Model Checking Using SMT and Theory of Lists

Aleksandar Milicevic[1] and Hillel Kugler[2]

[1] Massachusetts Institute of Technology (MIT), Cambridge, MA, USA
aleks@csail.mit.edu
[2] Microsoft Research, Cambridge, UK
hkugler@microsoft.com

Abstract. A main idea underlying bounded model checking is to limit the length of the potential counter-examples, and then prove properties for the bounded version of the problem. In software model checking, that means that only program traces up to a given length are considered. Additionally, the program's input space must be made finite by defining bounds for all input parameters. To ensure the finiteness of the program traces, these techniques typically require that all loops are explicitly unrolled some constant number of times. Here, we show how to avoid explicit loop unrolling by using the SMT Theory of Lists to model feasible, potentially unbounded program traces. We argue that this approach is easier to use, and, more importantly, increases the confidence in verification results over the typical bounded approach. To demonstrate the feasibility of this idea, we implemented a fully automated prototype software model checker and verified several example algorithms. We also applied our technique to a non software model-checking problem from biology – we used it to analyze and synthesize correct executions from scenario-based requirements in the form of Live Sequence Charts.

1 Introduction

We present a finite-state model-checking technique, based on satisfiability solving, that does not require the user to explicitly bound the length of the search traces. We use the SMT *Theory of Lists* [7] to model potentially infinite search traces. A benefit of this approach is that it does not require providing the number of loop unrollings. Similarly, when trying to solve a planning problem, we do not have to specify the maximum number of steps needed to solve the problem. This way, we can achieve most of the benefits of the unbounded case. Unfortunately, in some cases our approach cannot prove that no counter-example exists (e.g., in the presence of infinite loops in the program), so it is not fully unbounded.

We use a list to model an unbounded search path. Every list element represents a single state traversed during the search. In order to find a path to an error state, we impose the following constraints on that list: (1) the first element is a valid initial state, (2) every two consecutive elements represent a valid state transition; and (3) the last element corresponds to one of the states we want to reach (*error states*). Having formulated the problem in this way, we can run an SMT solver, namely Z3 [23], to search for such a list, without constraining its length. If Z3 terminates and reports that the problem is unsatisfiable, we

have proved that the error states are unreachable; otherwise, we have found a counter-example.

This idea is readily applicable to *software model checking*. In the presence of loops, program traces become infinite. A common resort is to explicitly perform *loop unrolling*, as it is the case with CBMC [10], Forge [16] and [5]. The limitation of this approach is that the number of unrollings must be specified beforehand by the user. Typically, the number of unrollings and the bounds for the input space are specified independently of each other, even though they are almost never independent in practice. For example, in order to verify the "selection sort" algorithm for arrays of length up to N, at least $N-1$ loop unrollings are needed. If the user provides a number less than $N-1$, a tool for bounded verification will typically report that no counter-example can be found within the given bounds, which may trick the user into believing that the algorithm is proven to be correct for all arrays of length up to N. With our approach, to verify the "selection sort" algorithm, the user only specifies the bound for N. Bounds for array elements are not needed in this case, so we can prove the algorithm correct for **all** integer arrays up to the given length N.

The main contributions of this paper are:

– A novel approach to model checking using SMT and the theory of lists: we explain how lists can be used to model unbounded traces;
– Application of this idea to software model checking: we present an optimized encoding of a program, and show that loops need not be explicitly unrolled;
– Execution of Live Sequence Charts case study: we analyzed scenario-based models of biological systems [19], written in the language of Live Sequence Charts (LSC) [15]. We show that declarative scenario-based specifications, written in LSC, can be translated into the logic of SMT, and an off-the-shelf solver can be used to automatically execute them.

2 Background

In order to check whether a safety property holds within some number of states k, one can define k sets of variables, one set for each state, $s_1, s_2, \cdots, s_k$, and then, as with any model-checking problem, assert that the following hold:

1. **Initial State** constraint: $\Theta(s_1)$;
2. **Transition** constraint: $\rho(s_1, s_2) \wedge \rho(s_2, s_3) \wedge \cdots \wedge \rho(s_{k-1}, s_k)$; and
3. **Safety Property** constraint: $\mathcal{P}(s_1) \wedge \mathcal{P}(s_2) \wedge \cdots \wedge \mathcal{P}(s_{k-1}) \wedge \neg\mathcal{P}(s_k)$.

Θ encodes constraints that must hold in the initial state; $\rho(s_{i-1}, s_i)$ is a transition function which returns `true` if and only if the system is allowed to go from state s_{i-1} to state s_i; finally, $\mathcal{P}(s_i)$ is the safety property that we want to prove. In order to find a counter-example, we assert that the safety property doesn't hold in the last state while holding in all previous states. Additionally, the transition function must hold for every two consecutive states. The conjunction of these three formulas is passed to an off-the-shelf solver, which either returns a model encoding a counter-example, or proves that the formula is unsatisfiable

(meaning that the safety property is verified for the given k). This approach is commonly referred to as *bounded model checking using satisfiability solving.*

We focus on how to use the theory of lists to avoid having k copies of the state variables. The theory of lists is currently supported by many state-of-the-art SMT solvers. A description of how other theories can be used to encode programs and why that can be advantageous is presented in [5].

