Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions

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

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
{Randy.Bryant, Sanjit.Seshia}@cs.cmu.edu
[2] Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA
shuvendu@ece.cmu.edu

Abstract. In this paper, we present the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU). CLU generalizes the logic of equality with uninterpreted functions (EUF) with constrained lambda expressions, ordering, and successor and predecessor functions. In addition to modeling pipelined processors that EUF has proved useful for, CLU can be used to model many infinite-state systems including those with infinite memories, finite and infinite queues including lossy channels, and networks of identical processes. Even with this richer expressive power, the validity of a CLU formula can be efficiently decided by translating it to a propositional formula, and then using Boolean methods to check validity. We give theoretical and empirical evidence for the efficiency of our decision procedure. We also describe verification techniques that we have used on a variety of systems, including an out-of-order execution unit and the load-store unit of an industrial microprocessor.

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. Modeling and verification methods for these systems must trade off between the expressiveness of the modeling formalism and the efficiency and automation of the tool. Tools based on very general logics can express a variety of systems but require greater human assistance.

To verify pipelined processors, Burch and Dill presented a logic of equality with uninterpreted functions (EUF) [10], and then added interpreted operations *read* and *write* to model unbounded, random-access memories. EUF thus allows for abstract modeling of both data and data operations, as well as unbounded memories. In previous work, we presented PEUF, a logic of positive equality with uninterpreted functions [7]. PEUF has the same expressive power as EUF, but allows for a more efficient decision procedure based on Boolean methods. The main source of efficiency is a technique for transforming a PEUF formula into a propositional formula whose validity can be checked using either BDDs or a satisfiability solver. The advantages of using PEUF have been demonstrated in reasoning about pipelined processors [22].

In this paper, we continue our research into logics that have an efficient transformation into propositional logic. We generalize EUF to yield a logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU). The generalizations are of two kinds. The first is to include a restricted class of lambda expressions as a means of defining state variables that are functions or predicates. As we will discuss, this generalization subsumes the need for special *read* and *write* operations. The second is to introduce ordering and a highly restricted fragment of Peano arithmetic we call *counter arithmetic*. We do this by use of the interpreted predicate symbol "$<$" and interpreted function symbols **succ** (the successor function) and **pred** (the predecessor function). As with EUF, we consider only a quantifier-free subset of first-order logic. However, our generalizations give us richer expressiveness in modeling both data and control.

We make two main contributions in this paper. First, we demonstrate the expressiveness of CLU by modeling constructs found in several infinite-state systems, including processors, communication protocols, and unbounded process arrays. Second, we describe our decision procedure for CLU that retains the efficiencies of the decision procedure for PEUF. We give theoretical and empirical evidence for our procedure's efficiency, comparing it with the Stanford Validity Checker (SVC) [3]. We have built UCLID[3], a tool in which systems modeled using CLU can be specified and checked for safety properties and have applied it to a variety of systems including an out-of-order processor core, pipelined processors, a complex load-store unit from an industrial microprocessor, a cache coherence protocol, and the Alternating Bit Protocol. Our analysis of examples is more general than that possible by many traditional model-checking approaches in that we can handle arbitrary-size data structures and infinite data types without abstracting them away.

Related Work. A range of specification and verification methods currently exist for infinite-state systems. However, most of these methods are specialized for classes of problems. For example, for communication protocols, existing queue representations include QDDs [5] and regular expressions [1]. Regular expressions have also been used to model networks of identical processes and systems operating on unbounded data structures such as stacks [15, 6]. While regular expressions are good for modeling control based on the form of process arrays or data structures, they cannot be used to model data and operations on data. In contrast, CLU can be used to model both data and control. The applicability of QDDs is restricted to modeling queues. Bultan et al. [9] have used Presburger

[3] UCLID stands for "Uninterpreted functions, Counter arithmetic and Lambda expressions for Infinite Domains"

arithmetic to model process networks. Presburger arithmetic is a very powerful as it allows quantification and integer addition. However it suffers two drawbacks: First, the worst-case complexity of checking validity of formulas in this logic is prohibitively high [11], and second, adding uninterpreted functions to the logic makes it undecidable. The theory of *separation predicates* [19] differs from CLU in that it has neither uninterpreted functions nor lambda expressions, but can have real valued variables. Our work complements techniques for deciding this logic (e.g., [4, 20]) by adding the benefits of positive equality. Theorem proving systems (e.g., PVS [17] or HOL [13]) that use higher order logic can clearly express all the systems that CLU can, but at the cost of reduced automation and efficiency. Compositional model checking [14] can verify both safety and liveness properties and is effective when the system can be easily decomposed into components based on modularity, temporal separation, or if a "unit of work" uses a small finite amount of resources, but it still suffers from state explosion. The role played by lambda expressions in our logic is very similar to that played by state variables of infinite-length array-type in Cadence SMV.

