Symbolic Model Checking of Declarative Relational Models

Felix Sheng-Ho Chang
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
fschang@mit.edu

Daniel Jackson
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
dnj@mit.edu

ABSTRACT

This paper explores the idea of augmenting traditional model checkers with the expressiveness of a declarative, relational language. The goal is to enable programmers to write very intuitive and compact specifications, in order to allow the automatic verification of more complicated software systems. The key idea is that many structural operations (common in object-oriented programs) can be easily described using relations and relational operators, while other operations are best described using the primitive data types and their operations (such as simple arithmetic operations on numbers). By allowing a mixture of both, and by allowing parts of the model to be described declaratively rather than imperatively, the programmer has the freedom to model each part of the system differently, using the most intuitive and simple constructs. We built a BDD-based model checker for the language, and successfully verified a straightforward model of the dependency algorithm in Apache Ant for up to 5 nodes.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Verification

Keywords

Software model checking, computation tree logic

1. INTRODUCTION

Model checking has been successful at verifying systems with very large state spaces [11]. However, most of the existing tools have poor or no support for models containing relations and expressive relational operators. Previous work[3] has shown that relational operators can often be used to succinctly describe behaviors and operations on structural objects. Furthermore, existing model checking tools require all state transitions to be described explicitly. It is not enough to describe the desired effect of a state transition: an implementation describing how the state changes has to be provided. As a result, the model specification is often verbose, and this increases the likelihood that there is a bug in the specification, or that the specification and the actual system do not match.

We decided to explore the idea of bringing together relational and temporal logic. To test the idea, we designed a proof-of-concept specification language that augments traditional model checkers with relational logic. The model can be written declaratively, imperatively, or as a mixture of both. In addition to the commonly supported data types such as integers and booleans, the model can also use sets, relations, and relational operators. Safety properties can be specified using both relational and temporal operators, using the temporal logic CTL[7], and then checked automatically by a model checker. We implemented a proof-of-concept BDD-based model checker for this language and used it to verify several algorithms, including the topological sorting algorithm in the Apache Ant program.

Our main contributions are the development of techniques to embed relational data structures into symbolic model checkers, the design of a simple yet expressive specification language based on these techniques, and the implementation of a proof-of-concept model checker. To our knowledge, this is the first checker that smoothly combines first-order quantifiers, relational operators, and temporal logic operators in a single framework.

ICSE'06, May 20–28, 2006, Shanghai, China.

2. SYNTAX

2.1 Scalars

Our language supports two scalar data types: **integers** and **enumerations**. When an **integer variable** is declared, the programmer has to specify the size of the variable in terms of the number of bits required. Integer operations such as addition, subtraction, multiplication, and division are supported. They are performed using either 2's complement arithmetic or unsigned arithmetic, depending on the version of the operator being used.

An **enumeration** defines a set of distinct scalar values. Each variable can then take on one of the values at any given time. For example, we could define **Employee** to be an enumerated type with 4 possible values: Alex, Beth, Carl, and

```
Expr ::= emptyset
       | Variable
       | Expr {+ & - . ->} Expr        union, intersection, difference, relational join, and Cartesian product
       | {~ ^ *} Expr                  transpose, transitive closure, and reflexive transitive closure

Formula ::= true
          | false
          | Variable
          | Formula {&& || =>} Formula                  conjunction, disjunction, and implication
          | ! Formula                                   negation
          | {AG AX AF AU EG EX EF EU} Formula           CTL operation over a formula
          | {all some one no lone} Variable Formula     quantification over a formula
          | {all some one no lone} Expr                 cardinality test on a set-valued expression
          | Expr in Expr                                subset/membership test on two expressions

Statement ::= Var := Expr;
            | Var := Formula;
            | Formula;
            | if Formula Statement
            | if Formula Statement else Statement
            | foreach Variable:Expr Statement
            | FunctionName(Expression, Expression, ···);
```

Figure 1: Core Syntax of the Language

Dave. We could also omit the names. For example, to model the interaction of 10 communicating nodes, we could define **Node** to be an enumerated type with 10 distinct values, without the need to give them names.

Booleans are implemented as a built-in enumerated type with two possible values: **true** and **false**.

2.2 Sets and Relations

Variables can also be declared to be a **relation** among scalar types. For example, to store the languages that each employee can speak, a variable can be declared to be a relation from **Employee** to **Language**. Relations can be of any arity greater than zero. **Sets** are just relations with arity one. The list of operations we support is listed in Figure 1.

