Convergence Testing in Term-Level Bounded Model Checking *

Randal E. Bryant[1,2], Shuvendu K. Lahiri[2], and Sanjit A. Seshia[1]

School of Computer Science & Electrical and Computer Engineering Department
Carnegie Mellon University, Pittsburgh, PA 15213
Randy.Bryant@cs.cmu.edu, shuvendu@ece.cmu.edu, Sanjit.Seshia@cs.cmu.edu

Abstract. We consider the problem of bounded model checking of systems expressed in a decidable fragment of first-order logic. While model checking is not guaranteed to terminate for an arbitrary system, it converges for many practical examples, including pipelined processors. We give a new formal definition of convergence that generalizes previously stated criteria. We also give a sound semi-decision procedure to check this criterion based on a translation to quantified separation logic. Preliminary results on simple pipeline processor models are presented.

1 Introduction

Systems with parameters of finite but arbitrary or large size are often modeled as infinite-state systems. Such systems include superscalar processors, communication protocols with unbounded channels, and networks of an arbitrary number of identical processes. While state elements can still be of Boolean type, richer data types such as unbounded integers or unbounded arrays of integers are also used. Employing this richer expressive power is one approach to tackling the state explosion problem.

In the area of hardware verification, the logic of Equality with Uninterpreted Functions and Memories (EUFM) has been successfully used for the automated verification of pipelined processor designs [8, 3]. The more general logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions [4] (CLU) has been used for bounded model checking and inductive invariant checking of out-of-order microprocessors with unbounded resources [14]. Bounded model checking proceeds by symbolically simulating the system for a finite number of steps starting from an initial state, checking on each step that a state property holds. As the state elements can be terms in a first-order logic, we will refer to this technique as *term-level bounded model checking*. Since term-level models can express Turing machines [12], the symbolic simulation might never reach a

* This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029 and by ARO grant DAAD 19-01-1-0485.

fixpoint in general. However, in many practical cases, the simulation does converge. It is therefore necessary to check, after each simulation step, whether the simulation has converged.

In this paper, we make two main contributions. First, we give a formal definition of convergence for term-level bounded model checking, where CLU logic is used as the modeling formalism. The convergence criterion is formulated as a quantified second-order formula with one quantifier alternation and is undecidable in general. Second, we give a semi-decision procedure for this class of second-order formulas. Our procedure is based on a sound translation to a decidable fragment of first-order logic called *quantified separation logic* (QSL). QSL formulas are quantified Boolean combinations of Boolean variables and predicates of the form $x_i < x_j + c$ or $x_i = x_j + c$, where x_i and x_j are real or integer variables, and c is a constant. The QSL formulas are then decided by a translation to quantified Boolean logic [15]. Although we use the semi-decision procedure for convergence checking, our results are also more generally applicable to automated theorem proving of second-order formulas.

Previous term-level model checkers vary in expressiveness of the underlying logic, and either use syntactic convergence criteria or approximation techniques that guarantee convergence at the cost of completeness. Hojati et al. [12] presented a modeling formalism called ICS which is similar in expressiveness to EUFM. They showed that ICS models do not converge in general, except under highly restrictive assumptions that are not of practical interest. Isles et al. [13] built on this work, giving a conservative, syntactic definition of convergence of ICS models, and using it to verify versions of the DLX pipeline. Our logic is more expressive than ICS. Also, as we show in Section 5.2, their convergence criterion is a special case of the one we present in this paper. Corella et al. [9] have used Multiway Decision Graphs (MDGs) for term-level model checking. MDGs are BDD-like data structures used for representing formulas in quantifier-free logics such as EUFM and CLU; the exact logic represented depends on the set of interpreted function symbols used in the model. Thus, Corella et al. use MDGs to represent the characteristic function of the set of states of a term-level model. Unlike our work, their models cannot have variables of function type, and hence cannot verify systems with embedded memories. However, they address a more general class of properties expressible in a first order temporal logic. With respect to convergence checking, Corella et al. use syntactic rewriting techniques similar to those employed for ICS [13]. Bultan et al. [6] have used Presburger arithmetic for verifying concurrent algorithms. Checking convergence for systems expressed in Presburger arithmetic is decidable; however, since the model checking might not converge in general, they conservatively approximate the fixpoint, allowing the possibility of spurious counterexamples. In comparison, our use of CLU logic allows us to use uninterpreted functions and also lets us model richer systems with memories. This expressive power, however, results in convergence checking becoming undecidable.

The rest of the paper is organized as follows. Section 2 presents CLU logic and our system modeling formalism. Section 3 defines the term-level bounded model checking problem. In Section 4, we formally define the convergence criterion. Section 5 describes how we check this criterion. Finally, we conclude in Section 6 with some preliminary results with pipelined processor models. For brevity, we have omitted proofs of theorems and an alternate complete semi-decision procedure; these can be found in an accompanying technical report [5].

2 Preliminaries

2.1 CLU Logic

Syntax. The syntax includes four classes of *expressions*, representing computations of truth values or integers, as well as functions over integers yielding truth values or integers. We use *symbols* to represent abstract values and functions.

$$
\begin{aligned}
\textit{bool-expr} &::= \mathbf{true} \mid \mathbf{false} \mid \textit{bool-symbol} \mid \neg\textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \\
&\quad\mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\
&\quad\mid \textit{predicate-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{int-expr} &::= \textit{lambda-var} \mid \textit{int-symbol} \mid \mathit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
&\quad\mid \textit{int-expr} + \textit{int-constant} \mid \textit{function-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{predicate-expr} &::= \textit{predicate-symbol} \mid \lambda\, \textit{lambda-var}, \ldots, \textit{lambda-var} \,.\, \textit{bool-expr} \\
\textit{function-expr} &::= \textit{function-symbol} \mid \lambda\, \textit{lambda-var}, \ldots, \textit{lambda-var} \,.\, \textit{int-expr}
\end{aligned}
$$

Fig. 1. Expression Syntax. Expressions can denote computations of Boolean values, integers, or functions yielding Boolean values or integers.

