

In [10, 20] this function is given by the following equation:

```
num_cycles =
  IF zero?(alu(IRD4, getOp(program(pc))))
     THEN 5
     ELSE 1
  ENDIF
```

where $\mathtt{IRD4}$ and $\mathtt{pc}$ are registers in the implementation machine, i.e. state variables in GTL.

The first step in verifying the correctness of the microprocessor is to split the proof into cases based on the definition of $\mathtt{num\_cycles}$. Thus for each case we have a precise number of cycles that we have to cycle the implementation machine through. In the following we call this number $\mathtt{nc}$. We denote the conditions under which $\mathtt{num\_cycles} = \mathtt{nc}$ by $\mathtt{cond}_{\mathtt{nc}}$.

In the microprocessor verifications we have looked at, the state variables of the specification state are simply a subset of the state variables of the implementation state. For example, in [20] the specification machine is characterized by the contents of the program counter and register file. The implementation machine is also characterized by the contents of the program counter and register file as well as additional pipeline registers. The abstraction mapping maps each specification register from the corresponding implementation register, but not necessarily from the exactly corresponding state. For example, in [20] the abstraction mapping is given by the following equations:

$$\mathtt{pc}_A = \mathtt{pc}_I \tag{11}$$

$$\mathtt{reg\_file}_A = \circ\circ\circ\,\mathtt{reg\_file}_I. \tag{12}$$

In other words the specification $\mathtt{reg\_file}$ is the implementation $\mathtt{reg\_file}$, but three cycles into the future.

In the following we denote the ith state variable of the specification machine as V_A^i, and the ith state variable of the implementation machine as V_I^i. The abstraction mapping from the specification machine to the implementation machine can then be given by equations of the form:

$$\bigwedge_i (V_A^i = \circ^{a_i}\, V_I^i) \tag{13}$$

where a_i gives the number of lookahead cycles for state variable V^i.

Now, given our assumption that the specification machine takes only one cycle per instruction, we can *symbolically* execute the specification machine to obtain expressions capturing the state of the specification machine state variables after executing one instruction. For each specification machine state variable, V_A^i, we denote its symbolic next state expression as: $N\_V_A^i$.

Now, for each distinct value of $\mathtt{nc}$, we can express the correctness of the microprocessor as:

$$I \supset S \tag{14}$$

where I encodes the transition system of the implementation machine and S is of the form:

$$\Box[\bigwedge_{\mathtt{nc}} \mathtt{cond}_{\mathtt{nc}} \wedge A \supset \bigwedge_i N\_V_A^i = \circ^{\mathtt{nc}}\, V_I^i]. \tag{15}$$

where A encodes the abstraction mapping.

It is an easy exercise to show that equations 14 and 15 are in $GTL2$.

Pipeline Invariants The verification of some pipelined microprocessors requires the use of pipeline invariants. See [21] for example. Such microprocessors cannot be directly verified in $GTL2$ without being first provided the pipeline invariant. However, once the pipeline invariant is provided, the invariance of the pipeline invariant and the correctness of the microprocessor, assuming the pipeline invariant, can be stated in $GTL2$.

Theorem Proving We have carried out our experiments in processor verification in the context of a higher-order theorem prover [19]. In the theorem prover we state the correctness of the microprocessors in a more natural manner than indicated by equations 14 and 15. In the process of the verification we generate as intermediate goals statements that are instances of *GTL*2 formulas. We thus envision making use of logics such as *GTL*2, not just on their own, but as a way to integrate a user directed verification effort based on theorem proving with more automated verification tools such as model checkers or validity checkers for *GTL*2. Investigations concerning *GTL* can be viewed as a principled attempt to determine how much of the verification task can be automated.

7 Conclusions, Related Work, and Future Work

We have presented *GTL*, a new temporal logic that extends propositional temporal logic. We do this by providing an additional temporal operator, $\circ$ that operates on terms. Other temporal logics have also provided notation equivalent to our $\circ\, t$ [17, 15]. To our knowledge we are the first to analyze the complexity of ground temporal logics making use of this operator.