We find that the relational join and transitive closure operators especially useful in modeling structural operations and properties.

Two arbitrary tuples **a** and **b** can be joined if and only if the last component of **a** is equal to the first component of **b**. If they can be joined, the join is the concatenation of the two tuples, with the last component of **a** and the first component of **b** removed. The **relational join** of two relations **A** and **B**, with arity **n** and **m**, is a new relation with arity **n+m−2**, consisting of all the possible joins of tuples from **A** and **B**.

This operator nicely captures the notion of objects and fields: given a relation f representing the mapping from objects to their field contents, and given a set S containing just a single object, then the C/C++/Java-like notation of $S.f$ will contain exactly S's value of the field f.

The **transitive closure** is defined only on binary relations. Given a binary relation **A**, its transitive closure is the smallest transitive relation that contains A. In other words, it is the limit of the infinite series $\mathbf{A} + \mathbf{A}.\mathbf{A} + \mathbf{A}.\mathbf{A}.\mathbf{A} + \cdots$

This operation is especially useful in modeling operations that traverse complicated structures. For example, searching a tree or DAG is easily accomplished by taking the transitive closure of the pointer fields, then dereferencing the result: there's no need to write complicated traversal procedures to walk over the pointers.

2.3 Statements

Our tool supports both imperative and declarative statements. The syntax for imperative statements is similar to Pascal. For example, if `A`, `B`, and `C`, are integer variables, then the statement "`A:=B+C;`" changes the value of A to be equal to the sum of B and C. Other control constructs such as "for" loops and "if-then-else" statements are also provided.

Unlike many other modeling languages, our language also allows declarative statements that declare the desired effects rather than explicitly describing the exact action. Inside a declarative statement, primed variables (`A'`, `B'`, ···) refer to the value of the variables after this statement, where as the unprimed variables (`A`, `B`, ···) refer to their values before this statement.

For example, the statement "`(A == A' && A !in B');`" specifies that the value of A shouldn't change, but the new value of B can be any value that does not contain A as a subset. In particular, there are no implicit frame conditions in declarative statements: anything can change unless explicitly specified to be unchanged.

Borrowing from Alloy's syntax, we use the `in` keyword to mean "is a subset of". Because scalars are represented as sets constrained to be singletons, and since we do not allow higher-order sets, scalar `A` is in `B` if and only if the singleton set containing `A` is a subset of `B`. Thus there is no ambiguity in using the `in` keyword to mean both "is a subset of" and "is a member of".

2.4 Procedures

A procedure is simply a list of statements. When a procedure is executed, each statement is executed sequentially, except in the case of loops or recursive calls.

At least one of the procedures must be declared as a top-level procedure. The semantics of the model is simply that, at any given step, one of the top-level procedures is nondeterministically chosen and executed.

Executing a model generates an infinite list of successive system states. Since there may be more than one top-level procedures, and some of them may be nondeterministic, the possible system states form a tree: each state s has a finite number of possible next states $s_1 \cdots s_n$.

Since the state space is finite, every branch of the tree eventually leads to a previously visited state. Therefore, this infinite tree of states can be viewed as a finite Kripke structure. A model in this language thus defines a finite Kripke structure, and the top-level procedures define its allowable transitions.

2.5 Assertions

Our approach should support CTL [7], LTL[15], or richer logic, but our choice to build the proof-of-concept BDD model checker makes CTL the ideal logic for writing property assertions.

At any given state, the A operator specifies a property that must hold for every possible future execution trace, and the E operator specifies a property that must hold for at least one possible future execution trace. The A and E operators must be coupled with one of the four state operators: G, X, F, and U.

On any given execution trace, the G operator specifies a property that must be true at every state along the path. The X operator specifies a property that must be true at the next state along the path. The F operator specifies a property that must eventually become true, at least once. The U operator specifies a pair of properties p_1 and p_2, where p_2 must eventually become true in at least one future state, and p_1 must be true at every state prior to that.

Together, there are 8 possible CTL operators: AG, AX, AF, AU, EG, EX, EF, and EU.

3. EXAMPLE: MUTUAL EXCLUSION

Even though the classic problem of mutual exclusion is trivial, it helps to illustrate the features of this system.