Symbols are written with a typewriter font, such as `a` or `f`. Associated with each symbol is a *type* indicating what kind of value it represents (truth, integer, function, or predicate). For function and predicate symbols, the type includes its *arity* indicating the number of arguments it takes. For function symbol `f`, we write its arity as $\mathit{arity}(\mathtt{f})$. For a set of symbols A, we let $E(\mathcal{A})$ denote the set of all expressions that can be formed using these symbols, obeying the usual rules on type matching.

The syntax includes integer *lambda variables*. These only serve to represent the arguments to lambda expressions. Note also that the lambda expression syntax is constrained so that they cannot have functions as arguments, and they cannot express any form of looping or recursion.

Sets of Expressions. We use two ways to refer to sets of expressions in which we must identify the different elements. The first is a *vector notation*, in which we index the elements with integer subscripts. We use the notation $\overline{e_n}$ to denote a vector with elements $e_1, \ldots, e_n$. The second is a *named-element notation*, in which we have a set of symbolic names $\mathcal{A}$ and write a set of expressions e as having an element e_a for each $\mathtt{a} \in \mathcal{A}$.

With both notations, we can indicate the syntactic substitution of elements for symbols or variables in an expression. That is, the expression $s\,[\overline{e_n}/\overline{x_n}]$ denotes the expression where each instance of x_i in s is replaced by the expression e_i for $1 \leq i \leq n$. These substitutions are performed in parallel, so there is no ambiguity of some expression e_i contains the symbol x_j. Similarly, $s\,[\overline{e}/\mathcal{A}]$ indicates the result of replacing each instance of a symbol $\mathtt{a} \in \mathcal{A}$ with the expression $e_{\mathtt{a}}$.

Semantics. For a set of symbols $\mathcal{A}$, we let $\sigma_{\mathcal{A}}$ indicate an *interpretation* of each of these symbols. That is, $\sigma_{\mathcal{A}}$ maps each symbol to an integer, a truth value, or a function according to the symbol type. For any expression $e \in E(\mathcal{A})$, we define its *evaluation under interpretation* $\sigma_{\mathcal{A}}$, denoted $\langle e \rangle_{\sigma_{\mathcal{A}}}$ as the value obtained by evaluating e when each symbol $\mathtt{a}$ is replaced by its interpretation $\sigma_{\mathcal{A}}(\mathtt{a})$. We omit the detailed definition.

A truth expression $e \in E(\mathcal{A})$ is said to be *universally valid* when it evaluates to **true** for all interpretations of its symbols, i.e., when $\langle e \rangle_{\sigma_{\mathcal{A}}} = \mathbf{true}$ for all $\sigma_{\mathcal{A}}$.

As a final notation, for disjoint symbol sets $\mathcal{A}$ and $\mathcal{B}$, each having interpretations $\sigma_{\mathcal{A}}$ and $\sigma_{\mathcal{B}}$, we let $\sigma_{\mathcal{A}} \cdot \sigma_{\mathcal{B}}$ denote the interpretation over the symbols in $\mathcal{A} \cup \mathcal{B}$ obtained by applying the respective interpretations to the symbols in $\mathcal{A}$ and $\mathcal{B}$.

As noted earlier, our syntax for function applications requires all arguments to be integer expressions. We can therefore transform any integer or truth expression containing lambda expressions into an equivalent lambda-free one by performing *Beta reduction*, in which the actual parameter expressions are syntactically substituted in parallel with the actual parameter expressions.

2.2 System Model

We model the system as having a number of *state elements*, where each state element may be a truth or integer value, or a function or predicate. This latter class of state elements allows us to describe various forms of memories. For example, a conventional random-access memory can be modeled as a function that yields an integer data value given an integer address as argument. We use symbolic names to represent the different state elements giving the set of *state symbols* $\mathcal{S}$. We also introduce a set of *input symbols* $\mathcal{T}$, representing a set of input signals that can be set to different values on each step of operation. That is, on each step i, we introduce a symbol $\mathtt{a}_i$ for each input symbol $\mathtt{a}$. We refer to such signals as the *indexed input symbols*. We introduce two more sets of symbols $\mathcal{K}$ and $\mathcal{I}$ to allow one run by the verifier to compute the behavior of systems with different functionality operating with different initial state and input values. The symbols in $\mathcal{K}$ parameterize system functionality. This could include, for example, function symbols for the ALU, and the contents of the instruction memory. The symbols in $\mathcal{I}$ parameterize the initial state and system input sequence. These could include a function symbol to encode the initial state of a memory. They also include the indexed input symbols.

The overall system operation is characterized by an *initial state* s^0 and a *transition behavior* δ. The initial state contains an expression for each state element. The initial value of state element $\mathtt{a}$ is given by an expression $s^0_{\mathtt{a}} \in E(\mathcal{I})$. The transition behavior consists of an expression for each state element. The behavior for state element $\mathtt{a}$ is given by an expression $\delta_{\mathtt{a}} \in E(\mathcal{K} \cup \mathcal{S} \cup \mathcal{T})$. In this expression, we use the state element symbols to represent the current system state, and the input symbols to represent the current values of the inputs. The expression then gives the new state for that state element.