The rest of the paper is organized as follows. In the next two sections, we present the syntax and semantics of CLU, and show how it can be used to model various systems. We next discuss our decision procedure for CLU, and describe UCLID, the verification tool we have built. Finally, we present results demonstrating the efficiency of our decision procedure, and conclusions.

2 Counter arithmetic with Lambda expressions and Uninterpreted functions

Expressions in CLU describe a means of computing four different types of values. *Boolean* expressions yield **true** or **false**. We also refer to Boolean expressions as *formulas*. *Integer* expressions, also referred to as *terms*, yield integer values. *Predicate* expressions denote functions from integers to Boolean values. *Function* expressions, on the other hand, denote functions from integers to integers. Figure 1 summarizes the expression syntax.

$$
\begin{array}{rcl}
\textit{bool-expr} & ::= & \mathbf{true} \mid \mathbf{false} \mid \neg \textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \mid (\textit{bool-expr} \vee \textit{bool-expr}) \\
& & \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\
& & \mid \textit{predicate-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{int-expr} & ::= & \textit{int-var} \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
& & \mid \mathbf{succ}(\textit{int-expr}) \mid \mathbf{pred}(\textit{int-expr}) \mid \textit{function-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{predicate-expr} & ::= & \textit{predicate-symbol} \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \textit{bool-expr} \\
\textit{function-expr} & ::= & \textit{function-symbol} \mid \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \textit{int-expr}
\end{array}
$$

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

The simplest truth expressions are the values **true** and **false**. Boolean expressions can also be formed by comparing two term expressions for equality (referred to as an *equation*) or for ordering (referred to as an *inequality*), by applying a predicate expression to a list of term expressions, and by combining Boolean expressions using Boolean connectives. Integer expressions can be integer variables, used only as the formal arguments of lambda expressions. They can also be formed by applying a function expression (including interpreted functions **succ** and **pred**) to a set of integer expressions, or by applying the *ITE* (for "if-then-else") operator. The *ITE* operator chooses between two values based on a Boolean control value, i.e., $ITE(\mathbf{true}, x_1, x_2)$ yields x_1 while $ITE(\mathbf{false}, x_1, x_2)$ yields x_2. Function expressions can be either function symbols, representing uninterpreted functions, or lambda expressions, defining the value of the function as an integer expression containing references to a set of argument variables. Function symbols of arity 0 are also called *symbolic constants*. They denote arbitrary integer values. Since these symbols are instantiated without any arguments, we will omit the parentheses, writing a instead of $a()$. Similarly, predicate expressions can be either predicate symbols, representing uninterpreted predicates, or lambda expressions, defining the value of the predicate as a Boolean expression containing references to a set of argument variables. Predicate symbols of arity 0 are also called *symbolic Boolean constants*. They denote arbitrary Boolean values. We will also omit the parentheses following the instantiation of such a predicate.

Notice that we restrict the parameters to a lambda expression to be integers, and not function or predicate expressions. There is no way in our logic to express any form of iteration or recursion. The lambda expressions in CLU are very useful for modeling, as we show in Section 3, but, in the theoretical sense, they do not add expressive power to the logic.

An integer variable x is said to be *bound* in expression E when it occurs inside a lambda expression for which x is one of the argument variables. We say that an expression is *well-formed* when it contains no unbound variables. The value denoted by a well-formed expression in CLU is defined relative to an interpretation I of the function and predicate symbols. Let $\mathcal{Z}$ denote the set of integers. Interpretation I assigns to each function symbol of arity k a function from $\mathcal{Z}^k$ to $\mathcal{Z}$, and to each predicate symbol of arity k a function from $\mathcal{Z}^k$ to $\{\mathbf{true}, \mathbf{false}\}$. Given an interpretation I of the function and predicate symbols and a well-formed expression E, we can define the *valuation* of E under I, denoted $[E]_I$, according to its syntactic structure. The valuation of E is either a Boolean value, an integer, a function from integers to Boolean values, or a function from integers to integers, according to whether E is a Boolean expression, an integer expression, a predicate expression, or a function expression, respectively. We omit the details. A well-formed formula F is *true under interpretation* I if $[F]_I$ is **true**. It is *valid* when it is true under all possible interpretations.

