# Composite Model Checking: Verification with Type-Specific Symbolic Representations*

Tevfik Bultan
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA

Richard Gerber
Department of Computer Science
University of Maryland
College Park, MD 20742, USA

Christopher League
Department of Computer Science
Yale University
New Haven, CT 06520, USA

**Abstract**

In recent years, there has been a surge of progress in automated verification methods based on state exploration. In areas like hardware design, these technologies are rapidly augmenting key phases of testing and validation. To date, one of the most successful of these methods has been symbolic model checking, in which large finite-state machines are encoded into compact data structures such as binary decision diagrams (BDDs) – and are then checked for safety and liveness properties.

However, these techniques have not realized the same success on software systems. One limitation is their inability to deal with infinite-state programs – even those with a single unbounded integer. A second problem is that of finding efficient representations for various variable types. We recently proposed a model checker for integer-based systems that uses arithmetic constraints as the underlying state representation. While this approach easily verified some subtle, infinite-state concurrency problems, it proved inefficient in its treatment of boolean and (unordered) enumerated types – which are not efficiently representable using arithmetic constraints.

In this paper we present a new technique that combines the strengths of both BDD and arithmetic constraint representations. Our composite model merges multiple type-specific symbolic representations in a single model checker. A system's transitions and fixpoint computations are encoded using both BDD (for boolean and enumerated types), and arithmetic constraint (for integers) representations, where the choice depends on the variable types. Our composite model checking strategy can be extended to other symbolic representations provided that they support operations such as intersection, union, complement, equivalence checking and relational image computations. We also present conservative approximation techniques for composite representations to address the undecidability of model checking on infinite-state systems.

We demonstrate the effectiveness of our approach by analyzing two example systems which include a mixture of booleans, integers and enumerated types. One of them is a requirements specification for the control software of a nuclear reactor's cooling system, and the other one is a transport protocol specification.

*This research is supported in part by ONR grant N00014-94-10228 and NSF CCR-9619808.

## 1 Introduction

Symbolic model checking has proved to be a highly successful technique for verifying hardware systems [23, 51]. Recently, it has been used for analyzing software specifications with encouraging results [4, 6, 7, 9, 25, 44, 45, 57]. The success of model checking (especially in hardware verification) has been partially due to the advent of innovative data structures like Binary Decision Diagrams (BDDs), which can encode boolean functions in a highly compact format [14]. The main idea in BDD based model checking is to represent sets of system states and transitions as boolean formulas, and manipulate them efficiently using the BDD data structure [23, 51].

An important property of the BDD data structure is that it supports operations such as intersection, union, complement, equivalence checking and existential quantifier elimination (used to implement relational image computations) efficiently – which also happen to be the main operations required for model checking. However, an efficient encoding for boolean domains may not be as useful for analyzing software systems as it is for analyzing hardware systems. Unlike hardware systems, software systems tend to include various variable types with different characteristics. It is not likely for the BDD data structure to provide an efficient encoding for every variable type in a software system. For example, BDD based model checkers can be very inefficient in representing arithmetic constraints [25].

Another shortcoming of the BDD representation is its inability to encode infinite domains. Without abstraction, BDDs cannot be used for analyzing infinite-state systems – even those with just one unbounded integer. BDDs encode all underlying datatypes as boolean variables; hence all BDD-based model checkers inherently require the underlying types to be bounded.

In recent years several new symbolic representations have been proposed. For example, HyTech, a symbolic model checker for hybrid systems, encodes real domains using linear constraints on real variables [1, 2]. Automata based representations such as Queue content Decision Diagrams (QDDs), are effective in encoding communication channels and are used in analyzing protocols [11, 12]. Recently, we proposed a model checker for general integer based systems, which uses Presburger arithmetic constraints (integer arithmetic without multiplication) as its underlying state representation [19]. Each of these symbolic representations support the operations required for model checking and they all provide efficient encodings for certain variable types. However, they fail to be efficient for all variable types.

As with BDDs for boolean variables, Presburger constraints can compactly represent huge (even unbounded) sets of integer states over multiple dimensions. Specifically, our Presburger model checker represents sets of state-valuations using unions of convex polytopes, each of which is formed by linear constraints over the system's variables. And like BDDs, this representation also affords efficient techniques for carrying out pertinent set-theoretic operations (we use a Presburger solver called the Omega library [48, 53] for this purpose).