The problem involves a system of P processes and M mutexes. At any given time, a process is either waiting or running. A running process can choose to either do nothing, or it can give up a mutex that it currently holds. A running process can also request a mutex that it doesn't currently hold. If no other process holds that mutex, then the requesting process will get possession of the mutex and remain in the running state. Otherwise, the requesting process enters the waiting state. Whenever a running process chooses to give up one of its mutexes, if there is one or more processes waiting on that mutex, then one of them will be arbitrarily chosen to receive the mutex. Only the chosen process will enter the running state; all other waiting processes will remain in the waiting state.

When implementing a mutual exclusion algorithm, the main problem to avoid is deadlock: suppose initially process 1 holds mutex 1 and process 2 holds mutex 2; if process 1 chooses to wait on mutex 2, and process 2 chooses to wait on mutex 1, then both processes will be in the waiting state forever. (A waiting process cannot choose to give up a mutex until its request is first granted; since process 1 can resume only when process 2 releases its mutex, and vice versa, neither process can make any progress from that point on.)

```
// There are exactly 4 mutexes and 4 processes.
enum mutex={m1,m2,m3,m4};
enum process={p1,p2,p3,p4};

// "has" is a relation from processes to mutexes,
// representing the mutexes currently held
// by a process.
process->mutex has;

// "wait" is a relation from processes to mutexes,
// representing the mutex each process
// is waiting for, if any.
process->mutex wait;

toplevel void ReleaseMutex(process p, mutex m) {
  if (no p.wait) && (m in p.has) {
    p.has := p.has - m;
  }
}

toplevel void GrabMutex(process p, mutex m) {
  if (p.wait == m) && (no has.m) {
    p.has  := p.has + m;
    p.wait := emptyset;
  }
}

toplevel void Wait(process p, mutex m) {
  if (no p.wait) && (p.has < m) { p.wait := m; }
}
```

Figure 2: Mutual Exclusion

The model in Figure 2 describes one particular solution to the problem: by assigning a total ordering on the mutexes, then requiring that each process can wait only on a mutex whose position is 'greater' than all other mutexes that the process currently holds.

In this example, there are four processes competing for four mutexes. All three procedures are top-level procedures, and they describe the allowable state transitions. At any given point in time, if a process is waiting for a mutex that has just become available, the process can grab it. If a process isn't waiting for anything, it can choose to release a mutex it currently holds, or it can choose to wait on any mutex that is greater than any mutex that the process currently holds. (In this example, there are four mutexes: **m1**, **m2**, **m3**, and **m4**. So if a process already holds **m2**, it can choose to wait on only **m3** or **m4**).

Assertion 1: Mutual Exclusion. The following assertion ensures that mutual exclusion is always satisfied:

```
assert !bad && (no has+wait)
=> !bad && AG (all m:mutex | sole has.m);
```

It says that if initially none of the processes is holding any mutex or is waiting for any mutex, then in all possible future execution paths, at every state along each path, each mutex is always held by zero or one process.

"bad" is a built-in boolean variable that is set to true whenever a runtime exception has occurred; and once it becomes true in a given system state, it will remain true in all

future system states. For example, assigning a set of multiple values into a scalar variable will set the "bad" flag to true.

Assertion 2: No Deadlock. The following assertion ensures that the processes never deadlock in this system:

```
assert !bad && (no has+wait)
  =>
!bad &&
AG all p:process | all m:mutex |
(p->m in wait) => (EF p->m in has);
```

It says that if initially none of the processes is holding any mutex or is waiting for any mutex, then in all possible future execution paths, at every state along each path, if a process waits for a mutex, it is possible for the process to eventually acquire the mutex. The EF operator is used instead of AF, because AF would be trivially violated by a process that never releases its mutex.

4. EXAMPLE: TOPOLOGICAL SORT

We wanted to see if we could model a realistic Java procedure that performs graph manipulations. We chose the dependency algorithm in the open-source build tool Ant (version 1.5.4), because its size is moderate, it has recursive calls, and its correctness depends on five rich properties on the graph. The code from the org.apache.tools.ant.Project class is in Figure 3. It has a single top-level procedure (**topoSort**) and a recursive procedure (**tsort**) that does the actual topological sort. Our model (shown in Figure 4) is a straightforward translation of the Java code.

There are two important observations. First of all, our system allows the input graph to be specified declaratively. By using sets and relations as the state components, we can allow an arbitrary topology, and automatically verify that the algorithm will perform correctly given any input that satisfies some declarative constraint (in this case, the requirement that the input graph be acyclic).