In earlier work [7], it was shown that formulas in PEUF can be efficiently decided by only considering *maximally diverse interpretations*. We will show in Section 4 how the benefits of PEUF are retained to yield an efficient decision procedure for CLU.

3 System Modeling

In this section, we give representative examples of structures modeled using CLU. We use a record notation to represent data structures that are characterized by multiple CLU expressions.

3.1 Memories

Lambda notation allows us to model the effect of a sequence of read and write operations on a memory. At any point of system operation, a memory is represented by a function expression M denoting a mapping from addresses to values. The initial state of the memory is given by an uninterpreted function symbol m_0 indicating an arbitrary memory state. The effect of a write operation with integer expressions A and D denoting the address and data values yields a function expression M':

$$M' = \lambda\, addr \,.\, ITE(addr = A,\, D,\, M(addr))$$

Other forms of memory can be modeled as well. For example, we can model a Content Addressable Memory (CAM) that stores associations between keys and data. We represent a CAM C at any point in the system operation by two expressions: a predicate expression $C.present$ such that $C.present(k)$ is true for any key k that is stored in the CAM, and a function expression $C.data$, such that $C.data(k)$ yields the data associated with key k, assuming the key is present. As an initial state in invariant checking we can represent a CAM C having an arbitrary state by letting $C.present = p_0$ and $C.contents = c_0$, where p_0 (respectively, c_0) is an uninterpreted predicate (resp., function).

Insertion into a CAM is expressed by the operation $Insert(C, K, D)$. This operation yields a new CAM C' where:

$$\begin{aligned} C'.present &= \lambda\, key \,.\, key = K \lor C.present(key) \\ C'.data &= \lambda\, key \,.\, ITE(key = K,\, D,\, C.data(key)) \end{aligned}$$

On the other hand, the effect of deleting the entry associated with key K is expressed by the operation $Delete(C, K)$. This operation yields a new CAM C' where

$$\begin{aligned} C'.present &= \lambda\, key \,.\, \neg(key = K) \land C.present(key) \\ C'.data &= C.data \end{aligned}$$

3.2 Queues

A queue of arbitrary length can be modeled as a record Q having components $Q.contents$, $Q.head$, and $Q.tail$. Conceptually, the contents of the queue are represented as some subsequence of an infinite sequence, where $Q.contents$ is a function expression mapping an integer index i to the value of sequence element i. $Q.head$ is an integer expression indicating the index of the head of the queue, i.e., the position of the oldest element in the queue. $Q.tail$ is an integer expression indicating the index at which to insert the next element. In general, we require $Q.head \leq Q.tail$ as an invariant property. Q is modeled as having an arbitrary state by letting $Q.contents = c_0$, $Q.head = h_0$, and $Q.tail = t_0$, where c_0 is an uninterpreted function and h_0 and t_0 are symbolic constants satisfying the constraint $h_0 \leq t_0$. This constraint is enforced by including it in the antecedent of the formula whose validity we wish to check.

The operation testing if the queue is empty can be expressed quite simply as:

$$isEmpty(Q) = (Q.head = Q.tail)$$

Using this operation we can define the following three operations on the queue:

1. $Pop(Q)$: The pop operation on an non-empty queue returns a new queue Q' with the first element removed; this is modeled by incrementing the head.

$$Q'.head = ITE(isEmpty(Q),\ Q.head,\ \mathbf{succ}(Q.head))$$

2. $First(Q)$: This operation returns the element at the head of the queue, provided the queue is non-empty. It is defined as $Q.contents(Q.head)$.
3. $Push(Q, X)$: Pushing data item X into Q returns a new queue Q' where

$$\begin{aligned} Q'.tail &= \mathbf{succ}(Q.tail) \\ Q'.contents &= \lambda i\ .\ ITE(i = Q.tail,\ X,\ Q.contents(i)) \end{aligned}$$

Assuming we start in a state where $h_0 \leq t_0$, $Q.head$ will never be greater than $Q.tail$ because of the conditions under which we increment the head.