While many model checking queries are undecidable for general infinite-state systems [13, 36], we often appeal to conservative approximation techniques which guarantee convergence. With this approach we were able to verify some nontrivial infinite-state programs from the concurrency literature using our Presburger model checker. These programs (and their associated formulas) are the type usually analyzed with hand proofs, due to the subtle way their infinite-state variables influence the control flow.

However, our Presburger formula encoding proved ill-suited for handling boolean and (unordered) enumerated types. When all state sets are represented as Presburger constraint expressions, all boolean variables are mapped to integers – which ends up being extremely wasteful. Since programs usually include many such variables, and since model checkers often generate large sets of states, this wastage quickly adds up to a major obstacle. Also, while both boolean and Presburger logics afford compact encodings for sets of valuations formed over their operators, there is no natural mapping between them which preserves this compactness. Consider, for example, a set of linear constraints over $n$ integers, which compactly describes all states lying within some $n$-dimensional polytope. The benefit of this encoding extends directly from the expressiveness of the arithmetic inequality operators – which are useless over 2-valued domains like booleans, and marginally less so for small enumerated types like $\{red, green, blue\}$.

Our solution to this problem is a general framework for combining multiple type-specific symbolic representations in a single composite model checker. Our current prototype demonstrates the effectiveness of this approach by combining BDD and Presburger arithmetic encodings in a single composite representation [18, 17]. Our strategy can be extended to include other symbolic encodings as well – provided that they support operations such as intersection, union, complement, equivalence checking and relational image computation. All symbolic representations mentioned above can be included in our composite model – with the result being a general-purpose model checker.

The key to our framework rests on some fundamental observations. First, we map each variable in the input system to a symbolic representation (which we call the symbolic encoding type of the variable). For example, boolean and enumerated variables can be mapped to BDD encoding, and integers can be mapped to Presburger formula encoding. Then, given a system whose specification excludes (1) arbitrary functions over mixed variable types, and (2) type-coersions (e.g., to allow booleans in arithmetic operators), we (a) orthogonally partition each atomic event into a set of conjuncts, where each conjunct specifies the effect of the event on the variables represented by a single encoding type (e.g., one conjunct specifies the effect of the event on variables encoded using BDDs, whereas another conjunct specifies the effects of the event on variables encoded using Presburger formulas); (b) store the sets of system states as a disjunction of conjunctively partitioned type specific representations (e.g., a disjunct may consist of a boolean formula stored as a BDD representing the states of boolean and enumerated variables, and a Presburger formula representing the states of integer variables); (c) compute the pre- and post-conditions independently for each encoding type exploiting this conjunctive partitioning; and also (d) use semantically sound rules in combining the intersection, union, complement and equivalence checking computations for different symbolic representations. The key observation here is the fact that conjunctive partitioning of the atomic events allows pre- and post-condition computations to distribute over the encoding types. We also extend the conservative approximation techniques such as widening to composite representations by combining type-specific widening operators.

We already implemented a prototype toolset based on these ideas using our own BDD library and the Omega Library (used for manipulating Presburger formulas) [18, 17]. Our prototype toolset is structured in a layered class hierarchy. The lowest layer consists of libraries for manipulating various symbolic representations such as BDDs and arithmetic constraints. At the next level we have our composite-model library, which handles operations over mixed-type constraints (e.g., equivalence check, intersection, etc.); in turn, these operations invoke their relevant type-specific counterparts in the lower level to help carry out the desired effect. The model checker's class library, which imports the composite-model operations, and exports functions for forward and backward fixpoint computations is on top of the composite-model library. Our composite approach will allow us to expand to additional symbolic representations such as automata in the future.

We demonstrate the effectiveness of our approach by analyzing two example systems using our composite model checker: (1) a requirements specification for the control software of a nuclear reactor's cooling system, and (2) a transport protocol specification.