From these expressions, we define the *state sequence* for the system $s^0, \ldots, s^i, \ldots$, where the state at step i consists of an expression for each state element $s^i_{\mathtt{a}} \in E(\mathcal{K} \cup \mathcal{I})$. This expression is given by performing the double substitution

$$s^i_{\mathtt{a}} = \delta_{\mathtt{a}}\left[s^{i-1}/\mathcal{S},\, t^i/\mathcal{T}\right], \tag{1}$$

where the input expression t^i has $t^i_{\mathtt{a}} = \mathtt{a}_i$ for each $\mathtt{a} \in \mathcal{T}$. As mentioned earlier, we always perform Beta reduction following a substitution such as this. We use the shorthand $s^i = \delta(s^{i-1}, t^i)$ to indicate this process of generating the expressions for the state at step i.

3 Property Checking

A *system property* P is represented as a Boolean expression over the state elements $P \in E(\mathcal{S})$. Typically we want to determine whether P holds at some particular step k, or whether P holds at every step. We can determine whether P holds at some particular step k by applying a decision procedure for CLU logic. However, our interest here is to prove that P holds for every step $i \geq 0$. In general, this task is undecidable. The problem remains undecidable even if we restrict the class of systems to ones with only integer state elements, and where the system behavior is described using a logic of equality with uninterpreted functions [12].

Instead, we focus on a more restricted class of systems that satisfy a property we call *k-convergence*. With these systems, every reachable state can be reached within k steps for some combination of initial state and inputs, for some fixed bound k. If we can prove that a system is k-convergent, then we can guarantee property P holds on every step by verifying that it holds on every step up through s^k.

Formally, we say that a system with initial state s^0 and transition behavior δ *converges in k steps*, when for every interpretation $\sigma_{\mathcal{I}}$ of the initial state and inputs and for every interpretation $\sigma_{\mathcal{K}}$ of the system parameters, there exists a step $i \leq k$ and an alternate interpretation $\theta_{\mathcal{I}}$ of the initial state and inputs, such that for every state symbol $\mathtt{a} \in \mathcal{S}$

$$\left\langle s^i_{\mathtt{a}} \right\rangle_{\theta_{\mathcal{I}} \cdot \sigma_{\mathcal{K}}} = \left\langle s^{k+1}_{\mathtt{a}} \right\rangle_{\sigma_{\mathcal{I}} \cdot \sigma_{\mathcal{K}}}. \tag{2}$$

We use the shorthand $\langle s^i \rangle_{\theta_{\mathcal{I}} \cdot \sigma_{\mathcal{K}}} = \langle s^{k+1} \rangle_{\sigma_{\mathcal{I}} \cdot \sigma_{\mathcal{K}}}$ to indicate this equality for every state element. Property (2) states that by step $k+1$, the system will not reach any new states. That is, for every possible interpretation of the system parameters $\theta_{\mathcal{K}}$, and for every possible operation of the system for $k+1$ steps, as determined by the interpretation $\sigma_{\mathcal{I}}$ of the initial state and indexed input symbols $\mathcal{I}$, there is some alternate initial state and input sequence, given by interpretation $\theta_{\mathcal{I}}$ that would have led to the exact state in i steps for some $0 \leq i \leq k$.

We show that this property guarantees that the system will not reach new states beyond step k.

Theorem 1. *If a system converges in k steps, then for any $j \geq 0$ and any interpretation $\sigma_{\mathcal{K}}$ of the system parameters, there exists a step $i \leq k$ and an alternate interpretation $\theta_{\mathcal{I}}$ of the initial state and inputs, such that*

$$\langle s^i \rangle_{\theta_{\mathcal{I}} \cdot \sigma_{\mathcal{K}}} = \langle s^j \rangle_{\sigma_{\mathcal{I}} \cdot \sigma_{\mathcal{K}}} . \tag{3}$$

4 Formulation of the Convergence Criterion

We now reach the main topic of this paper: determining whether a system is k-convergent for some value of k. We can express this as a problem in second-order logic as follows. Introduce a symbol set $\mathcal{J}$ consisting of a symbol $\mathtt{a}'$ for each initial state symbol $\mathtt{a} \in \mathcal{I}$, and a symbol $\mathtt{a}'_i \in \mathcal{I}$ for each indexed input signal $\mathtt{a}_i$, for $1 \leq i \leq k$. Rewrite each state expression s^i, for $0 \leq i \leq k$ to an expression r^i, by replacing each symbol in $\mathcal{I}$ with its counterpart in $\mathcal{J}$.

Using the notation of predicate calculus, we consider the symbols in $\mathcal{I}$, $\mathcal{J}$, and $\mathcal{K}$ to be quantified *variables*, either first-order (for integer or Boolean symbols) or second-order (for function or predicate symbols). We can then write the convergence criterion as:

$$\forall \mathcal{K}\ \forall \mathcal{I}\ \exists \mathcal{J} \left[\bigvee_{0 \leq i \leq k} \ \bigwedge_{\mathtt{a} \in \mathcal{S}} r^i_{\mathtt{a}} = s^{k+1}_{\mathtt{a}} \right] \tag{4}$$

With these quantifiers, we are really quantifying over the possible interpretations of the symbols. Note that this formula cannot be expressed in first-order logic, because we have existentially quantified function symbols.

Example 1. Consider a system with the integer state variables $\mathtt{x}$, $\mathtt{y}$ and Boolean state variable $\mathtt{b}$. The operations are defined by:

$$\begin{array}{lll} \mathsf{init}[\mathtt{x}] = \mathsf{c}_0 & \mathsf{init}[\mathtt{y}] = \mathsf{c}_0 & \mathsf{init}[\mathtt{b}] = \mathbf{true} \\ \mathsf{next}[\mathtt{x}] = \mathtt{f}(\mathtt{x}) & \mathsf{next}[\mathtt{y}] = \mathtt{f}(\mathtt{y}) & \mathsf{next}[\mathtt{b}] = (\mathtt{x} = \mathtt{y}) \end{array}$$

where $\mathtt{c}_0$ is an integer symbol and $\mathtt{f}$ is an uninterpreted function symbol. Using our notation, the sets of symbols are defined as follows — $\mathcal{S} = \{\mathtt{x}, \mathtt{y}, \mathtt{b}\}$, $\mathcal{K} = \{\mathtt{f}\}$, $\mathcal{I} = \{\mathtt{c}_0\}$ and $\mathcal{J} = \{\mathtt{c}'_0\}$.