Bounded length queues can be similarly expressed, with an additional constraint in the case of the push operation disallowing a push when the queue is full. In particular, to bound a queue to a maximum length of k (where k is an integer, not a symbolic constant), we add the condition for pushing that $Q.tail$ is incremented only when $Q.tail < \mathbf{succ}^k(Q.head)$, where $\mathbf{succ}^k$ indicates k compositions of the successor operation. We can use similar guard conditions to to model lossy behavior and duplication as well.

3.3 Process Arrays

Lambda expressions can be used to represent systems containing an arbitrary number of identical processes, such as an array of processors in a cache coherence protocol. For each integer state variable of the process state, we define a function expression S, where $S(i)$ denotes the value of this state variable for process i. Similarly, we represent a Boolean state variable as a predicate expression.

We implement an interleaving model of concurrency in CLU by defining a process identifier state variable *pid* that is updated on each step of operation to designate a single active process. Given uninterpreted function symbols A and N of arity 1, *pid* is defined as having a value equal to $A(ctr)$, where *ctr* is an integer state variable with initial value c_0 and next state value ctr' defined as $ctr' = N(ctr)$. Since our verifier checks the validity of the formula for all possible interpretations of A and N, it will include the case where each successive value of *ctr* is unique. The different possible interpretations of A will then cover all possible sequences of process identifiers. Other concurrency models (e.g., parallel updates to elements of the process array that satisfy a predicate) can also be implemented quite readily.

As an example, consider an array *Add1* of processes each having a single state variable indicating the value of a counter. On each step of operation, one process is selected to increment its counter. The process state table for this array is thus a table of counts represented by a lambda expression *cntTbl*. The initial value of *cntTbl* is given by an uninterpreted function symbol c_0 of arity 1 and the next state expression is given by

$$cntTbl' = \lambda\, i \,.\, ITE(i = pid,\, \mathbf{succ}(cntTbl(i)),\, cntTbl(i))$$

3.4 Observations

Uninterpreted functions provide a natural means for abstracting data and data operations. Lambda expressions provide a powerful notation for describing state transformations. Counter arithmetic provides us the ability to express counters and some forms of pointers. The combination of these three modeling constructs enables CLU to express a wide variety of data structures and system types.

4 Decision Procedure

Assume we start with a well-formed formula F_{ver} in CLU expressing some desired system property. The decision procedure must determine whether it is *valid*, i.e., true under all possible interpretations of the function and predicate symbols. Through a sequence of transformations, described below, we convert a formula over the logic to a propositional formula and then use a Boolean satisfiability checker to determine validity.

Expand Lambda Applications. Since CLU syntax does not permit recursion or iteration, each lambda application can be expanded by *beta-substitution*, i.e., by replacing each argument variable with the corresponding argument term. Let us call the resulting formula F_{exp}.

Identify P-Function Symbols. As with PEUF, we can exploit the restricted uses of equations and inequalities to greatly reduce the number of interpretations that must be encoded when we reduce the formula to propositional logic. As described in [7], we can automatically analyze an arbitrary formula to determine those function symbols that satisfy the restrictions of p-functions. The general idea is to determine the polarity of each equation, i.e., whether it appears under an even (positive) or odd (negative) number of negations. Terms can then be classified as either p-terms, i.e., used only under positive equalities, or g-terms, i.e., general terms. Function symbols for which all applications are p-terms can then be classified as p-function symbols. Applications of p-function symbols can be encoded in propositional logic with fewer symbolic variables than can those of general "g-function" symbols. The extensions required for CLU are to deal with inequalities and the successor and predecessor operations.

The first stage in the analysis labels the subformulas occurring in F_{exp} as being *negative* and/or *positive*. First, we start by labeling F_{exp} as being positive. In addition, for each term of the form $ITE(F, T_1, T_2)$, we label F as being both negative and positive. Then we recursively label the subformulas as follows: If formula $F \doteq F_1 \wedge F_2$ is labeled as being positive (respectively, negative), then so are F_1 and F_2. Similarly for $F \doteq F_1 \vee F_2$. If formula $F \doteq \neg F_1$ is labeled as being positive (respectively, negative), then F_1 is labeled as being negative (respectively, positive).