The first system analyzed is an "enhanced" version of a known SCR specification [9, 31, 40]. The underlying model contains a good mixture of booleans, unbounded integers and enumerated types, each of which retain their exact semantic interpretation in our composite model checker. Specifically, this means that during automated analysis checks, important integer values get propagated through the system's transitions – along with the relevant (bounded) values for modes and conditions. This allows us to fully interpret integer-valued functions and predicates appearing in an SCR specification; hence, the model checker automatically deduces inferences such as:

$$(\mathrm{temp} > \mathrm{High} \;\longleftrightarrow\; \mathrm{Alarm}) \;\Longrightarrow\; ((\exists\ \mathrm{x} < 0\ .\ \mathrm{temp} + \mathrm{x} = \mathrm{High}) \rightarrow \mathrm{Alarm}).$$

With this capability, we need not map SCR integer predicates to boolean literals (e.g., as in [4, 6, 7, 24]); moreover, for our model checker this type of inference comes at no additional cost. Finally, our framework allows us to test mixed integer-boolean environmental hypotheses. These tests involve queries on possible feedback relationships between values conveyed to actuators (i.e., "controlled variables") and subsequent samples on sensors (i.e., "monitored variables").

The second system we analyze is a one-directional transport protocol based on TCP. Although abstractions and simplifications have been made, much of TCP's complexity remains. We analyze various properties of the protocol, specifically: handshaking for call setup and take-down, as well as reliable data-transfer using sliding-window protocol [55] over an unreliable channel, which can delay, duplicate, lose or reorder messages to an unlimited extent. Moreover, the protocol's send and receive window sizes are not specified in advance; rather, they are represented as symbolic constants. There are several points to the exercise: (1) the properties examined rely on multiple state changes, triggered by conditions over both boolean and integer-valued variables; (2) we verify these properties without bounding window sizes or sequence numbers; hence, correctness is assured for any implementation of the protocol; (3) automated verification was only possible using composite models, as well as conservative approximation techniques based on their operators.

Both of the systems we analyze have a mixture of integer and boolean-valued variables. Hence, they were not amenable to verification by a BDD-based model checker alone; moreover, their size (and the number of boolean variables) made pure Presburger model checking infeasible. In this paper we show how these systems were automatically verified using our composite model checking approach.

The rest of the paper is organized as follows. In Section 2, we discuss the related work. In Section 3, we present the model checking technique and its extension to symbolic representations. Section 4 presents our composite model checking approach. In Section 5, we discuss using conservative analysis techniques on composite models. Our prototype composite model checker is described in Section 6. Sections 7 and 8 present automated analysis of a requirements specification for a control software and a transport protocol, respectively. In Section 9, we summarize our results and point out future research directions.

## 2 Related Work

We first summarize recent results in model checking software specifications. Then we discuss various symbolic representations which have been proposed recently, and finally we discuss conservative approximation techniques.

**Model Checking Software Specifications:** There has been significant work in using model checking to verify tabular-style SCR requirements. In [7] Atlee and Gannon map queries about SCR mode-transition tables to the MCB model checker, which uses explicit state enumeration as its underlying representation. In [54] Sreemani and Atlee improve this technique by using the SMV model checker for analyzing the software requirements of the A-7E aircraft. Since SMV [51] is a symbolic, BDD-based model checker, it generates more efficient encodings of the SCR state space, which makes it possible to check larger systems. The same tool is also used to analyze parts of the RSML specification of the Traffic Alert and Collision Avoidance System II (TCAS II) [4, 25]. The main difficulty in using SMV for checking requirements specifications seems to be the fact that every variable is represented in the same symbolic format, namely BDDs. This can easily result in inefficient encodings of arithmetic constraints [25]; moreover, the sizes of the resulting BDD representations (not to mention their inherent finiteness) often makes verification untenable, at least without human-guided abstractions.

In [24], Chan *et al.* report that representing integers using bitwise BDD representations is not efficient when the input system contains non-linear constraints. They present a technique in which (both linear and non-linear) constraints are mapped to BDD variables (similar representations were also used in [6, 7]). These constraints are used for specifying guarding conditions of transitions. A constraint solver is used during model checking computations (in conjunction with SMV) to prune infeasible combinations of these constraints. Although this technique is capable of handling non-linear constraints, it is restricted to systems where transitions are either *data-memoryless* (i.e., next state value of a data variable does not depend on its current state value), or *data-invariant* (i.e., data variables remain unchanged). Hence, even a transition which increments a variable (i.e., $x' = x + 1$) is ruled out. It is reported in [24] that this restriction is partly motivated by the semantics of RSML, and it allows modeling of a significant portion of TCAS II system.