The second observation is that the relational operators allow the safety properties to be specified very succinctly. Even though the data structures here and the graph traversal algorithm can be represented using primitive data types in other model checkers, to do so is painful at best. This example showcases the ease of using relational operators to model operations on dynamic structures. For example, the ability to take the transitive closure of the "dependOn" relation makes it very easy to state the "has_cycle()" property that checks whether there is a cycle in the graph or not.

In the case of topological sort, there are exactly 5 properties that any topological sorting algorithm must satisfy. Our model checker can verify that indeed all five properties are satisfied:

Assertion 1: The assertion that every node appears exactly once in the sorted list can be simply stated as follows:

```
bool has_cycle() {
  some t1:Node | t1 in t1.^dependOn;
}

assert !bad && !has_cycle()
=> AX (!bad && (all S:Node | one list.S));
```

Here, the initial condition of the system is stated declaratively: any non-cyclic graph is allowed. This avoids the need to explicitly enumerate all possible graph configurations; instead, all possible configurations are considered.

Assertion 2: If node A depends on node B, then node B's position in the sorted list will always come before node A.

```
assert all a,b:Node |
(!bad && !has_cycle() && b in a.^dependOn)
=> AX (!bad && one list.a && one list.b
       && list.b < list.a);
```

Assertion 3: If the starting node "Node1" does not depend on some node B, then B's position in the sorted list will come after Node1.

```
assert all S:Node |
(!bad && !has_cycle() && S!=Node1
&& S !in Node1.^dependOn)
=> (!bad && (AX list.Node1 < list.S));
```

Assertion 4: This assertion states that if there are no cycles in the dependency graph, then the system won't enter the error state:

```
assert !bad && !has_cycle() => (AX !bad);
```

Assertion 5: Finally, this assertion states that if there are cycles in the dependency graph, then the system will always enter the error state:

```
assert !bad && has_cycle() => (AX bad);
```

5. IMPLEMENTATION

5.1 System States

This implementation uses BDD nodes [16] to represent the system states and state transitions.

As is standard, each state variable is simply encoded by a number of BDD variables, representing the range of possible values. For example, a 4-bit integer variable can take on 16 possible values; so it can be encoded by 4 BDD variables. An enumeration variable that can take on 8 possible values can be encoded by 3 BDD variables.

Sets and relations are more expensive: a relation variable can contain many possible tuples. Each tuple that could possibly be in a relation requires a separate BDD variable to represent whether the tuple is in it or not.

The arithmetic and relational operators are translated into the corresponding operations on the BDD graphs representing the corresponding bit positions, as is standard in other symbolic model checkers.

One complication involves performing operations between scalars and non-scalar values: since the bit positions are encoded differently for scalars and non-scalars, the high-level operations between them cannot be decomposed into simple operations on each bit. Instead, our approach first promotes the scalar into a set containing exactly 1 element and then performs the relational operations on the resulting sets. On the other hand, there is no way to automatically demote a set expression into a scalar value, since the set may not always contain exactly one element.

5.2 State Transitions

As is standard, state transitions are represented by doubling the number of BDD variables used: one set for the

```
// tsort() performs a Depth-First-Search from S.
void tsort(String root..) {
  state.put(root, VISITING);
  visiting.push(root);
  Target target = (Target) targets.get(root);
  if (target == null) { ... throw new BuildException(...); }
  for (Enumeration en = target.getDependencies(); en.hasMoreElements();) {
    String cur = (String) en.nextElement();
    String m = (String) state.get(cur);
    if (m == null) tsort(cur, targets, state, visiting, ret);
    else if (m == VISITING) throw makeCircularException(cur, visiting);
  }
  String p = (String) visiting.pop();
  if (root != p) { throw new RuntimeException(..); }
  state.put(root, VISITED);
  ret.addElement(target);
 }

// It first calls tsort() on the root node.
// It then calls tsort() on any unvisited node in order to detect cycles.
Vector topoSort(String root, Hashtable targets) {
  Vector ret = new Vector();
  Hashtable state = new Hashtable();
  Stack visiting = new Stack();
  tsort(root, targets, state, visiting, ret);
  for (Enumeration en = targets.keys(); en.hasMoreElements();) {
    String curTarget = (String) en.nextElement();
    String st = (String) state.get(curTarget);
    if (st == null) tsort(curTarget,targets,state,visiting,ret);
    else if (st == VISITING) throw new RuntimeException(..);
  }
  return ret;
 }
```