Once the subformulas have been labeled, we identify which subterms in F_{exp} must be considered g-terms. We start by considering every formula $F \doteq T_1 = T_2$ that was labeled as being negative, as well as every inequality $T_1 < T_2$. For these, we must mark T_1 and T_2 as g-terms. Then we recursively label the subterms as follows: If $T \doteq ITE(F, T_1, T_2)$ was labeled as a g-term, then so must be T_1 and T_2. If $T \doteq \mathbf{succ}(T_1)$ was labeled as a g-term, then so must be T_1. Similarly for $T \doteq \mathbf{pred}(T_1)$.

Finally, we classify each function symbol as either a p-function or a g-function symbol. For function symbol f, if any term of the form $f(T_1, \ldots, T_k)$ was labeled as a g-term, then f must be classified as a g-function symbol. Otherwise, it is a p-function symbol.

Remove Function and Predicate Applications. As described in [7], we can replace all applications of uninterpreted functions or predicates of nonzero arity by terms containing only symbolic constants. Our method differs from the more common method introduced by Ackermann [2] in that it replaces each term by a nested series of ITE operations rather than a single symbolic constant. Our method makes it possible to exploit positive equality in encoding possible instantiations of the constants.

As an example, if function symbol f has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$, then we would generate 3 new symbolic constants vf_1, vf_2, and vf_3. We would then replace all instances of $f(a_1)$ by vf_1, all instances of $f(a_2)$ by $ITE(a_2 = a_1, vf_1, vf_2)$, and all instances of $f(a_3)$ by $ITE(a_3 = a_1, vf_1, ITE(a_3 = a_2, vf_2, vf_3))$.

Predicate applications can be removed by a similar process. In eliminating applications of some predicate p, we introduce symbolic Boolean constants $vp_1, vp_2, \ldots$.

This leaves us with a formula F_{const} containing only symbolic constants, ITEs, successors, predecessors, equations, inequalities, and logical connectives.

Partition into Subdomains. We first split the set of symbolic constants V into two sets V_p and V_g. V_p consists of those symbolic constants occurring in F_{exp} that were classified as p-function applications, as well as those constants vf_i that were introduced when eliminating an application of some p-function symbol f. The remaining symbolic constants are in V_g.

We then partition the set of symbolic constants into classes $V_1, \ldots, V_n$. Each constant in V_p is assigned to its own class. Constants in V_g are grouped according to whether their values may be compared by equations or inequalities. We start by assigning each constant in V_g to its own class. We then compute the *dependency* set for each term in F_{const}, denoting some subset of variables in V_g to which this term could evaluate. While doing this, we merge some of the classes so that each dependency set is a subset of some class. For term $T \doteq v$, its dependency set is $\emptyset$ if $v \in V_p$ and is $\{v\}$ if $v \in V_g$. For term $T \doteq \mathbf{succ}(T_1)$, its dependency set is the same as that of T_1. Similarly for $T \doteq \mathbf{pred}(T_1)$. For $T \doteq \mathit{ITE}(F, T_1, T_2)$, its dependency set is the union of those of T_1 and T_2. If the dependency sets of T_1 and T_2 are subsets of two distinct classes, then we merge those classes. For each equation $T_1 = T_2$ and each inequality $T_1 < T_2$, we perform a similar merging if the dependency sets of T_1 and T_2 are subsets of distinct classes.

Compute Ranges. For each symbolic constant v in F_{const} we must determine the maximum amount it can be incremented or decremented by successor and predecessor operations. We do this by labeling each distinct term T in F_{const} by an its lower bound $l(T)$ and its upper bound $u(T)$. These bounds indicate the range over which the term may be decremented or incremented.

The labeling can be implemented as a fixed-point computation, starting with $l(T) = u(T) = 0$ for each term T. Labels are then updated according to the following rules:

Term T	Lower Bound	Upper Bound
$\mathit{ITE}(F, T_1, T_2)$	$l(T_1) \leftarrow \min(l(T_1), l(T))$	$u(T_1) \leftarrow \max(u(T_1), u(T))$
	$l(T_2) \leftarrow \min(l(T_2), l(T))$	$u(T_2) \leftarrow \max(u(T_2), u(T))$
$\mathbf{succ}(T_1)$	$l(T_1) \leftarrow \min(l(T_1), l(T) + 1)$	$u(T_1) \leftarrow \max(u(T_1), u(T) + 1)$
$\mathbf{pred}(T_1)$	$l(T_1) \leftarrow \min(l(T_1), l(T) - 1)$	$u(T_1) \leftarrow \max(u(T_1), u(T) - 1)$

Eventually, this process will reach a point where the bounds do not change. We then use the values of $l(v)$ and $u(v)$ to determine the range of offsets for symbolic constant v.