Nitpick is a finite state model checker which uses explicit state enumeration for analyzing a subset of the Z specification language [43]. Since Nitpick restricts the variable domains (to make the state enumeration feasible) it does not guarantee that a property holds, rather it searches for counter-example behaviors to falsify the given property. It has been used in analyzing a design feature of Microsoft Word[44] and properties of mobile IP [45].

In [9] Bharadwaj and Heitmeyer used the SPIN model checker to analyze behaviors of SCR specifications as a whole, including all conditions and events, as well as mode transition tables. But since SPIN relies on a finite-state model (like SMV), it can not check systems with unbounded variables. And indeed, in most of the abovementioned work, abstraction techniques were used to simplify the systems being analyzed. Although we believe that abstraction is bound to play a key role in any automated analysis technique, in some cases it can be avoided by using symbolic representations *which can capture the inherent properties of the underlying types*. Under certain situations, using arithmetic constraints provides an opportunity to investigate general properties of integer variables, without abstracting their behavior or bounding their domains.

**Symbolic Representations for Model Checking:** Recently, linear constraints on real variables have been used as a symbolic representation in analyzing hybrid and real-time systems [2, 1, 3, 39]. A hybrid system is a discrete program which interacts with an analog environment. It is modeled as an automaton that has continuously evolving variables according to some dynamical laws. HyTech, a model checker for hybrid systems, represents states of continuous variables using linear constraints [3]. However discrete variables are represented using explicit state enumeration. This representation may not scale to large numbers of discrete variables or to large discrete domains.

Queue content Decision Diagrams (QDDs) are proposed for encoding sets of queue-configurations [11, 12], and are used to carry out reachability queries on communicating state machines. Queue configurations are modeled via deterministic finite automata, where the language accepted by the automata encodes the set of queue-contents. QBDDs extend QDDs, by combining QDD representation with BDD encoding [37]. QBDDs have limited expressiveness for infinite sets. They are more appropriate for encoding bounded queues.

The MONA tool is a library for manipulating automata-based representations of formulas in WS1S (Weak Second order theory of One Successor) [8, 10, 41, 50]. WS1S corresponds to regular languages (languages accepted by finite automata) and subsumes a fragment of arithmetic (including Presburger arithmetic). The name MONA comes from monadic second order logic on finite strings, which can also be represented by the MONA tool using a different semantics for the automata representation. The tool is based on some efficient algorithms for minimizing automata that uses BDDs to represent transition functions in compressed form. QBDD representation corresponds to a special case of the automata-based representation used in MONA [10].

Several symbolic representations have been proposed for modeling functions over boolean variables with integer ranges, including Multi-Terminal Binary Decision Diagrams (MTBDDs) [26], Binary Moment Diagrams (BMDs) [15], and their generalization Hybrid Decision Diagrams (HDDs) [26, 27]. These are especially useful for datapath circuit verification in hardware design, since they can encode functions that map boolean vectors to integers [27].

Each symbolic representation mentioned above can provide efficient encodings for certain variable types. And they all support the functionality required for symbolic model checking. In [49], such an encoding is called an *adequate language* for model checking. Our techniques for combining different symbolic encodings can be extended to all these representations (i.e., all adequate languages) – with the result being a general-purpose model checker.

**Conservative Approximation Techniques:** Our conservative approximation techniques are based on the ideas from from abstract interpretation [32]; specifically, we use approximation methods such as widening first developed for that domain. A convex widening operator for polyhedral representations was originally presented in [33]. It has been used in delay analysis in synchronous programs [38] and in reachability analysis of linear hybrid systems [39]. Recently, we proposed a generalized multi-polyhedra widening operator in conjunction with our Presburger model checker [19].

Our recursive approximation technique for temporal properties with nested temporal operators is similar to the one used in [47] where a temporal property expressed in $\mu$-calculus is computed conservatively using two abstractions of the same program. One abstraction overestimates the behavior of the program and is used for computing universal properties. Another abstraction that underestimates the behavior of the program is used for existential properties. Together they can be used to conservatively approximate any $\mu$-calculus property. In [35] these ideas are extended using the abstract interpretation framework. Similar methods are also used in [34, 52]. Our approach is based on approximating the fixpoint computations using the widening technique instead of approximating the program as a whole using abstractions. It is possible to use both of these techniques together which we see as a promising direction for future research.