SMT lists are defined recursively: `List<E> = nil | cons (head: E, tail: List)`. For a given list, only two fields, `head` and `tail` are immediately accessible. In addition, predicates `is_cons` and `is_nil` are readily available to check whether a given list variable is `cons` or `nil`. As a consequence, it is not possible to directly access the list element at a given position, or immediately get the length of the list, which is inconvenient when asserting properties about lists.

3 Approach

Our approach is based on the idea of bounded model checking using satisfiability solving, except that instead of explicitly enumerating all state variables (s_1, s_2, $\cdots$, s_k), and thus bounding the length of a potential counter-example, we use only a single variable of type *List of States*. Every list element is of type *State*, which is a tuple of all variables needed to represent the problem state. We still assert the same three constraints, (1) initial state, (2) transition; and (3) safety constraint, but now in terms of a single list variable.

Expressing the initial state constraint is easy, since the first element of the list is immediately accessible. To express the other two constraints, we use an uninterpreted function accompanied with an axiom. More precisely, in order to enforce the transition constraint between every two consecutive elements of the list, we first define an uninterpreted function, named `check_tr`, that takes a list and returns a boolean value. Next we add an axiom (*transition axiom*) to assert that `check_tr` returns `true` when applied to a list if and only if every two consecutive states of that list represent a valid state transition.

A recursive definition of the transition axiom is given in Figure 1. The only case of importance is when the list argument, namely `lst`, is not `nil` and has a non-`nil` next element (`tail`). This is because we only care to assert the transition property between two consecutive elements. We do that by inlining the actual model-checking transition constraint between the current and the next list element. In addition, we have to make sure that all subsequent consecutive elements represent valid state transitions, so we recursively assert that the same `check_tr` function returns `true` for the `tail` of the given list argument.

In order to enforce the safety property on all list elements but the last one, we could similarly define another uninterpreted function and an additional axiom. However, since we already have an axiom that “traverses” the whole list, we decided to include the safety property check in the existing transition axiom. This can simply be done by checking whether the next list element (`tail(lst)`) corresponds to an error state (by inlining the *error condition*, i.e. $\neg\mathcal{P}(s_i)$). If the next element is in fact an error state, we have found a counter-example, so we

force the list to end right there (i.e. its next element must be `nil`). Otherwise, we must keep searching, so the next element in the list must be `cons`.

Finally, it is important to stress the purpose of the instantiation pattern (`PAT: {check_tr (lst)}`) in the `FORALL` clause. This axiom states something about *all* lists. However, it would be impossible for the SMT solver to try to prove that the statement indeed holds for all possible lists. Instead, the common approach is to provide an instantiation pattern to basically say in which cases the axiom should be instantiated and therefore enforced by the solver. In our case, we simply say that every time we apply `check_tr` function to a list, the axiom must be enforced, so that the evaluation of `check_tr` indeed indicates whether the list satisfies both transition and safety property constraints.

```
DEF check_tr: StateList → bool
ASSERT FORALL lst: StateList
    IF (is_cons(lst) ∧ is_cons(tail(lst))) THEN
        transition_condition(head(lst), head(tail(lst))) ∧
        check_tr(tail(lst)) ∧
        IF (error_condition(tail(lst)))
        THEN is_nil(tail(tail(lst)))
        ELSE is_cons(tail(tail(lst)))
  :PAT {check_tr(lst)}
```

Fig. 1: Axiom for the `check_tr` function

```
DEF states: StateList
ASSERT
    is_cons(states) ∧
    initial_condition(head(states)) ∧
    check_tr(states)
CHECK
```

Fig. 2: SMT logic context

The rest of the SMT logic context is given in Figure 2. It provides a generic template for model-checking problems. For a specific problem, the user only needs to define: (1) the `State` tuple (basically enumerate all state variables), (2) initial condition, (3) transition condition; and (4) error condition.

4 Applicability to Software Model Checking

4.1 The Idea

We observe a program as a traditional *Control Flow Graph* (CFG) [3]. The *state* of the execution of a program consists of the current basic block (at a given moment, the execution is exactly in a single basic block) and the evaluations of relevant program variables. The edges between the basic blocks are called *transitions*. An edge is *guarded* by a logic condition that specifies when the program execution is allowed to go from one basic block to another. The goal of model checking is to find a feasible execution *trace* (a path in the CFG) from the start node to one of the error nodes.

Programs with loops have cyclic control flow graphs, which means that some of their traces are infinite. Using unbounded lists seems like a very natural way to model program traces. Instead of truncating loops up front, we let the satisfiability solver simulate them, by effectively executing loops until the loop condition becomes `false`. Even though some traces may be infinite, the number of basic

blocks is always finite, meaning that the transition condition (i.e. the logic expression that defines all valid transitions from a given state) is also finite and can be expressed in a closed form.