After simulating the system for one step, the convergence condition (given by equation 4, where $k = 0$) becomes:

$$\forall \mathtt{f}\ \forall \mathtt{c}_0\ \exists \mathtt{c}'_0\ [\mathtt{c}'_0 = \mathtt{f}(\mathtt{c}_0) \wedge \mathtt{c}'_0 = \mathtt{f}(\mathtt{c}_0) \wedge \mathbf{true} = (\mathtt{f}(\mathtt{c}_0) = \mathtt{f}(\mathtt{c}_0))]$$

which simplifies to $\forall \mathtt{f}\ \forall \mathtt{c}_0\ \exists \mathtt{c}'_0\ [\mathtt{c}'_0 = \mathtt{f}(\mathtt{c}_0)]$, which is clearly valid, with $\mathtt{c}'_0$ taking the value $\mathtt{f}(\mathtt{c}_0)$.

Therefore the system converges after one step of simulation. As expected, the state variable $\mathtt{b}$ is always **true** in the reachable set of states.

For a function or predicate state element $\mathtt{F}$, the expression $r^i_{\mathtt{F}} = s^{k+1}_{\mathtt{F}}$ is a *second-order equation*—it states that two functions or predicates are identical for all possible arguments.

For systems without function or predicate state elements, our convergence criterion yields a formula with the quantification structure shown in (4), with only first-order equations. Even for the simple case of a system with one integer symbol in $\mathcal{I}$, one function symbol of arity 2 in $\mathcal{K}$, deciding the truth of a formula with this structure is undecidable [2].

Again we find ourselves facing an undecidable property. We deal with this by 1) using syntactic transformations to eliminate the second-order equations for function and predicate state elements, and 2) using a sound, but incomplete decision procedure for second-order formulas of the form shown in (4). Our procedure is quite simple, but it seems to work well for the formulas arising in our convergence testing.

5 Checking Convergence

5.1 Function and Predicate State Elements

We can convert our convergence formula (4) to one containing only first-order equations by introducing a set of *argument symbols* $\mathcal{Z} = \mathtt{z}_1, \ldots, \mathtt{z}_n$, where n is the maximum arity of any predicate or function state element. Suppose state element $\mathtt{F}$ has arity $\mathit{arity}(\mathtt{F}) = m$. Then define $\tilde{r}^i_{\mathtt{F}} \doteq r^i_{\mathtt{F}}(\mathtt{z}_1, \ldots, \mathtt{z}_m)$, and similarly define $\tilde{s}^i_{\mathtt{F}} \doteq s^i_{\mathtt{F}}(\mathtt{z}_1, \ldots, \mathtt{z}_m)$. Then we can rewrite the convergence criterion as:

$$\forall \mathcal{K}\ \forall \mathcal{I}\ \exists \mathcal{J}\ \forall \mathcal{Z} \left[\bigvee_{0 \le i \le k} \bigwedge_{\mathtt{a} \in \mathcal{S}} \tilde{r}^i_{\mathtt{a}} = \tilde{s}^k_{\mathtt{a}} \right] \tag{5}$$

Unfortunately, we have no general approach to handle formulas with this quantifier structure. Instead, we use rewriting techniques to handle limited forms of function and predicate state elements. Our technique is sufficient to handle random-access memories, including the data memory and register file of a microprocessor.

A random-access memory is modeled as a function state element `Mem` where the argument is an address, and the function returns the value stored at that address. Consider a memory with address input `Adr`, data input `Dat` and write-enable signal `Wrt`. We describe the memory operation in our term-level modeling language as:

$$\begin{aligned}\mathsf{init}[\mathtt{Mem}] &= \mathtt{m_0}\\ \mathsf{next}[\mathtt{Mem}] &= \lambda\, x\, .\, ITE(\mathtt{Wrt} \wedge x = \mathtt{Adr},\, \mathtt{Dat},\, \mathtt{Mem}(x))\end{aligned}$$

where $\mathtt{m_0}$ is an uninterpreted function giving the initial memory contents. Note the restricted class of expressions that will result when modeling the operation of this memory over time to generate the expression $\tilde{r}^i_{\mathtt{Mem}}$. At the base is an uninterpreted function, which can be assigned an interpretation that matches any desired functionality. There will then be a bounded number of updates due to write operations, but these will each be to a single (symbolic) address.

Suppose we wish to determine whether the system has converged for some fixed time point i, so that Equation 5 reduces to

$$\forall \mathcal{K}\; \forall \mathcal{I}\; \exists \mathcal{J}\; \forall \mathcal{Z} \left[\bigwedge_{\mathtt{a} \in \mathcal{S}} \tilde{r}^i_{\mathtt{a}} = \tilde{s}^k_{\mathtt{a}} \right] \tag{6}$$

Then the convergence criterion for state element `Mem` will have the general form:

$$\forall \mathcal{A}\; \exists \mathcal{B}\; \forall \mathtt{z}\; F'(\mathtt{z}) = F(\mathtt{z}) \tag{7}$$

where expression F has only symbols in $\mathcal{A}$, while expression F' has symbols from both $\mathcal{B}$ and $\mathcal{A}$.

We apply a set of rewrites to the symbols in $\mathcal{B}$ and generate a set of verification conditions that guarantees (7) holds, based on the structure of expression F'. In general, our rules apply to equations of the form $P(\mathtt{z}) \implies F'(\mathtt{z}) = F(\mathtt{z})$, where P is a predicate expression with symbols from both $\mathcal{B}$ and $\mathcal{A}$. At the top level, we start with P being an expression that always yields **true**.