Figure 3: Java code for topological sort

```
enum Node { Node1, Node2, Node3, Node4, Node5 };
enum Label { VISITING, VISITED };

Node->lone Label label; // Each Node can be unlabeled, VISITING, or VISITED. ('lone' means zero or one)

Node->Node dependOn;    // Each Node depends on 0/more other Nodes.

int->lone Node list;    // The sorted list and its length.
int length;

void tsort(Node S) {
  S.label := VISITING;
  foreach Y:S.dependOn {
    if (no Y.label) tsort(Y); else if (Y.label==VISITING) bad:=true;
  }
  S.label := VISITED;
  list := list + length->S;
  if (length>5) bad:=true; else length++;
}

toplevel void topoSort() {
  bad:=false;
  length:=0;
  label:=emptyset;
  list:=emptyset;
  tsort(Node1);
  foreach S:Node {
    if (no S.label) tsort(S); else if (S.label==VISITING) bad:=true;
  }
}
```

Figure 4: the model

"present state" and one set for the "future state". An assignment statement such as `a:=b+1` is translated into a relation between present and future states: `(a'==b+1 && b'==b)`, where the presence of the apostrophe indicates that the variable refers to the future rather than the present state.

One complication arises when trying to assign a set into a scalar variable. To handle this correctly, we must make sure that the set always contains exactly one element. More precisely, whenever the set contains zero or more than one element, we want that to be reported as an error. This problem cannot be solved by having stronger typing rules: the program being analyzed could perform a series of set manipulations knowing the end result will be a singleton, but a type checker cannot determine this in general.

To solve this problem, we introduce an additional global variable named "bad". Legal statements will preserve the value of "bad", but illegal statements should set "bad" to true. Assuming there are three variables in the system states `a`, `b`, and `c`, we translate the expression `a:=expr;` into the boolean formula:

```
(a'==expr && b'==b && c'==c
 && bad'==bad && size(expr)==1)
    ||
(bad'==true && size(expr)!=1)
```

where `size(x)` denotes the number of elements in the set `x`. The clauses `b'==b'` and `c'==c` are frame conditions ensuring `b` and `c` are not changed by this statement. Programmers can then assert that the "bad" flag is never true in any reachable states, for example.

As is standard, we translate each statement in a procedure into a relation representing the set of allowable state transitions. Each procedure then simply represents the sequential composition of all the statements in the procedure. Consider a procedure containing two statements, s_1 and s_2:

```
void proc() {
  s1; s2;
}
```

Let a and b be the transition relations represented by s_1 and s_2, respectively. Then the transition relation for the entire procedure simply represents the effect of first performing the first statement, and then performing the second statement on the outcome of the first statement. That can be easily computed by first declaring a set of temporary BDD variables $\hat{s}$ representing the intermediate system state, composing the two relations a and b together, then existentially quantify out the intermediate system state:

$$\exists\, \hat{s}\ (a[\hat{s}/s']) \wedge (b[\hat{s}/s])$$

The approach given above is standard technique in symbolic model checking, and can deal with everything in this language, except loops and recursive procedure calls. The following sections explain how they are translated by our model checker. In particular, the treatment of potentially infinite recursive calls and loops is interesting, as they are not found in typical model checkers.

5.3 Recursive Calls

The standard way of deriving the semantics of a recursive function is to compute the least fixed point of its generating function. It typically starts by translating the function body as if the inner call does nothing. This gives an approximate transition relation for F. Then the function body is translated again where this approximation is used to translate the inner call. This process repeats until a fixed point is reached, and that additional substitution yields no further changes to the transition relation. The standard algorithm has a drawback: sometimes it cannot detect the presence of possible infinite loops. Consider the following piece of code:

```
void F() {
   S := an arbitrary value from 1 and 2.
   if (S==1) F();
}
```

The standard approach first assumes that F() is the empty transition relation. In the next iteration, the first statement gives 2 possible next states: S=1 and S=2. The first choice does not have a successor, and will be eliminated in the sequential composition, leaving the second choice as the only choice. The third iteration finds no changes in the translation of F(), and concludes that for any input value of S, F() will have a successor state with S set to 2. The possibility that the value 1 may be chosen indefinitely and thus leading to infinite loops is not detected.