Instantiate Subdomains. For each class V_i we compute its range as:

$$\mathit{range}(V_i) = \sum_{v \in V_i} (u(v) - l(v) + 1).$$

This determines the size of the finite instantiation we must consider for each symbolic constant in V_i.

Suppose there are K different classes and let M be the maximum value of $range(V_i)$ for any class V_i. Let $k = \lceil \log_2 K \rceil$ and $m = \lceil \log_2 M \rceil$. Then we encode each symbolic constant as a vector of $k + m$ Boolean formulas $\boldsymbol{v}$. For variable v in class V_i, the high order k elements of $\boldsymbol{v}$ correspond to the binary encoding of i. If class V_i contains just a single constant v, then the low order m elements of $\boldsymbol{v}$ are simply the binary representation of $-l(v)$. Since $l(v)$ must be less than or equal to zero, the effect of this is to bias the value used to encode variable v such that this value will never be decremented below zero by any of the **pred** operations. Otherwise, for each variable v we must introduce m' Boolean variables $\boldsymbol{x_v} \doteq x_v^{m'-1}, \ldots, x_v^0$, where $m' = \lceil \log_2 |V_i| \rceil$. The low order m elements of $\boldsymbol{v}$ are then the Boolean formulas expressing the bit-level representation of $\boldsymbol{x_v} - l(v)$.

We then recursively translate F_{const} into a symbolic Boolean formula, where each term is represented as a vector of $k+m$ formulas and each subformula as a single Boolean formula. Each symbolic constant v is represented by the vector $\boldsymbol{v}$, while each symbolic Boolean constant is represented by a Boolean variable. ITE operators are translated to perform a bit-wise multiplexing of the arguments. Successor and predecessor operations are translated as bit-level incrementers and decrementers. Equations and inequalities are translated as comparators. Boolean connectives are translated as Boolean operators.

This translation process takes advantage of the positive equality structure of the formula in a manner similar to that described in [7]. Each symbolic constant in V_p is assigned a fixed bit pattern, greatly reducing the number of Boolean variables required. Beyond the optimizations described here, we could exploit the equation structure between g-terms using some of the techniques described in [18]. However, many of these optimizations cannot be used when terms are compared by inequalities.

Let F_{bool} denote the resulting Boolean formula. We can then use Boolean satisfiability to see if $\neg F_{bool}$ is satisfiable. If it is, then our decision procedure generates a counterexample to the macro-expanded formula F_{exp} by constructing a partial interpretation of the function and predicate symbols over bit vectors of length $k + m$. If $\neg F_{bool}$ is not satisfiable, then we have determined that the original formula F_{ver} is valid.

Analysis. The decision procedure is efficient because the translation to propositional logic only gives rise to a low-degree polynomial blowup in the formula size. Suppose we represent a formula in CLU as a directed acyclic graph (DAG). The size of the formula is the number of nodes in its DAG representation. Consider the CLU formula F_{exp} of size N in which all lambda applications have been expanded. Assuming that the arities of function and predicate symbols are bounded, we can prove that the size of the final propositional formula F_{bool} is $O((N + M^2 + P^2)lg(N))$, where M and P are the number of function and predicate application terms in F_{exp} respectively (including applications of **succ** and **pred**). The M^2 and P^2 terms come from introducing nested *ITE* expressions while eliminating function and predicate applications, and the $lg(N)$ comes from the binary encoding of integer symbolic constants.

In practice, the number of function and predicate applications is far smaller than the total number of DAG nodes, and so the size of F_{bool} grows as $O(Nlg(N))$. In the worst case, expanding lambda applications can result in an exponential blowup in formula size. In our experience, however, the expressions tend to have a linear structure, with each lambda instantiated only once. With this structure, there is no blowup from lambda expansion.

Modifications. The decision procedure described above uses small-domain instantiation to encode integer symbolic constants. We have also experimented with using Boolean variables to encode equations, as in previous work on PEUF [8]. The latter method performs better in some cases because it directly encodes equations that control system operation. For brevity, we omit a detailed comparison from this paper.