1. For equations of the form $P(\mathtt{z}) \implies \mathtt{f}'(\mathtt{z}) = F(\mathtt{z})$, where $\mathtt{f}'$ is a function symbol in $\mathcal{B}$, rewrite all occurrences of $\mathtt{f}'$ in $\tilde{r}^i$ to be $\lambda\, x . ITE(P(x),\, F(x),\, \mathtt{f}'(x))$.
2. For equations of the form $P(\mathtt{z}) \wedge \mathtt{z} = E \implies F'(\mathtt{z}) = F(\mathtt{z})$, where E is an expression with symbols from both $\mathcal{B}$ and $\mathcal{A}$, reduce the equation to $P(E) \implies F'(E) = F(E)$. This eliminates any reference to $\mathtt{z}$ in the equation.

3. For equations of the form $P(\mathtt{z}) \implies [\lambda\, x \,.\, \mathit{ITE}(Q(x),\, G'(x),\, H'(x))]\,(\mathtt{z}) = F(\mathtt{z})$, where Q, G', and H' are predicate and function expressions containing symbols in both $\mathcal{A}$ and $\mathcal{B}$, we generate two verification conditions: $P(\mathtt{z}) \wedge Q(\mathtt{z}) \implies G'(\mathtt{z}) = F(\mathtt{z})$, and $P(\mathtt{z}) \wedge \neg Q(\mathtt{z}) \implies H'(\mathtt{z}) = F(\mathtt{z})$, and solve these recursively.
4. For equations of the form $P(\mathtt{z}) \implies \mathtt{f}(\mathtt{z}) = F(\mathtt{z})$, where $\mathtt{f}$ is a function symbol in $\mathcal{A}$, we recursively analyze the structure of F.
   - If F is of the form $\mathit{ITE}(Q(x),\, G(x),\, H(x))$, where Q, G, and H are predicate and function expressions containing symbols in $\mathcal{A}$, we generate two verification conditions: $P(\mathtt{z}) \wedge Q(\mathtt{z}) \implies \mathtt{f}(\mathtt{z}) = G(\mathtt{z})$, and $P(\mathtt{z}) \wedge \neg Q(\mathtt{z}) \implies \mathtt{f}(\mathtt{z}) = H(\mathtt{z})$, and solve these recursively.
   - If F is of the form $\mathtt{g}(\mathtt{z})$, then the symbols $\mathtt{f}$ and $\mathtt{g}$ need to be the same. If the two symbols are different, we return **false** which implies that no rewrite exists.
5. For equations of the form $P(\mathtt{z}) \implies F'(\mathtt{z}+c) = F(\mathtt{z})$ with integer constant c, transform the equation to be $P(\mathtt{z}-c) \implies F'(\mathtt{z}) = F(\mathtt{z}-c)$, and solve it recursively.

Similar rules hold for equations of the form $P \implies F'(\mathtt{z}) = F(\mathtt{z})$, i.e., P is a Boolean expression independent of $\mathtt{z}$.

Given the special form of the expressions describing the updating of a random-access memory, we can see that by repeated application of these rules, we can eliminate all occurrences of symbol $\mathtt{z}$ in (6). The first rule handles the uninterpreted function representing the initial memory state. The second rule handles updates to individual memory addresses. The third rule lets us split based on the case structure of the expression. The last two rules would be required for more complex memory structures.

Note that CLU logic can be used to model memories in which multiple entries can be updated in parallel [14]. The rewriting techniques proposed in this section do not work for such memories.

5.2 Convergence with First-Order Equations

Assume we have applied transformation rules to eliminate all second-order equations, and hence the convergence criterion is expressed by an equation of the form shown in (4) with only first-order equations. We would therefore like to decide the validity of a formula ψ of the form

$$\psi \doteq \forall \mathcal{A}\; \exists \mathcal{B}\; \phi \tag{8}$$

where ϕ does not contain any quantifiers. In fact, ϕ is a CLU formula, and we can assume that transformations have been applied to eliminate all ITE operations[1] and lambda applications.

[1] These can be eliminated by the "push to the leaves" transformation [16].

Our system model is sufficiently general that we can generate any second-order formula having the structure shown in (8) as part of a convergence test. To see this, let the variables in ϕ be $\mathcal{A} = \overline{a_n}$ and $\mathcal{B} = \overline{b_m}$. Introduce a set of $m+1$ state elements, consisting of an element $\mathtt{q}_i$ for each existentially quantified variable $b_i \in \mathcal{B}$, and a final truth-valued state element $\mathtt{q}_{m+1}$. For each universally quantified variable $a_i \in \mathcal{A}$, introduce a system parameter $\mathtt{a}_i$. Let the system have transition behavior δ such that $\delta_{\mathtt{q}_{n+1}} \doteq \phi\left[\overline{\mathtt{q}_m}/\overline{b_m}, \overline{\mathtt{a}_n}/\overline{a_n}\right]$, and $\delta_{\mathtt{q}_i} = \mathtt{q}_i$ for $1 \leq i \leq m$. Finally, let the initial state $s^0_{\mathtt{q}_i}$ of each state element $\mathtt{q}_i$ for $1 \leq i \leq m$ be a_i, and the initial state of $\mathtt{q}_{m+1}$ be **true**. Then the system is 0-convergent if and only if the formula $\forall \mathcal{A}\ \exists \mathcal{B}\ \phi$ is valid.

This construction shows that we cannot assume any particular restrictions on the formulas we must decide to prove convergence, other than the quantifier structure shown in (8).

Syntactic Approach. Previous approaches to convergence have been based on finding syntactic similarities between the earlier state r^i and the current state s^{k+1}. The convergence criterion given by Isles et al. [13] is a more conservative check than the criterion we give in Equation 5, and hence is less general. We can see that their syntactic substitution-based technique is simply a strategy for proving the validity of a formula with the structure shown in (8) as follows.