# 3 Model Checking

In model checking, the system to be analyzed is represented as a transition system $TS = (S, I, R)$ with a set of states $S$, a set of initial states $I \subseteq S$, and a transition relation $R \subseteq S \times S$. The transition system model is never explicitly generated in symbolic model checking. For example BDD based model checkers represent transition relation $R$ as a set of boolean logic formulas.

A popular temporal logic for specifying temporal properties of transition systems is Computation Tree Logic (CTL) [28] which consists of a set of temporal operators (the "quantified-next-state" operators EX and AX, the "quantified-until" operators EU and AU, the "quantified-invariant" operators EG and AG, and the "quantified-eventuality" operators EF and AF) for specifying temporal properties, and a set of basic properties $BP$ for specifying properties of states. The semantics of temporal operators are defined as:

- A state $s_0$ satisfies AX $p$ (EX $p$) if and only if for all (some) maximal paths $(s_0, s_1, s_2, \ldots)$ with length $\geq 2$, $s_1$ satisfies $p$.
- A state $s_0$ satisfies $p$ AU $q$ ($p$ EU $q$) if and only if for all (some) maximal paths $(s_0, s_1, s_2, \ldots)$ there exists a state $s_i$ such that, $s_i$ satisfies $q$ and for all $j < i$, $s_j$ satisfies $p$.

A maximal path is one which is either infinite, or ends with a state that has no successors. The remaining temporal operators are defined by the following equivalences:

$$\begin{array}{lcllcl} \mathrm{EF} f & \equiv & \textbf{true} \ \mathrm{EU} \ f & \qquad \mathrm{EG} f & \equiv & \neg(\textbf{true} \ \mathrm{AU} \ \neg f) \\ \mathrm{AF} f & \equiv & \textbf{true} \ \mathrm{AU} \ f & \qquad \mathrm{AG} f & \equiv & \neg(\textbf{true} \ \mathrm{EU} \ \neg f) \end{array}$$

If all the initial states of a transition system satisfy a temporal property, then we say that the system itself satisfies the property. In other words (using the truth set interpretation of the temporal formulas) $TS \models p$ if and only if $I \subseteq p$.

Finding a bug in a system is equally valuable (if not more valuable) as verifying it. Finding a bug corresponds to finding an initial state $s \in I$ which does not satisfy the temporal property, i.e., $s \in I \cap \neg p$. It is possible to generate a counter-example execution path, which demonstrates violation of property $p$, starting from such a state [29].

Then, our goal in analyzing a system $TS = (S, I, R)$ and a temporal property $p$ is :

- Either to prove that the system $TS$ satisfies the property $p$ by showing that $I \subseteq p$.

- Or to demonstrate a bug by finding a state $s \in I \cap \neg p$, and generating a counter-example path starting from $s$.

Assume that there exists a representation for sets of states which supports tests for equivalence and membership. Then, if we can represent the truth set of the temporal property $p$, and the set of initial states $I$ using this representation, we can check the two conditions listed above. If the state space is finite, explicit state enumeration would be one such representation. Note that as the state spaces of the programs grow, explicit state enumeration will become more expensive since the size of this representation is linearly related to the number of states in the set it represents. Unfortunately, state space of programs increase exponentially with the number of variables and concurrent components. This state space explosion problem makes a naive implementation of the explicit state enumeration infeasible.

Another approach is to use a *symbolic representation* for encoding sets of states. For example, a logic formula which is semantically interpreted as a set of states, can be used as a symbolic representation. Boolean logic formulas (stored using the BDD data structure) are the most common symbolic representation used in model checking [23, 51]. Recently, we used Presburger arithmetic (integer arithmetic without multiplication) formulas for the same purpose [19, 20].

Model checking procedures use state space exploration to compute the set of states which satisfy a temporal property. Fixpoints corresponding to truth sets of temporal formulas can be computed by iteratively aggregating states using pre-condition computations. Temporal properties which require more than one fixpoint computation can be computed recursively starting from the inner fixpoints and propagating the partial results to the outer fixpoints. Below we demonstrate how to implement a model checker, and discuss the functionality that a symbolic representation has to support to be included in such a procedure.