4.2 Formal Definitions

Program Graph (PG) We formally introduce Program Graphs, which are a variation of Control Flow Graphs.

A PG is defined over a set of typed variables Var. We will use $Eval(Var)$ to denote the set of possible evaluations of variables, $Expr(Var)$ to denote the set of all expressions over Var (e.g., constants, integer arithmetic, "select" and "store" operations over integer arrays, and boolean expressions), and $Cond(Var)$ to denote the set of all boolean expressions over Var ($Cond(Var) \subset Expr(Var)$). A PG is then defined as a tuple:

$$PG = (\mathsf{L}, \mathsf{Act}, \mathsf{Eff}, \rightarrow, l_0, \mathsf{E})$$

L is a set of program locations (corresponding to basic blocks), l_0 is the start location ($l_0 \in \mathsf{L}$) and E is a set of error locations ($\mathsf{E} \subset \mathsf{L}$). Act is a set of actions (program statements) and function $\mathsf{Eff} : \mathsf{Act} \times Eval(Var) \mapsto Eval(Var)$ defines the effects of actions on variable evaluations. Finally, $\rightarrow: \mathsf{L} \times Cond(Var) \times Act \times \mathsf{L}$ is the conditional transition relation with side effects (i.e., actions assigned to it). This definition is very similar to the one presented in [6].

The semantics of the $\rightarrow$ relation is defined by the following rule

$$\frac{\eta \models g \quad \eta' = \mathsf{Eff}(\alpha, \eta)}{\langle l, \eta \rangle \overset{g:\alpha}{\rightarrow} \langle l', \eta' \rangle}$$

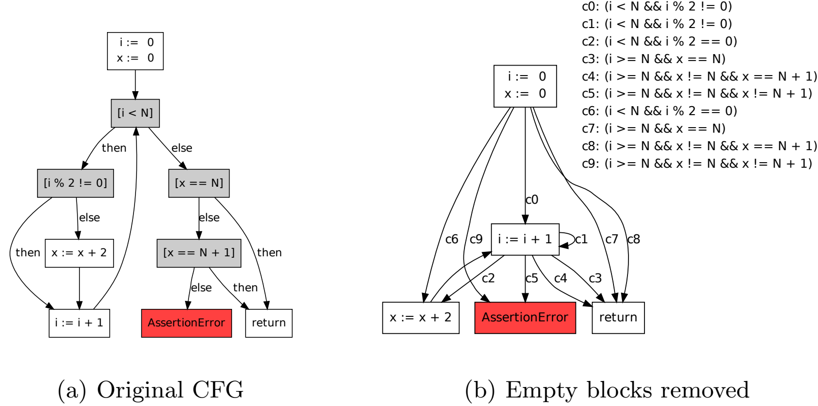
where the notation $l \overset{g:\alpha}{\rightarrow} l'$ is a shorthand for $(l, g, \alpha, l') \in \rightarrow$.

4.3 Example

We introduce a simple example that will be used throughout this section to explain optimizations and the actual translation to SMT logic. The code is shown in Figure 6, the algorithm is named `simpleWhile`, the corresponding CFG is shown in Figure 3(a). Blocks with grey background are simply branch conditions, and they do not modify the program state. The red block represents the error state. All steps presented here are fully automated.

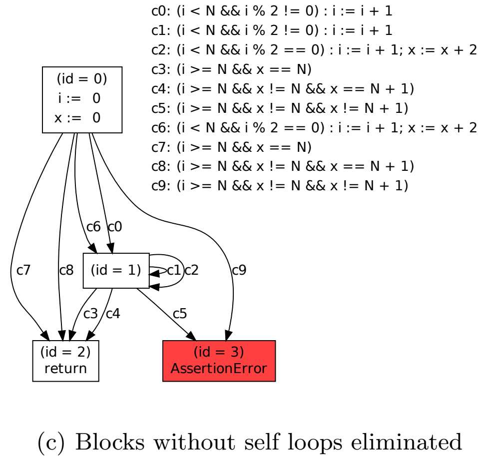
4.4 Optimizing Transformations from CFG to PG

We decided to model state changes as transitions between basic blocks, and not between single statements. This is useful because it makes the traces explored by the solver much shorter. While searching for a counter-example, the solver creates a list node for every new state it explores. If every statement caused a state transition (which is what happens in reality), then the solver would have to add a new node to the list after every variable assignment, growing the list



(a) Original CFG

(b) Empty blocks removed



(c) Blocks without self loops eliminated

Fig. 3: Control Flow Graphs for the “SimpleWhile” example

rapidly. Instead, we accumulate the effects of all statements of a basic block (by symbolically executing them) and use the resulting effect to define a single state transition. That way we enable the solver to perform more computation in every step (basically execute the entire basic block at once), thus reducing the overall number of states it has to explore, and significantly improving the solving time.