Proposition 1. *Let b denote a set containing an expression $b_{\mathtt{a}} \in E(\mathcal{A})$ for each $\mathtt{a} \in \mathcal{B}$. If $\forall \mathcal{A}\ \phi\,[b/\mathcal{B}]$ is valid, then so is $\forall \mathcal{A}\ \exists \mathcal{B}\ \phi$.*

The proof of this proposition follows by instantiating any symbol $\mathtt{a} \in \mathcal{B}$ with the value $\langle b_{\mathtt{a}} \rangle_{\sigma_{\mathcal{A}}}$.

With this approach, we can prove convergence by using a decision procedure for CLU logic to prove the universal validity of $\phi\,[b/\mathcal{B}]$. The challenge, of course, is to find an appropriate set of substitutions to the symbols in $\mathcal{B}$.

Semantic Approach. We describe a way to transform formulas of the structure $\psi \doteq \forall \mathcal{A}\ \exists \mathcal{B}\ \phi$ into a formula in the logic we call *Quantified Separation Logic* (QSL). QSL consists of quantified Boolean and integer variables, Boolean connectives, and predicates of the form $\mathtt{x} = \mathtt{y} + c$ and $\mathtt{x} < \mathtt{y} + c$, where $\mathtt{x}$ and $\mathtt{y}$ are integer variables, and c is an integer constant. Our translation $T_s(\psi)$ (for "sound") yields a formula that is valid only if ψ is valid. By deciding the validity of the translation we can test for definite convergence.

We can rewrite any Boolean or integer expression in CLU into a *normal form*, in which all *ITE* operations have been eliminated, and the additions of integer constants are grouped together. Define an *atomic expression* as either an integer or Boolean symbol, or an application of a function or predicate symbol.

Without loss of generality, let us assume ϕ is in normal form. We start by enumerating all of the atomic expressions occurring in ϕ as a sequence $g_1, \ldots, g_n$.

Let $top(g_i)$ denote the top-level symbol in subexpression g_i. We can see that each atomic expression g_i must be of one of the following forms:

1. Boolean symbol. $g_i \doteq \mathtt{b}$, giving $top(g_i) = \mathtt{b}$.
2. Predicate application. $g_i \doteq \mathtt{p}(g_{i_1} + c_{i,1}, \ldots, g_{i_k} + c_{i,k})$, giving $top(g_i) = \mathtt{p}$.
3. Integer symbol. $g_i \doteq \mathtt{x}$, giving $top(g_i) = \mathtt{x}$.
4. Function application. $g_i \doteq \mathtt{f}(g_{i_1} + c_{i,1}, \ldots, g_{i_k} + c_{i,k})$, giving $top(g_i) = \mathtt{f}$.

We require the sequence to be ordered according to subexpression containment. That is, for the function and predicate application forms listed above, we require $i_l < i$ for $1 \leq l \leq k$. The soundness property of translation T_s holds for any such ordering, but we get a tighter bound by listing the subexpressions having top-level symbols in $\mathcal{A}$ as early as possible. That is, if $top(g_i) \in \mathcal{A}$ and $top(g_j) \in \mathcal{B}$, then $i < j$, unless g_j is a subexpression of g_i.

Now introduce a sequence of symbols $\overline{\mathtt{v}_n} \doteq \mathtt{v}_1, \ldots, \mathtt{v}_n$, where $\mathtt{v}_i$ is an integer (respectively, Boolean) symbol when $top(g_i)$ is an integer or function symbol (respectively., Boolean or predicate symbol). We generate two formulas $C_{\mathcal{A}}$ and $C_{\mathcal{B}}$, each of which is a conjunction of consistency constraints by considering each pair of subexpressions g_i and g_j, with $i < j$ and $top(g_i) = top(g_j)$. These are the same constraints used by Ackermann for removing function applications from a formula [1]. For subexpression g_i of the form $\mathtt{f}(g_{i_1} + c_{i,1}, \ldots, g_{i_k} + c_{i,k})$, and g_j of the form $\mathtt{f}(g_{j_1} + c_{j,1}, \ldots, g_{j_k} + c_{j,k})$, we include the constraint

$$\mathtt{v}_{i_1} = \mathtt{v}_{j_1} + (c_{j,1} - c_{i,1}) \wedge \cdots \wedge \mathtt{v}_{i_k} = \mathtt{v}_{j_k} + (c_{j,k} - c_{i,k}) \implies \mathtt{v}_i = \mathtt{v}_j \tag{9}$$

This constraint is included in either $C_{\mathcal{A}}$ or $C_{\mathcal{B}}$ according to whether $\mathtt{f} \in \mathcal{A}$ or $\mathtt{f} \in \mathcal{B}$. Similar constraints are generated when the top-level symbol in g_i and g_j is a predicate symbol $\mathtt{p}$.

Let $\hat{\phi}$ be the formula generated by replacing each atomic expression g_i in ϕ with the symbol $\mathtt{v}_i$. We always replace maximal subexpressions, so that the resulting formula no longer contains any symbols from ϕ.

Let quantifier Q_i be $\forall$ when $top(g_i) \in \mathcal{A}$, and $\exists$ when $top(g_i) \in \mathcal{B}$.

The soundness-preserving translation of ψ is given by

$$T_s(\psi) \doteq Q_1 \mathtt{v}_1\ Q_2 \mathtt{v}_2\ \cdots\ Q_n \mathtt{v}_n\ \left[C_{\mathcal{A}} \implies (C_{\mathcal{B}} \wedge \hat{\phi})\right] \tag{10}$$

Theorem 2. *For any formula ψ having the structure $\psi \doteq \forall\mathcal{A}\ \exists\mathcal{B}\ \phi$, if $T_s(\psi)$, as given by (10), is valid, then so is ψ.*

We also provide a completeness preserving translation in [5]. We can test for possible convergence by deciding the validity of this translation.