Our technique solves this problem by introducing one more system state called "loop". The transition relation for every statement is augmented, such that whenever the system enters the "loop" state, it stays in that state forever.

We still perform the iterative fixed point computation as above. But instead of assuming F() does nothing initially, we assume F() takes every input state to the "loop" state.

This is basically a slight modification to the standard approach of using $\perp$ in the denotational semantics. Instead of having just $\perp$, we added a state in the lattice on top of $\perp$ to denote a "could loop" state.

Theorem: Fixed point computation using the augmented system state always terminates

Proof: The fixed point computation always terminates if the generating function is monotonic in some pointed Complete Partial Order (CPO). Let S be the finite set of system states, including a special "loop" state labeled L. Let E be Powerset(S)\{$\emptyset$}. We'll define the relation $\sqsubseteq_{\mathrm{E}}$ on E as follows: $\forall e_1, e_2 \in E \mid e_1 \sqsubseteq_{\mathrm{E}} e_2$ if and only if ($e_1 \subseteq e_2 \cup \{\mathrm{L}\}$) and ($e_1 = e_2$ or $\mathrm{L} \in e_1$). Then every transition relation in our system is a total function in S→E. If we define the relation $\sqsubseteq_{\mathrm{F}}$ on S→E as:

$\forall f_1, f_2 \mid f_1 \sqsubseteq_{\mathrm{F}} f_2$ if and only if $\forall s \in S\ f_1(s) \sqsubseteq_{\mathrm{F}} f_2(s)$

then both relations are reflexive, transitive, and antisymmetric; thus, the set of functions S→E form a pointed CPO under $\sqsubseteq_{\mathrm{F}}$, with the function "$\lambda s.\{\mathrm{L}\}$" as the bottom value. It is obvious that every generating function in our system is monotonic with respect to $\sqsubseteq_{\mathrm{F}}$. Thus, by Tarski's theorem [1], every generating function has a least fixed point.

Our new algorithm is sound because for every recursive function F, the transition relation T that it finds will always satisfy the following condition: substituting T for every recursive call sites inside the body of F will produce T as the resulting transition relation (since, by definition, T is the least fixed point of the recursive function's generating function).

For example, our new algorithm correctly translates the previously-shown recursive function F() into the transition relation that takes any input S into either the state S=2 or the "loop" state. Programmers can now incorporate the

"loop" predicate in the assertions to check for the possibility or inevitability of infinite loops.

Mutually recursive functions can be handled easily, since we can always transform a set of mutually recursive functions into a single recursive function with an additional "selection" parameter.

5.4 Loops

All loops can be converted into "while" loops surrounded by possible additional statements. For example, the "for" loop "`for(a;b;c) body;`" can be translated as follows:

```
a;
while(b) {body;c;}
```

Likewise, the loop "`do body; while(p);`" is equivalent to the following piece of code:

```
body;
while(p) body;
```

Therefore, it suffices to consider only the translation for the following simple "while" loop:

```
while(p) body;
```

For every loop, we can automatically introduce a new recursive procedure representing the loop. In the above case, the corresponding recursive procedure is as follows:

```
void rec() {
   if (p) { body; rec(); }
}
```

In the original model, the occurrence of the loop can then be replaced by a simple call to this recursive function instead. For example:

```
..                  ..
while(p) c;         rec();
..                  ..

[Original Model]    [Model where loops are transformed
                     into recursive procedures]
```

Since the additional recursive functions are anonymous, they cannot possibly be invoked anywhere else. Therefore, their introduction does not alter the set of execution traces specified by the model. It is then straightforward to show that the behavior of the model is preserved after replacing each loop by a call to its corresponding anonymous recursive function.

This way, we can apply our new algorithm for recursive functions on loops and use it to detect the possibility of infinite loops.

5.5 CTL Assertions

As is standard, the CTL operators take a transition relation, and computes the set of states that satisfy the relation, one or two predicates, and the operator. What is novel is the mixing of CTL and relational operators. If relational operators are used in the transition relation or the predicate itself, our technique translates them into simple relations between pre-state and post-state variables. Thus, no change to the CTL model checking algorithm is required.

Likewise, set comprehension operators can be applied to formulas containing CTL operators, because each CTL operator simply computes a set of satisfying states. It is no different than any other formula that represents a set of states by evaluating to true for every state in the set.

Therefore, the mixing of CTL and relational operators was surprisingly straightforward.