### 3.1 Fixpoint Computations

All temporal properties in CTL can be expressed using least fixpoints. For example truth set of the temporal formula $p$ EU $q$ corresponds to [5]:

$$p \text{ EU } q \equiv \mu x \,.\, q \ \vee\ (p \ \wedge\ \text{EX } x).$$

Note that $q \ \vee\ (p \ \wedge\ \text{EX } x)$ defines a functional from $2^S$ to $2^S$. For example, given a set of states $x$, the functional $q \vee (p \wedge \text{EX } x)$ maps $x$ to a set of states $y$ such that, a state $s$ is in set $y$ if and only if $s$ satisfies the property $q$ or $s$ satisfies the property $p$ and there exists a next state of $s$ which is included in the set $x$. Then the truth set of the temporal property $p$ EU $q$ is the smallest set of states which is a fixpoint of that functional. We can represent the property $p$ AU $q$ similarly as a least fixpoint:

$$p \text{ AU } q \equiv \mu x \,.\, q \ \vee\ (p \ \wedge\ \text{EX } x \ \wedge\ \text{AX } x).$$

Least fixpoint of a monotonic functional can be computed by starting from the bottom of the chain (i.e., **false** $\equiv \emptyset$) and by iteratively applying the functional [56]. For example, consider the property EF $p$. A state $s$ satisfies EF $p$ if there exists a path starting from $s$ which has a state that satisfies $p$. Based on the properties discussed above we have

$$\text{EF } p \ \equiv\ \textbf{true} \text{ EU } p \ \equiv\ \mu x \,.\, p \ \vee\ (\textbf{true} \ \wedge\ \text{EX } x) \ \equiv\ \mu x \,.\, p \ \vee\ \text{EX } x$$

i.e., EF $p$ is equivalent to the least fixpoint of the functional $\mathcal{F}\, x \equiv\ p \ \vee\ \text{EX } x$. We can compute the truth

```
Procedure Evaluate(f)
 Case
  IsEvaluated(f)   :  Return(f)
  f = ¬f1          :  Return(Not(Evaluate(f1)))
  f = f1 ∧ f2      :  Return(And(Evaluate(f1),Evaluate(f2)))
  f = f1 ∨ f2      :  Return(Or(Evaluate(f1),Evaluate(f2)))
  f = EX f1        :  Return(EX(Evaluate(f1)))
  f = AX f1        :  Return(Not(EX(Not(Evaluate(f1)))))
  f = f1 EU f2     :  q0 ≡ false    q_{i+1} ≡ Evaluate(f2 ∨ (f1 ∧ EX qi))
                      Return(qn) when qn ≡ q_{n+1}
  f = f1 AU f2     :  q0 ≡ false    q_{i+1} ≡ Evaluate(f2 ∨ (f1 ∧ EX qi ∧ AX qi))
                      Return(qn) when qn ≡ q_{n+1}
```

Figure 1: A Model Checker for CTL

set of EF $p$ by generating the following sequence:

$$\underbrace{\underbrace{\underbrace{\mathbf{false} \vee p}_{\mathcal{F}\ \mathbf{false}} \vee \text{EX}\ p}_{\mathcal{F}^2\ \mathbf{false}} \vee \text{EX}\ (\text{EX}\ p)}_{\mathcal{F}^3\ \mathbf{false}} \vee \text{EX}\ (\text{EX}\ (\text{EX}\ p)) \vee \ldots$$

If this sequence converges to a fixpoint, the result will be equal to the truth set of the property EF $p$.

Based on these properties a CTL model checker can be implemented as shown in Figure 1. The procedure `Evaluate`($f$) computes the truth set of CTL formula $f$. The function `IsEvaluated`($f$) returns **true** if the truth set of $f$ is already computed. For a basic property $f \in BP$, `IsEvaluated`($f$) always returns **true**, i.e., we assume that the truth set of basic properties are represented in the form we want before the procedure `Evaluate` is called. Using the temporal operator equivalences, the procedure `Evaluate` can compute truth set of any CTL formula.

As discussed above, after the truth set of $f$ is computed, we can check if $I \subseteq f$. If this condition holds the model checker reports that the property is proved. If $I \not\subseteq f$, we compute the truth set of $\neg f$ and look for a state $s \in I \cap \neg f$, and generate a counter example [29].