Since the solver can thus execute an entire basic block at once, we can think of the search process as a graph path finding problem: the solver is given a task of finding a path from the start block to one of the error blocks in the CFG. The search traces become sequences of basic blocks. The idea of shortening traces explored by the solver (i.e., reducing the number of basic blocks) is the basic idea behind our optimizations.

Symbolic Execution of Basic Blocks In order to arrive at the final expression for every variable at the end of a basic block, we must execute the entire basic block symbolically. Since our goal is to formulate how variables are updated when transitioning from one basic block to another, the final expressions must be in terms of symbolic variables in the previous state. For example, the effect of the following code fragment `x++; y = 2*x; x--;` is `x := (x+1)-1; y := 2*(x+1);`. Formally, we introduce the *expression update* operator $\diamond$, which takes an expression e and an action α and updates variables in e according to α:

$$e \diamond \alpha = \begin{cases} e, & \text{if } \alpha = \emptyset \\ e[v_1/e_{v_1}, \cdots, v_k/e_{v_k}], & \text{if } \alpha = v_1 := e_{v_1}, \cdots, v_k := e_{v_k} \end{cases}$$

In short, we start with an empty action α, we go through all the basic block instructions of type $v = e$, and for each of them we add $v := e \diamond \alpha$ to α (overwriting the previous assignment, if one existed).

Optimization 1: Empty Location Removal We do not want the solver to grow the list by exploring basic blocks that do not change the state. Therefore, the first optimization step takes the original CFG and removes all locations that do not have any actions that modify the program state (so-called *empty locations*). For such a location l_x, selected for removal, every incoming transition is split into several new transitions so that, after the transformation, each of the l_x's parents points to all of the l_x's successor locations. The guards of the newly created transitions are the same as the guards of the original outgoing transitions conjoined with the guard of the original incoming transition:

$$(\forall l_p \xrightarrow{g_p} l_x)\ (\forall l_x \xrightarrow{g_s} l_s)\ l_p \overset{g_p \wedge g_s}{\longrightarrow'} l_s$$

Optimization 2: Non-looping Location Elimination Here, the idea is to completely remove basic blocks that do not have any self-loops. We can split the incoming transitions, similarly to what we did in the previous step. However, we cannot simply move the actions to their parent locations, since they are not to be executed every time the parent locations are executed. The solution is to switch from CFG to PG, since program graphs allow us to associate actions with transitions instead of locations, which is exactly what we need here: we will add the actions of the location to be removed to newly created transitions.

Before this optimization step is performed, the CFG has to be converted to its corresponding PG. This can trivially be done by moving actions associated with states to their incoming transitions. Next, we iteratively keep eliminating locations that do not contain any self-loops (*non-looping locations*) until only locations with self-loops are left in the graph. Elimination of a non-looping location l_x involves three steps: (1) splitting the incoming transitions (similarly as before); (2) merging their actions; and (3) updating their guards:

$$\mathit{elim}((L, \mathit{Act}, \mathsf{Eff}, \rightarrow, l_0, E), l_x) \mapsto (L \setminus \{l_x\}, \mathit{Act}, \mathsf{Eff}, \rightarrow', l_0, E)$$

$$(\forall l_p \xrightarrow{g_1 : \alpha_1} l_x)\ (\forall l_x \xrightarrow{g_2 : \alpha_2} l_s)\ l_p \overset{g_\circ : \alpha_\circ}{\longrightarrow'} l_s \text{ , where } \alpha_\circ = \alpha_1 \circ \alpha_2,\ g_\circ = g_1 \wedge (g_2 \diamond \alpha_1)$$

We have introduced another operator, the *action merge* operator $\circ$. The idea of merging two actions α_1 and α_2 is to get a new action whose effect is going to be the same as the final effect of α_1 and α_2 when executed in that order on

any variable evaluation η: $\mathsf{Eff}(\alpha_1 \circ \alpha_2, \eta) \mapsto \mathsf{Eff}(\alpha_2, \mathsf{Eff}(\alpha_1, \eta))$. In terms of merging actions α_1 and α_2, expressions in α_2 refer to the state after α_1 has been executed, therefore, it would be incorrect to simply append α_2 to α_1. Instead, α_2 has to be updated first ($\star$ operator in the listing below) so that for each variable assignment $v_2 := e_2$ in α_2, expression e_2 is updated with respect to α_1 ($e_2 \diamond \alpha_1$). Once α_2 has been updated, the result of the merge operation is the updated α_2 appended with variable assignments in α_1 that do not already appear in it. A similar intuition holds for updating transition conditions, it is not correct to simply conjoin g_1 and g_2, instead, g_2 has to be updated first.

$$\alpha \star \beta = \begin{cases} \emptyset, & \text{if } \alpha = \emptyset \\ \{v := e_v \diamond \beta\} \cup (\alpha \setminus \{v := e_v\}) \star \beta, & \text{if } \exists (v := e_v) \in \alpha \end{cases}$$
$$\alpha_1 \circ \alpha_2 = (\alpha_1 \setminus \alpha_2) \cup (\alpha_2 \star \alpha_1)$$

Figure 3(c) shows the PG for the "Simple While" example, after all non-looping locations have been eliminated. First, the action `i:=i+1` from the state with `id=1` is moved to its incoming transitions `c0`, `c1`, `c2`, and `c6`. Next, the location with `x:=x+2` action is eliminated, and as a result, edges `c2` and `c6` are redirected and updated to include the `x:=x+2` action.