6. RESULTS

BDD representations are canonical: given the same state space and the same variable ordering, an imperative description of an operation will take up the same amount of memory as a declarative description of the same operation.

Since our relational data types and the corresponding algorithms operate on BDD nodes directly, it should be straightforward to embed them as front ends to existing BDD-based model checkers such as SMV[13] with no performance penalty (other than the compilation cost of translating these relational operators).

Nevertheless, to allow rapid exploration of different design trade-offs, we found it useful to build our own proof-of-concept model checker. The results shown below are based on our own tool which does not yet use partial order reduction, disjunctive partition, or many other standard optimization techniques commonly found in model checkers such as SMV.

N	Mutual Exclusion	Topological Sort
2	0.05 seconds	0.07 seconds
3	0.4 seconds	0.5 seconds
4	42 seconds	10.5 seconds
5	25 min 12 seconds	5 minutes 49 seconds
6	$>$ 35 minutes	$>$ 20 minutes

Table 1: Time

N	Mutual Exclusion	Topological Sort
2	77 nodes	233 nodes
3	1649 nodes	1559 nodes
4	93587 nodes	11054 nodes
5	835049 nodes	283372 nodes
6	Out of memory	Out of memory

Table 2: Memory Usage

The experiments were run on a Pentium 4 machine running FreeBSD 4.11 with up to 1G of user-space memory.

7. RELATED WORK

Alloy[4] is a structural modeling language that supports relations and relational operators based on first-order logic. Alloy models are always written declaratively, and it is difficult to model integer arithmetic using Alloy. Furthermore, since Alloy specifications do not have a built-in notion of states, they cannot be automatically checked against properties containing temporal operators.

This work combines the relational features of Alloy with imperative constructs, control constructs (such as loops and recursive function calls), and full integer arithmetic support. By allowing models to be written declaratively or imperatively using simple data types as well as relations, the programmer can concentrate more on writing the model and less on struggling with the limited expressiveness of the tool.

B[2] is a rich formal specification language. B and Alloy are most similar to our specification style. While B has extensive tool support, most of it is focused on aiding the programmer in the software development process. Tools exist for deriving proof obligations and integrating with theorem provers, but in general the proofs cannot be automated. In contrast, our approach finitizes the problem so that the properties can be checked automatically within a bounded size without requiring user guidance.

SMV[13] is a BDD-based model checker that takes a description of a finite Kripke structure, and verifies it against safety and liveness properties written in the temporal logic CTL[7]. Its input language supports only simple data types, and does not support dynamic structures and relational operators.

SMV is very efficient, and has been successfully used to verify many hardware and software systems. One of its weakness is its lack of rich relational data types. To describe the common operations on structural objects, the programmer has to encode the relations using arrays of primitives, and then iterate through them whenever a value is required.

Not only is it cumbersome and error-prone, this approach also hardcodes the sizes of the various objects. To change the size later, the programmer often needs to go through the whole model to modify the loop boundaries and to change the list of constants. This can lead to more modeling errors.

In contrast, our approach allows the programmer to mix relational, arithmetic, and logical operators throughout the specification.

Java PathFinder[18] is a tool and a framework for checking Java programs. At its core is a custom-made Java Virtual Machine that executes Java bytecodes and uses backtracking and other searching algorithms to check for property violations.

Its main advantage is the ability to check all pure Java programs. However, being an explicit-state model checker based on a Java Virtual Machine, it can check only properties expressed in terms of legal Java expressions. It is cumbersome to describe properties involving complex data structures containing pointers. It is not straightforward, in Java, to specify transition relations such as "dereferencing a pointer an arbitrary number of times" or "taking the union of a set of traversed fields".

SLAM[17] automatically generates a boolean abstraction of C programs and uses traditional model checkers to check properties. Through a series of refinement, it infers additional clauses and assigns boolean bits to represent them. It has been very successful in verifying Windows device drivers, but its inference algorithm is optimized for low-level predicates and cannot infer complex relations between objects.

SPIN[8] is an explicit-state model checker that checks a system by attempting to visit all reachable states. Safety properties specified in linear temporal logic (LTL[15]) and liveness properties can be checked automatically.

Furthermore, the user can optionally choose to enable various state compression methods (such as bitstate hashing) to check systems with even larger state spaces.

SPIN has been successful applied to verify many types of systems, especially systems consisting of concurrent communicating processes. However, its basic specification language also does not support relations and relational operators; describing complex graph manipulation algorithms and their safety properties is cumbersome.