5 UCLID

We have built UCLID, a tool to specify and verify systems modeled in CLU. The UCLID specification language can be used to specify a state machine, where the state variables either have primitive types — Boolean, enumerated, or (unbounded) integer — or are functions of integer arguments that evaluate to these primitive types. Details about the specification language may be found in the user's guide [21]. We mention one notable feature about the internal encoding of enumerated types in UCLID. A enumerated type E of k values is encoded as an integer sequence $\{z_E, \mathbf{succ}(z_E), \ldots, \mathbf{succ}^{k-1}(z_E)\}$, where a different symbolic constant z_E is used for each type E. Since variables of an enumerated type can only be compared for equality against other variables of the same enumerated type[4], the decision procedure assigns the function symbol z_E to its own singleton subdomain, and encodes values of the enumerated type with exactly $\lceil lg(k) \rceil$ bits.

The UCLID verification engine comprises of a symbolic simulator that can be "configured" for different kinds of verification tasks, and a decision procedure for CLU. The following verification methods are supported:

1. *Bounded property checking*: The system is symbolically simulated for a fixed number of steps starting from a reset state. At each step, the decision procedure is invoked to check the validity of a safety property. If the property fails, we generate a counterexample trace from the reset state.
2. *Inductive invariant checking*: The system is initialized in a most general state satisfying the invariant to be proved, symbolically simulated for one step, and the invariant is checked on the resulting state.

[4] Enforced by the type-checker in the UCLID front-end

3. *Proving simulation diagrams*, showing that a specification machine simulates an implementation machine. This includes the method of *correspondence checking* for superscalar processors, such as in the style of Burch and Dill [10]. UCLID allows the user to set the values of control variables at different steps of the symbolic simulation. For example, in verifying pipelined processors, this allows the user to specify the steps at which the pipeline must be flushed.

UCLID's decision procedure checks the satisfiability of $\neg F_{bool}$ using either a BDD package or a SAT solver. A very useful feature of UCLID is its ability to generate counterexample traces, like a model checker. A counterexample to a CLU formula F_{ver} is a partial interpretation I to the function and predicate symbols in the formula, which is generated from a satisfying assignment to $\neg F_{bool}$. If the system has been symbolically simulated for k steps, then the interpretation I generated above can be applied to the expressions at each step, thereby resulting in a complete counterexample trace for k steps.

We have used UCLID to model and check safety properties of a variety of systems, including an out-of-order execution unit, a complex load-store unit of an industrial microprocessor, a cache coherence protocol [12], a 5-stage DLX pipeline, and the Alternating Bit Protocol. In particular, using bounded property checking we can handle models with large state spaces such as the load-store unit (which has about 150 state variables with over half of integer type, after abstraction from RTL). The specifications of most of these models are available on the UCLID website [21].

6 Decision Procedure Benchmarking

We have run experiments to compare UCLID's decision procedure with decision procedures for logics of comparable expressiveness, such as the Stanford Validity Checker (SVC) [3]. SVC can decide a superset of CLU, including, in addition, linear arithmetic and bit-vector arithmetic. Most of the example formulas were generated by performing bounded property checking for some number of steps. By varying the number of steps we can generate benchmark formulas of different lengths. All experiments were run on an Intel Pentium III 550 MHz processor with 256 MB of main memory running Linux. For satisfiability checking, we used the mChaff version of the Chaff SAT solver [16].

Figure 2 shows empirical results comparing UCLID against SVC 1.1 over a set of valid formulas. We can draw four conclusions. First, the conversion from F_{exp} to F_{bool} agrees with the theoretical $O(Nlg(N))$ bound. Second, exploiting positive equality has substantial benefits as deciding satisfiability of $\neg F_{bool}$ is much faster. Third, for the CLU logic, UCLID's decision procedure scales better than SVC, outperforming it for large formulas. The times for UCLID, even on the largest formulas, are less than 2 minutes. Finally, the time taken in converting F_{exp} to F_{bool} dominates the time taken by Chaff for small formulas, but the conversion overhead reduces for larger formulas.