4.5 Translation of PG to SMT

Figure 4 shows the actual translation of the PG in Figure 3(c) to *initial*, *transition*, and *error* conditions, needed for the template SMT context given in Figures 1 and 2. The translation is pretty straightforward. An extra field, *stateId*, is first added to the state tuple to identify the current location. In this case, the state consists of 3 variables: $stateId, x, i$ (the variable N is constant so it is kept outside of the state tuple). The *initial_condition* is a direct representation of the state in the entry block. The *error_condition* is also easy to formulate, since all error states are explicitly known upon the CFG creation. The *transition_condition* contains two big nested *if-then-else* statements. The outer *if-then-else* has a case for every non-leaf location. Inside each such case, there is an inner *if-then-else* that has a case for each of the location's outgoing transition, where it specifies how the state is updated when that transition is taken.

Finally, we need to define the set of possible values for the input variable N (e.g., $N > 0 \wedge N \leq 10$). This additional constraint is necessary because integers are unbounded in SMT theories. Recall that this technique effectively simulates program loops inside SMT. Since the value of N influences the number of loop iterations, if a bound is not provided for N, the solver will try to simulate the loop for all possible values of N, and thus never terminate.

5 Execution of Live Sequence Charts

In this section we show how this model-checking technique can be applied to a non-trivial biological model-checking problem. We use the theory of lists to encode Live Sequence Charts and then run Z3 to analyze and execute them.

```
initial_condition ≡ head(statesList).stateId = 0 ∧ head(statesList).x = 0 ∧ head(statesList).i = 0

transition_condition

  ≡ IF head(lst).stateId = 0 THEN
        IF i < N ∧ i % 2 ≠ 0 THEN
            head(tail(lst)).stateId = 1 ∧ head(tail(lst)).i = head(lst).i + 1
        ELSE IF i < N ∧ i % 2 = 0 THEN
            head(tail(lst)).stateId = 1 ∧ head(tail(lst)).x = head(lst).x + 2 ∧ head(tail(lst)).i = head(lst).i + 1
        ELSE IF i ≥ N ∧ x = N THEN
            head(tail(lst)).stateId = 2
        ELSE IF i ≥ N ∧ x ≠ N ∧ x = N + 1 THEN
            head(tail(lst)).stateId = 2
        ELSE
            head(tail(lst)).stateId = 3
    ELSE IF head(lst).stateId = 1 THEN
        IF i < N ∧ i % 2 ≠ 0 THEN
            head(tail(lst)).stateId = 1 ∧ head(tail(lst)).i = head(lst).i + 1
        IF i < N ∧ i % 2 = 0 THEN
            head(tail(lst)).stateId = 1 ∧ head(tail(lst)).x = head(lst).x + 2 ∧ head(tail(lst)).i = head(lst).i + 1
        ELSE IF i ≥ N ∧ x = N THEN
            head(tail(lst)).stateId = 2
        ELSE IF i ≥ N ∧ x ≠ N ∧ x = N + 1 THEN
            head(tail(lst)).stateId = 2
        ELSE
            head(tail(lst)).stateId = 3

error_condition ≡ head(lst).stateId = 3
```

Fig. 4: Translation of the CFG shown in Figure 3(c) to SMT logic

5.1 Example

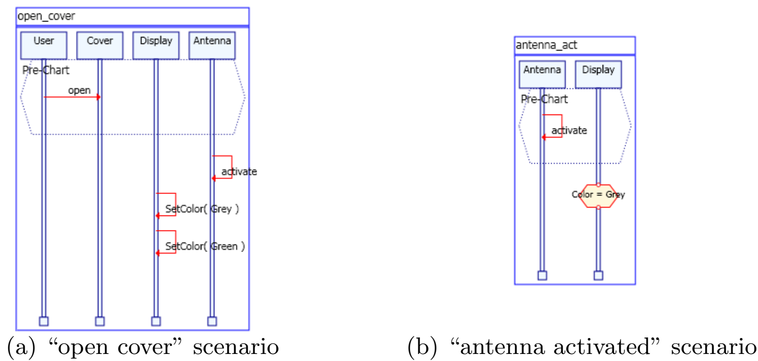
We will use an example to briefly introduce LSCs and their semantics. Figure 5(a) shows the specification of the interaction between a cell phone and the user. A single LSC consists of a number of *Instances* passing messages between them. Instances either belong to the *System* or the *Environment*. Every Instance has an associated *timeline* (represented as vertical bars) which is used to impose the ordering between messages. The upper portion of the chart (bordered with a dotted line) is called the *Pre-Chart*, whereas the rest of the chart is called the *chart body*. Every chart is initially *inactive*. It becomes *active* when its Pre-Chart is satisfied, i.e., when messages that appear in the Pre-Chart occur in the specified order. The semantics of LSCs require that once a chart becomes active, it must finish its execution according to the specification in its body, when it becomes *closed*. The chart specifies only partial ordering of the message occurrences: only messages that have a common timeline as either source or target must happen in the given ordering; messages that do not have a timeline in common may appear in an arbitrary order.