8. CONCLUSION

The goal of this research is to show that it is possible to embed relations and expressive relational operators into a standard CTL symbolic model checker. Our techniques should be applicable to many other symbolic or explicit-state model checkers.

This paper demonstrates that the traditional BDD-based model checkers can be augmented to work with declarative, relational models. The relational operators allow many common structural operations to be modeled more clearly and succinctly, and the temporal logic operators make it easy to specify complicated safety and liveness properties that depend on unbounded execution traces. Our approach extends the data types of existing model checkers by finitizing sets and relations and then encoding them using standard representations.

There were three main difficulties in the research. First of all, the language design underwent many changes in the hopes of finding a right balance between imperative modeling language (like SMV [13], Murphi [5] [6], or SPIN [8]) and a rich object-oriented or relational language (like Java [12], Alloy[3], or OCL[10]). Rich, expressive languages reduce the gap between the specification syntax and the actual implementation language, but often impedes automatic analysis. Following the success and lessons of Alloy, we opted for a very simple language with just enough constructs to describe scalars and relations. By adding a carefully chosen set of relational operators, we are able to model structure manipulation algorithms as well as simple integer calculations.

The second challenge involved finding efficient and deterministic algorithms for computing the transition relation of statements and declarative constraints; in particular, the translation of (potentially infinite) recursive function calls was nontrivial.

The third main challenge is in finding ways to optimize the BDD operations and reducing the memory usage. In terms of speed, we rearranged our algorithms to reuse previously calculated values rather than recomputing them. On top of the BDD library's built-in cache, we added additional cache to remember the values of different phases of CTL computation and reuse the values when possible.

In terms of memory usage, we can only hope to reduce the graph size during computation, because the final graph size is fixed (since BDD representation is canonical). On top of the BDD package's automatic reordering, our system allows users to specify the initial ordering of variables. We have found that, with the topological sort algorithm, the penalty of a badly chosen initial ordering is severe: we were not even able to verify the algorithm for N=5. We found a locally optimal variable ordering for our experiment by trial and error.

9. FUTURE WORK

We hope to apply our techniques to embed relational data types into state-of-the-art model checkers and augment them with expressive relational operators. Many domain-specific model checkers already exist for different applications. We believe they can benefit from having the ability to clearly and succinctly describe complex data structures and operations that manipulate these structures.

One area of ongoing research involves the development of techniques to reduce the memory usage of the BDD model checker. Since the BDD representation for every boolean

function is always canonical, improving the method of computing them will not reduce the ultimate memory requirement. Instead, we are exploring compilation optimization techniques to exploit the inherent modularity of procedure boundaries in order to avoid BDD state space explosion.

Alloy 3's new type system [9] is much simpler than Alloy 2's type system, and yet enables richer analysis of relation types and the domains and ranges of relations. We are considering adopting a similar type system and doing similar compile-time analysis to reduce the number of BDD variables needed to represent certain relations.

Following the success of Alloy's use of SAT [14] solvers to achieve greater scalability, we are also exploring non-BDD based techniques such as SAT and other decision procedures.

10. REFERENCES

[1] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. In *Pacific Journal of Mathematics 5*, pages 285–309.

[2] J. Abrial. The B Book - Assigning Programs to Meanings. Cambridge University Press, 1996.

[3] Daniel Jackson. Automating First-Order Relational Logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, 2000.

[4] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A Micromodularity Mechanism. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, 2001.

[5] David L. Dill. The Murphi Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393, 1996.

[6] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 522–525, 1992.

[7] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*. ACM Press, 1986.

[8] Gerard J. Holzmann. The Model Checker SPIN. In *IEEE Transactions on Software Engineering, Vol. 23, No. 5*, 1997.

[9] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A Type System for Object Models. In *Foundations of Software Engineering, Newport, CA*, 2004.

[10] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[11] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[12] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[13] Ken L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th conference on Design automation*, pages 530–535. ACM Press, 2001.

[15] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symp. Foundations of Computer Science*, 1977.

[16] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[17] T. Ball and S.K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *POPL '02: Proceedings of the 29th ACM Symposium on the Principles of Programming Languages*, New York, NY, USA, 2002. ACM Press.

[18] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *ASE '2000: 15th IEEE International Conference on Automated Software Engineering*, pages 3–11, 2000.