We showed that the full *GTL* is undecidable. We then identified fragments of *GTL* that are both decidable and useful for real hardware verification, including microprocessor verification. These fragments were in part motivated by our experience in using PVS for hardware verification. Using PVS, its ground decision procedures, and a BDD package we have verified the ALU pipeline in 90 seconds. Much, if not most, of this time is spent dealing with the overhead of a general-purpose higher-order theorem prover. There is current work in PVS to build better decision procedures for combining decidable theories. [7] reports on independent work that efficiently decides a fragment of *GTL*2. Efficient decision procedures for *GTL*2 should be able to build upon this work.

In addition to using *GTL* as is, we envision it as a means to integrate decision procedures for a temporal logic into a theorem proving approach. By identifying fragments of *GTL* that are decidable we can identify instances of goals while doing theorem proving that can be directly dispatched. GTL *provides a logical/principled framework for exploring what fragments of hardware verification can be automated.*

In addition to identifying larger decidable fragments of *GTL*, we are currently exploring ways to extend *GTL* to be more expressive. One idea is to define new temporal operators that operate on terms much the same way the $\circ$ does. One possible operator is `atnext(p) t`, which would denote the value of term `t` at the next time instance that formula `p` was true. While not identical, this line of research is similar in spirit to that in [11]. One difference is that in [11] all terms are considered rigid in order to obtain decidability.

To summarize, in *GTL* and its fragments we have identified a practically useful temporal logic, that allows us to overcome some of the disadvantages inherent in both the model-checking and theorem proving approaches to hardware verification.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Third Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE, Computer Society Press, July 1988.

[2] Rajeev Alur. *Techniques for Automatic Verification of Real-time Systems.* Dept. of computer science, Stanford University, 1991.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8), 1986.

[4] R.E. Bryant, D. L. Beatty, and C.-J. Seger. Formal hardware verification by symbolic trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, 1991.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

[6] J. R. Burch, E.M. Clarke, and D.E. Long. Representing circuits more efficiently in symbolic model checking. In *28th ACM/IEEE Design Automation Conference*, 1991.

[7] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *CAV '94*, 1994. Submitted.

[8] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[9] W Cullyer and C Pygott. Hardware proofs using LCF-LSM and ELLA. Memorandum 3832, RSRE, September 1985.

[10] David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, December 1993.

[11] T. Henzinger. Half-order modal logic: How to prove real-time properties. In *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, pages 281–296. ACM Press, 1990.

[12] W.A. Hunt. The mechanical verification of a microprocessor design. In *Proc. of IFIP Working Conference on From H.D.L Descriptions to Guaranteed Correct Circuit Designs*, 1986.

[13] J. Joyce, G. Birtwistle, and M. Gordon. Proving a computer correct in higher order logic. Technical Report 100, Computer Lab., University of Cambridge, 1986.

[14] Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1987.

[15] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Systems Research Center, Palo Alto, California 94301, December 1991.

[16] Paul Loewenstein and David Dill. Verification of multiprocessor cache protocol using simulation relations and higher-order logic. In *Computer-Aided Verification '90*, pages 187–205. DIMACS, American Mathematical Society, 1991.

[17] Z. Manna and A. Pnueli. Verification of concurrent programs: A temporal proof system. In J. W. de Bakker annd J. van Leeuwen, editor, *Foundations of Computer Science IV, Distributed Systems: Part 2*, Mathematical Centre Tracts 159, pages 163–255. Center for Mathematics and Computer Science, Amsterdam, 1983.

[18] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[19] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Lecture Notes in Artifical Intelligence*, pages 748–752. Springer Verlag, June 1992.

[20] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. Technical Report 78, Digital Systems Research Center, Palo Alto, California 94301, September 1991.

[21] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.