In terms of the example in Figure 5(a), once the chart becomes active, as a result of `open` occuring, there are 3 possible *valid* executions: (1) `SetColor(Grey), SetColor(Green), activate`, (2) `SetColor(Grey), activate, SetColor(Green)`, and (3) `activate, SetColor(Grey), SetColor(Green)`. Note that it is not allowed that message `SetColor(Green)` appears before `SetColor(Grey)`, that would be considered as an immediate violation of the specification. Also note that it is allowed that some other messages, not shown in this chart are sent at any point during the execution of this chart. The chart's body specifies only messages that must happen, and partial ordering between them, it does not forbid other messages. This way, a formal contract is established saying that every time the user opens the cover (message `open` is sent from `User` to `Cover`), the cell phone must respond as specified in the chart's body.

5.2 Motivation

Single step is defined as a single message sent by the System that does not cause an immediate violation. *Super step* is a sequence of messages that drives all active charts to their completion, without causing any violations. It is allowed for a super step to activate some new charts along the way, but at the end of it, no charts must be active. For example, consider another scenario given in Figure 5(b). The message `activate` activates the `antenna_act` chart. Its body contains a single conditional element that states that the color must be `Grey` after the chart is activated. Adding this additional scenario rules out the first of the three valid executions of the "open cover" scenario given above.

We describe our solution for encoding of LSCs into the logic of SMT with the theory of lists, which allows for using the Z3 SMT solver to automatically find all valid super steps from a given point in the execution of the system. Here we illustrate the applicability and usefulness of our technique to this problem; a more detailed discussion and formal translation is not presented due to space limitations and will be reported in a future paper.



(a) "open cover" scenario (b) "antenna activated" scenario

Fig. 5: The cell phone LSC example

5.3 Solution

We formulate the problem of finding a super step as a model-checking problem. For every Instance, we keep an integer variable to keep track of its *location* (a point on its timeline) in the current state of the execution. We also maintain variables for object properties (e.g., `Color` as in the example) and a single variable for the message sent by the system in the current step. The initial state is explicitly given and consists of current locations of all instances and evaluation of all properties. In the transition constraint, we let the solver non-deterministically pick a message to be sent by the system and based on that decision we specify how the rest of the state should be updated. We assert that the chosen message must be *enabled* at the current step (i.e., that at least one Instance is at a location where this message can be sent from) and that it must not cause any violations in other charts. The safety property that we want the solver to prove is that the state where all charts are closed can never be reached from the initial state.

If the solver proves this property, that means that no valid super step exists. Otherwise, the solver will come back with a counter-example that contains a list of state changes, which lets us decode which message is sent at each step.

Formulating this problem using the theory of lists seems very convenient, since the number of steps needed to find a counter-example is not known in advance. We analyzed several models of biological systems [2] and were able to find valid super steps for systems with more than ten charts within seconds.

6 Evaluation and Results

We implemented a fully automated prototype model checker for Java programs to evaluate the idea of using the SMT theory of lists to model program traces. Currently, we support only a subset of Java programs. We used this tool to verify the correctness of several algorithms. We also applied this technique to solve the *Rush Hour* puzzle [1]. All experiments were conducted on a 64-bit Intel Core Duo CPU @2.4GHz box, with 4GB of RAM, running 32-bit Windows Vista.

Verifying Simple Algorithms We used this technique to verify the "Simple-While" algorithm, two sorting algorithms, and the integer square root algorithm from Carroll Morgan's book *Programming with Specifications* [22] (Figure 6). We present the comparison of verification times between the optimized and non-optimized translation for several different bounds. We compare our tool to a representative tool from the bounded model-checking category – JForge [16,26], and a finite model checker that doesn't require explicit loop unrolling – Java PathFinder [25]. The results are shown in Figure 7. The "Related Work" section describes these tools in detail and discusses the obtained results.

Non-monotonicity of some of the graphs in Figure 7 can be explained by the nature of satisfiability solvers. The solving time is highly dependent on internal heuristics (e.g., [20,24]), so it can happen that a larger problem is solved faster simply because the heuristics worked better (for example, it happened that a large portion of the search space was pruned early on).

Finally, this approach performs quite efficiently when a counter-example exists. For all of the presented benchmarks, our tool was able to find different (manually introduced) bugs within seconds.