We now give some examples to demonstrate the capabilities and limitations of our translation method.

Example 2. Our first example is a case where we successfully prove soundness.

$$\forall \mathtt{f}, \mathtt{y}\ [\forall \mathtt{x}\ \mathtt{x} = \mathtt{f}(\mathtt{x})] \implies \mathtt{y} = \mathtt{f}(\mathtt{f}(\mathtt{y})) \tag{11}$$

To get this into the required form, we rewrite it as

$$\forall \mathtt{f}, \mathtt{y}\ \exists \mathtt{x}\ [\neg(\mathtt{x} = \mathtt{f}(\mathtt{x})) \vee \mathtt{y} = \mathtt{f}(\mathtt{f}(\mathtt{y}))]$$

We write the subexpressions as follows. To make the resulting formulas more readable, we introduce symbols with names based on the subexpressions, rather than the more generic $\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_n$:

Subexpression	g_1 $\mathtt{y}$	g_2 $\mathtt{f}(\mathtt{y})$	g_3 $\mathtt{f}(\mathtt{f}(\mathtt{y}))$	g_4 $\mathtt{x}$	g_5 $\mathtt{f}(\mathtt{x})$
Symbol	$\mathtt{y}$	$\mathtt{fy}$	$\mathtt{ffy}$	$\mathtt{x}$	$\mathtt{fx}$

For $C_{\mathcal{A}}$ we then get

$$(\mathtt{x} = \mathtt{y} \implies \mathtt{fx} = \mathtt{fy}) \wedge (\mathtt{x} = \mathtt{fy} \implies \mathtt{fx} = \mathtt{ffy}) \wedge (\mathtt{y} = \mathtt{fy} \implies \mathtt{fy} = \mathtt{ffy})$$

For formula $C_{\mathcal{B}}$ we get **true**, while for $\hat{\phi}$ we get $\neg(\mathtt{x} = \mathtt{fx}) \vee \mathtt{y} = \mathtt{ffy}$, and the overall quantifier structure is:

$$\forall \mathtt{y}\ \forall \mathtt{fy}\ \forall \mathtt{ffy}\ \exists \mathtt{x}\ \forall \mathtt{fx}$$

It can be easily shown that the QSL formula is valid. We omit the details.

Example 3. Our second example illustrates a case where the formula is valid, but the soundness-preserving transformation fails to show this.

$$\forall \mathtt{f}\ [\forall \mathtt{x}\ \mathtt{f}(\mathtt{x}) < \mathtt{f}(\mathtt{x}+1)] \implies [\forall \mathtt{y}\ \mathtt{f}(\mathtt{y}) < \mathtt{f}(\mathtt{y}+2)] \tag{12}$$

To get this into the required form, we rewrite it as

$$\forall \mathtt{f}\ \forall \mathtt{y}\ \exists \mathtt{x}\ \neg(\mathtt{f}(\mathtt{x}) < \mathtt{f}(\mathtt{x+1}))\ \vee\ \mathtt{f}(\mathtt{y}) < \mathtt{f}(\mathtt{y}+2)$$

We write the subexpressions as follows.

Subexpression	g_1 $\mathtt{y}$	g_2 $\mathtt{f}(\mathtt{y})$	g_3 $\mathtt{f}(\mathtt{y}+2)$	g_4 $\mathtt{x}$	g_5 $\mathtt{f}(\mathtt{x})$	g_6 $\mathtt{f}(\mathtt{x}+1)$
Symbol	$\mathtt{y}$	$\mathtt{fy}$	$\mathtt{fy2}$	$\mathtt{x}$	$\mathtt{fx}$	$\mathtt{fx1}$

For $C_{\mathcal{A}}$ we then get

$$\begin{gathered}(\mathtt{x} = \mathtt{y} \implies \mathtt{fx} = \mathtt{fy}) \wedge (\mathtt{x} = \mathtt{y} - 1 \implies \mathtt{fx1} = \mathtt{fy}) \\ \wedge (\mathtt{x} = \mathtt{y} + 2 \implies \mathtt{fx} = \mathtt{fy2}) \wedge (\mathtt{x} = \mathtt{y} + 1 \implies \mathtt{fx1} = \mathtt{fy2})\end{gathered}$$

For formula $C_{\mathcal{B}}$ we get **true**, while for $\hat{\phi}$ we get

$$\neg(\mathtt{fx} < \mathtt{fx1}) \vee \mathtt{fy} < \mathtt{fy2}$$

and the overall quantifier structure is:

$$\forall \mathtt{y}\ \forall \mathtt{fy}\ \forall \mathtt{fy2}\ \exists \mathtt{x}\ \forall \mathtt{fx}\ \forall \mathtt{fx1}$$

This formula is not valid.

This example shows the limited capability of our translation T_s. It does not do the multiple instantiations of x required to replace the quantified antecedent in (12) with $\mathtt{f(y)} < \mathtt{f(y+1)}\ \wedge\ \mathtt{f(y+1)} < \mathtt{f(y+2)}$.

6 Results & Discussion

We have implemented a prototype of the convergence testing framework within the UCLID [4] verification tool. Currently, we have only implemented the soundness-preserving translation to QSL. The QSL solvers use different techniques to transform a QSL formula to a quantified Boolean formula (QBF) [15]. All the experiments are performed on a 2GHz Pentium-4 running Linux, with 1 GB of memory.

In this section, we describe our experience with the convergence testing framework for a three-stage arithmetic pipeline given in figure 2. This example originated with the first work on symbolic model checking [7], and has subsequently become a standard for verification research [10, 13]. In our version, we make use of both stalling and forwarding to resolve read-after-write hazards in the pipeline. Previous versions used only forwarding, with the result that a new result is written to the register file on each step of operation.