Model	steps	#(int vars) in F_{exp}	#(p -vars) in F_{exp}	#(prop vars) in F_{bool}	F_{exp} size	F_{bool} size	UCLID time (sec.) Conversion	SAT	Total	No + (total)	SVC time (sec.)
Load-Store	6	33	14	76	218	942	1.15	0.06	1.21	1.66	10.86
Unit	8	70	23	180	1085	4481	7.81	0.61	8.42	11.61	1851.60
	10	104	39	317	2467	16453	27.46	3.16	30.62	62.87	> 1 day
	12	149	65	466	4553	54288	78.00	33.09	111.09	295.35	> 1 day
Out-of-order	7	39	19	79	735	3658	4.58	0.20	4.78	9.79	2.96
Execution	9	53	24	158	1970	13775	16.29	2.00	18.29	37.71	102.35
Unit	11	67	30	255	3929	37179	44.90	17.00	61.90	149.46	4257.38
Cache	10	26	10	75	1829	6254	5.97	0.32	6.29	26.50	11.49
Coherence	12	30	12	102	2782	12144	11.72	4.41	16.13	165.91	231.12
Protocol	14	34	14	133	3939	21468	20.16	40.92	61.08	> 1 hr.	6640.00
DLX Pipeline	–	105	73	205	639	9476	11.13	2.09	13.22	1897	20.58

Fig. 2. Experimental results for decision procedure. "steps" indicates the number of steps of symbolic simulation, except when the formula was generated in correspondence checking. F_{exp} denotes the original CLU formula and F_{bool} the final propositional formula UCLID generates. "int-vars" is the number of integer symbolic constants in F_{exp} after eliminating function applications, and "p-vars" is number of those symbolic constants that correspond to p-function applications. "UCLID time – Total" is the time taken by our decision procedure. This time has two components: the time for converting F_{exp} into F_{bool}, labeled "Conversion", and the time taken by the SAT solver, labeled "SAT". "UCLID time – No +" indicates the time taken without exploiting positive equality in the conversion. "SVC time" is the time taken by SVC 1.1 to decide F_{exp}.

7 Conclusions and Future Work

Extending EUF by constrained lambda expressions, ordering and counter arithmetic substantially increases the range of systems that can be modeled without losing the benefits of the efficient decision procedure based on PEUF. Moreover, recent advances in building efficient Boolean satisfiability solvers lend support to our approach of deciding formulas in richer logics via efficient translations to propositional logic.

In terms of future work, we have extended the method of encoding equations with Boolean variables to integer equations and inequalities with constant offsets. We have built some support for quantifiers in CLU using automatic quantifier instantiation heuristics. Finally, we are also working on extending the verification capabilities of UCLID to handle some form of reachability analysis.

Acknowledgments

This research was supported in part by the Semiconductor Research Corporation, Contract RID 684, and by the Gigascale Research Center, Contract 98DT-660. The third author was supported in part by a National Defense Science and Engineering Graduate Fellowship.

References

1. P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In *CAV'98*, LNCS 1427, pages 305–318.
2. W. Ackermann. *Solvable Cases of the Decision Problem.* 1954.
3. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *FMCAD'96*, LNCS 1166, pages 187–201.
4. A. J. C. Bik and H. A. G. Wijshoff. Implementation of Fourier-Motzkin elimination. Technical Report 94-42, Dept. of Computer Science, Leiden University, 1994.
5. B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *SAS '97*, pages 172–186.
6. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV 2000*, LNCS 1855, pages 403–418.
7. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. *ACM Transactions on Computational Logic*, 2(1):93–134, January 2001.
8. R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. In *CAV 2000*, LNCS 1855, pages 85–98.
9. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *CAV '97*, LNCS 1254, pages 400–411.
10. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *CAV '94*, LNCS 818, pages 68–80.
11. M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. *Proc. SIAM-AMS*, 7:27–41, 1974.
12. Steven German. Personal communication.
13. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* 1993.
14. R. Jhala and K. McMillan. Microarchitecture verification by compositional model checking. In *CAV 2001*, LNCS 2102, pages 396–410.
15. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *CAV '97*, LNCS 1254, pages 424–435.
16. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC'01)*, pages 530–535, June 2001.
17. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE '92*, LNAI 607, pages 748–752.
18. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domain instantiations. In *CAV '99*, LNCS 1633, pages 455–469.
19. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977. Cambridge, Mass.
20. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *Proc. Computer-Aided Verification (CAV'02)*, July 2002. This volume.
21. UCLID. Available at `http://www.cs.cmu.edu/~uclid`.
22. M. N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Design Automation Conference (DAC '01)*, pages 226–231, June 2001.