Solving the Rush Hour Puzzle *RushHour* is a well known puzzle where the goal is to get the designated car (the red car in Figure 9) out of the traffic jam. This puzzle is easily expressible as a model-checking problem: the initial state is the given configuration of cars at the starting point, the transition function constrains the allowed movements of the cars so that they do not crash or go over each other, and the safety property is that the red car can never reach the far right side of the stage. If we find a counter-example to this model-checking problem, we have found the way to get the red car out of the jam.

We took several puzzles from [1] and compared the execution times of the two approaches: bounded (the case when we know the optimal number of steps) and unbounded with lists (Figure 8). SMT solvers are optimized to deal with large flat formulas, so the fact that the bounded encoding currently performs better

```
void simpleWhile(int N) {
 int x = 0, i = 0;
 while (i < N) {
  if (i % 2 == 0)
    x += 2;
  i++;
 }
 assert x == N || x == N + 1;
}
```

```
void selectSort(int[] a, int N) {
 for (int j=0; j<N-1; j++) {
  int min = j;
  for (int i=j+1; i < N; i++)
   if (a[min] > a[i]) min = i;
  int t = a[j]; a[j] = a[min]; a[min] = t;
 }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

```
void bubbleSort(int[] a, int N) {
 for (int j=0; j<N-1; j++)
  for (int i=0; i<N-j-1; i++)
   if (a[i] > a[i+1]) {
     int t = a[i];
     a[i] = a[i+1];
     a[i+1] = t;
   }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

```
int intSqRoot(int N) {
  int r = 1, q = N;
  while (r+1 < q) {
    int p = (r+q) / 2;
    if (N < p*p) q = p;
    else r = p;
  }
  assert r*r <= N && (r+1)*(r+1)>N;
  return r;
}
```

Fig. 6: Benchmark Algorithms

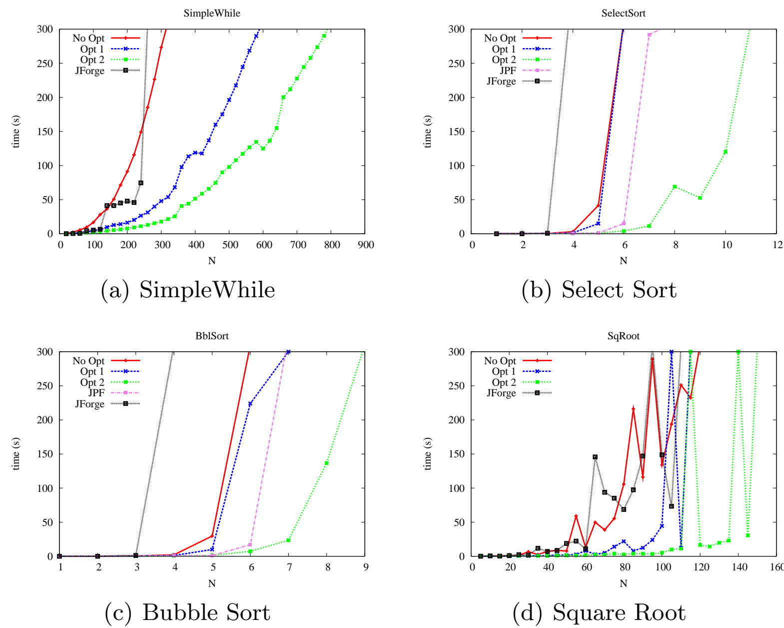
does not come as a surprise. However, we were able to solve the most difficult puzzles (e.g., Jam 38-40 require more than thirty steps) within a minute.

This problem is quite different from the software model-checking problems, because at every step, there are typically several available valid moves, so at every step, the solver has to non-deterministically decide which move to take in order to finally reach an error state (this never happens in software model checking if programs are deterministic). This puzzle is a typical example of how this technique can be used to solve planning problems without bounding the number of steps in advance.

One limitation of our current implementation is that it is not able to prove it if the solution does not exist. The solver gets stuck exploring the same states over and over again (e.g., moving the red car back and forth between the neighboring cells). However, if a solution exists, this problem is not manifested. Also note that this does not happen in software model checking if the target program always terminates. An obvious solution is to forbid the same states to appear in the `states` list. This additional constraint is expressible in SMT logic, but in practice it does not perform that well. Instead, we believe that the SMT solver could be tweaked so that it internally knows that while building the `states` list it should never include the same state twice in a single search path. It would be very efficient to implement this inside the solver, because the state is represented explicitly inside list elements, so it would be easy to compare states for equality.