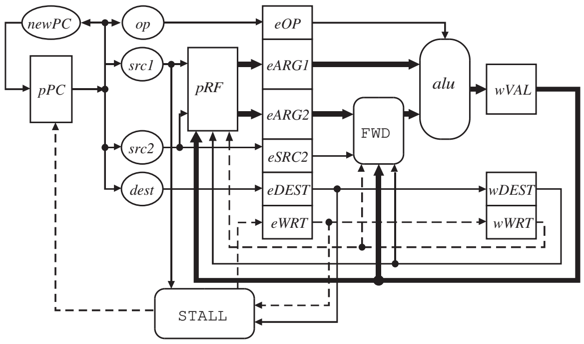

Fig. 2. Pipelined Version of ALU Circuit. The three stages of the pipeline: fetch, execute and write-back. Read-after-write hazards are resolved for the first operand by stalling and for the second by forwarding. The dashed lines indicate Boolean control and the solid lines represent the flow of integer values.

The state elements of the pipeline include a function state variable, an unbounded register file *pRF*. The integer state elements include the different register identifiers, namely *eSRC2*, *eDEST* and *wDEST*, the data values *eARG1*, *eARG2* and *wVAL*, and the program counter *pPC*. The Boolean state elements consist of the write enable registers *eWRT* and *wWRT*. The system functionality is parameterized by uninterpreted function symbols for decoding an instruction, updating the program counter and the ALU. The Boolean state elements are initialized to **false** and the rest of the state elements take on arbitrary initial values.

The pipeline was symbolically simulated starting from the initial state. The QSL formula produced by the soundness preserving translation was **false** after $k = 1$ and $k = 2$ steps of simulation. A look at the Boolean state elements indicated that the system indeed does not converge within two steps. However, after $k = 3$ steps of simulation, the QSL formula produced was too large to be solved with the current QSL solver implementation we use [15]. The formula had 53 quantified integer variables, with 6 levels of quantifier alternations, 836 nodes in a Directed Acyclic Graph (DAG) representation of the formula, and the BDD representing the QBF formula exceeds 1 GB of memory. However, we have been able to prove the convergence of two simplified versions of the pipeline processor.

1. For the first case, we removed the data-path components of the processor including the register file, operand values and the write-back value. The remaining pipeline still contains the entire control complexity of the original pipeline including the stalling and the forwarding mechanisms. This model converges after $k = 3$ steps of simulation and our decision procedure detects so within 2 seconds with less than 11 MB of memory. The QSL formula contains 27 quantified integer variables, with 4 levels of quantifier alternations and 249 nodes in the DAG form. Notice that this example contains uninterpreted function symbols but does not contain any function state elements.
2. For the second case, we combined the execute and the write-back stages of the pipeline into a single stage (making the pipeline 2-stage), but retained the register file *pRF* and the data-path. The pipeline was modified to accommodate both stalling and forwarding of data. This example converges after $k = 2$ steps of simulation and our decision procedure takes 8 seconds to prove it valid. The memory consumption was about 80 MB. The QSL formula contains 29 quantified integer variables, with 4 levels of quantifier alternations and 203 nodes in the DAG form.

We are currently working on alternate translations of QSL formulas to QBF formulas and hope to test the convergence of the pipeline with a few optimizations. We are also experimenting with enumeration based QBF solvers including Quaffle [17].

Discussion. The notion of k-convergence is not useful for systems with unbounded buffers, since many such systems do not converge. Moreover, our preliminary results indicate that the convergence criterion we present is precise, but computationally difficult to check. Abstraction techniques, such as predicate abstraction [11], allow for greater efficiency at the expense of using an approximate notion of convergence, and are a promising area for future work.

References

1. W. Ackermann. *Solvable Cases of the Decision Problem.* North-Holland, Amsterdam, 1954.
2. Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem.* Springer-Verlag, 1997.
3. R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.
4. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.
5. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Convergence testing in term-level bounded model checking. Technical Report CMU-CS-03-156, Carnegie Mellon University, 2003.
6. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *Computer-Aided Verification (CAV '97)*, LNCS 1254. Springer-Verlag, June 1997.
7. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conference*, 1991.
8. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80. June 1994.
9. Francisco Corella, Z. Zhou, Xiaoyu Song, Michel Langevin, and Eduard Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
10. D. Cyrluk and P. Narendran. Ground temporal logic: a logic for hardware verification. In *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 247–259. June 1994.
11. S. Graf and H. Saidi. Construction of abstract state graphs using PVS. In *Proc. Intl. Conference on Computer-Aided Verification, (CAV'97)*, LNCS 1254, 1997.
12. R. Hojati, A. Isles, D. Kirkpatrick, and R. K. Brayton. Verification using finite instantiations and uninterpreted functions. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, pages 218–232. November 1996.
13. A. J. Isles, R. Hojati, and R. K. Brayton. Computing reachable control states of systems modeled with uninterpreted functions and infinite memory. In *Computer-Aided Verification (CAV '98)*, LNCS 1427, pages 256–267. June 1998.
14. Shuvendu K. Lahiri and Randal E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proc. Intl. Conference on Computer-Aided Verification, (CAV'03)*, LNCS 2725, pages 341–354, 2003.
15. Sanjit A. Seshia and Randal E. Bryant. Unbounded, fully symbolic model checking of timed automata using Boolean methods. In *Proc. Intl. Conference on Computer-Aided Verification, (CAV'03)*, LNCS 2725, pages 154–166, 2003.
16. M. N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *38th Design Automation Conference (DAC '01)*, June 2001.
17. Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *Principles and Practice of Constraint Programming (CP '02)*, LNCS 2470, pages 200–215. Springer, 2002.