7 Related Work

Model checking was originally defined as a technique for proving properties about Finite State Machines (FSM) [12]. The pioneering tools had used an explicit representation of the entire state graph, which led to what is known as the *state explosion problem*. To mitigate that problem, Binary Decision Diagrams (BDD) were introduced by McMillan [14] to symbolically represent a set of states with a



(a) SimpleWhile (b) Select Sort

(c) Bubble Sort (d) Square Root

Fig. 7: Benchmark Results

single propositional logic formula. Both of these techniques used a custom search algorithm to explore paths in the FSM. Infinite traces were supported by computing a *fixpoint*, i.e., not visiting the same state twice on the same search path. The growing popularity and efficiency of satisfiability solvers had influenced another branch of model checking, called *Bounded Model Checking* [8,9,11], which significantly improved the scalability of model checking. The idea was to bound the traces by unrolling the FSM for some number of times k. As a result, the whole problem could be formulated as a single propositional formula, solvable by off-the-shelf SAT solvers. On the other side, Counter-Example Guided Abstraction Refinement [13] was developed to deal with infinite state machines. In comparison, our approach lies somewhere between bounded and unbounded

	B	U
Jam 25	1.20s	1.88s
Jam 30	1.21s	2.17s
Jam 38	4.47s	36.6s
Jam 39	1.90s	14.66s
Jam 40	6.31s	17.89s

Fig. 8: RushHour benchmark (B – Bounded, U – unbounded)

Fig. 9: RushHour instance

finite state model checking: in many cases, we achieve benefits of the unbounded method, but in some, our tool cannot prove the absence of counter-examples.

JForge is a bounded software model checker that uses SAT. It requires the user to bound the program input space by specifying the bit-width for integers, in addition to providing the number of loop unrollings. In all benchmarks, we used the minimal bit-width needed to represent the bound N, and an appropriate number of loop unrollings needed to verify the code for the given input size. JForge enumerates all integers within the given bit-width so that it has the explicit representation of the whole universe. That turns out to be the reason why JForge does not perform as well as our tool in these benchmarks.

Alloy [17] is a bounded model finder that can be used to search for traces (sequences of events) that satisfy certain logic property, but it also requires that the number of events is specified in advance.

JPF [25] is an extensible plaform for running model checkers for Java programs. The explicit-state version of JPF directly executes the program on all possible inputs, whereas we translate the program into logic and formulate a satisfiability problem. We present results for JPF only for the two sorting algorithms. In the other two examples, JPF is a clear winner. However, the sorting examples show the case where the ability of our tool to symbolically represent array elements brings a significant advantage. To verify the sorting algorithms using JPF on arrays of size exactly n, we ran the algorithm on all possible arrays of size n whose elements are between 1 and n, which turned out to be very expensive in terms of both memory and time. Symbolic JPF [4] can treat the variables symbolically, but it currently does not support arrays.

Armando et al. [5] present a bounded software model-checking technique (requires explicit loop unrolling) based on SMT, and report significant improvement over the traditional SAT-based technique. Other techniques for unbounded model checking with satisfiability solving (e.g., [18, 21]) iteratively invoke the solver until they reach a fixpoint, whereas our approach translates the whole problem into a single formula.

8 Conclusion

We have presented a novel technique for finite-state unbounded model checking using the theory of lists and satisfiability solving. Our technique is a finite-state technique, in the sense that it requires explicit bounds on certain parts of the input state (e.g., those that influence the length of the state machine traces). On the other hand, it can prove properties for infinite-state systems, as shown for the "sorting" examples. We have shown the generic pattern for solving model-checking problems, and also provided detailed explanation of how it can be applied to software model checking in particular. The results of the comparison with some of the existing tools for software model checking seem promising. The applicability of this method to analyzing and executing scenario-based models in the form of Live Sequence Charts seems to have a stong potential and will enable efficiently supporting a larger subset of the LSC language including arithmetic operations that are more natural to handle using SMT solvers.

References

1. Rush Hour Puzzle – http://www.puzzles.com/products/rushhour.htm.
2. Microsoft Research Cambridge, Synthesizing Biological Theories, 2011. `http://research.microsoft.com/SBT/`.
3. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. August 2006.
4. S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *TACAS*, 2007.
5. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, 11(1), 2009.
6. C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. 2008.
7. C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
8. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS*, 1999.
9. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. In *Formal Methods in System Design*, 2001.
10. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.
11. E. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. *DAC*, 2003.
12. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 1986.
13. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, 2000.
14. E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. In *DAC*, 1993.
15. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *Formal Methods in System Design*, 1998.
16. G. Dennis. *A Relational Framework for Bounded Program Verification*. PhD thesis, Massachusetts Institute of Technology, 2009. Advised by Daniel Jackson.
17. D. Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.
18. H.-J. Kang and I.-C. Park. SAT-based unbounded symbolic model checking. In *DAC*, 2003.
19. H. Kugler and I. Segall. Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.
20. J. Marques-silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA*, 1999.
21. K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *CAV*, 2002.
22. C. Morgan. *Programming from specifications*. 1990.
23. L. D. Moura and N. Bjrner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
24. R. Piskac, L. Moura, and N. Bjørner. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. *J. Autom. Reason.*, 44:401–424, April 2010.
25. W. Visser, K. Havelund, and G. Brat. Model Checking Programs. In *ASE*, 2000.
26. K. Yessenov. A light-weight specification language for bounded program verification. Master's thesis, May 2009. Advised by Daniel Jackson.