## 3.2 Symbolic Representations

Assume that `Symbolic` is the data type used for encoding sets of states, and `Formula` is the data type used for representing CTL formulas. Then, the signature of the procedure `Evaluate`, shown in Figure 1, is

`Symbolic Evaluate(Formula)`

The procedure `Evaluate` computes a representation of type `Symbolic` for the truth set of its argument formula. Note that, in Figure 1, the operators $\neg$, $\wedge$, $\vee$, and EX are implemented by procedures `Not`, `And`, `Or`, and `EX`. We specify these procedures below:

`Symbolic Not(Symbolic)` : Given an argument that represents a set $p \subseteq S$, it returns a representation for $S - p$.

`Symbolic And(Symbolic,Symbolic)` : Given two arguments representing two sets $p, q \subseteq S$, it returns a representation for $p \cap q$.

`Symbolic Or(Symbolic,Symbolic)` : Given two arguments representing two sets $p, q \subseteq S$, it returns a representation for $p \cup q$.

`Symbolic EX(Symbolic)` : Given an argument that represents a set $p \subseteq S$, it returns a representation for the set $\{s \mid \exists s' \,.\, s' \in p \;\wedge\; (s, s') \in R\}$.

`Boolean IsSatisfiable(Symbolic)` : Given an argument representing a set $p \subseteq S$, returns **true** if $p \equiv \emptyset$, returns **false** otherwise.

Combined with the other procedures, the procedure `IsSatisfiable` is used for testing subsumption and equivalence. This functionality is needed to check for convergence in the fixpoint computations. Using the procedures described above, given a temporal formula, the procedure `Evaluate` computes a symbolic representation for its truth set.

The computation of the procedure `EX` involves computing a relational image. Given a set $p \subseteq S$ and a relation $X \subseteq S \times S$ we use $X\ p$ to denote relational image of $p$ under $X$, i.e., $X\ p$ is defined as restricting the domain of $X$ to set $p$, and returning the range of the result. Note that we can think of relation $X$ as a functional $X \;:\; 2^S \rightarrow 2^S$. Then, $X\ p$ denotes the application of the functional $X$ to set $p$. Let $R^{-1}$ denote the inverse of the transition relation $R$. Then EX $p \equiv R^{-1}\ p$, i.e., functional EX corresponds to the inverse of the transition relation $R$. Hence, we can compute the procedure `EX` using a relational image computation. Most model checkers represent transition relation $R$ in a partitioned form to make the relational image computation more efficient [21].

Any representation which is able encode the set of initial states $I$ and the set of basic properties $BP$, and supports the above functionality can be used as a symbolic representation in a model checker. We call such a representation an *adequate language* for model checking [49]. For example, for finite-state systems, boolean logic would be one such representation. It is possible to implement procedures for negation, conjunction, disjunction and satisfiability of boolean logic formulas. If we can represent the transition relation $R$ as a boolean logic formula, then relational image computation $R^{-1}\ p$ can be computed by conjuncting the formula representing $R^{-1}$ and the formula representing $p$, and then eliminating the variables in the domain of the resulting relation using existential quantifier elimination. BDDs are an efficient data structure for representing boolean logic formulas, and they support all the functionality mentioned above [14]. They have been successfully used for model checking [23, 51]. However, they can not encode infinite variable domains.

Recently, we developed a model checker for systems with unbounded integer variables using Presburger formulas as a symbolic representation [19, 20]. There are effective procedures for manipulating Presburger formulas which support the above functionality – for example Omega Library implements a set of such procedures [48]. We implemented a model checker using Omega Library as our symbolic manipulator. Although our Presburger model checker implements the same procedure given in Figure 1, model checking computations become undecidable for infinite domains, i.e., the fixpoint computations given in Figure 1 for computing EU and AU operators may not always converge for infinite domains. We address this issue using conservative analysis techniques [19, 20] which we discuss below.

Although various symbolic representations have been successfully used in model checking, we think that it is unlikely to find a symbolic representation which is efficient for all variable types. Therefore, to analyze software systems which typically involve various variable types, we developed a composite representation framework which enables us to combine different symbolic representations in one model checker.

# 4 Composite Model Checking

We will demonstrate our composite model checking approach on the requirements specification of the control software of a reactor's cooling system (Figure 2). This example, called the safety injection system, was adapted from previous specifications in [9, 31, 40]. The full SCR specification is given in Section 7. The safety injection system functions as a feedback loop: its sensors monitor the coolant system's water pressure

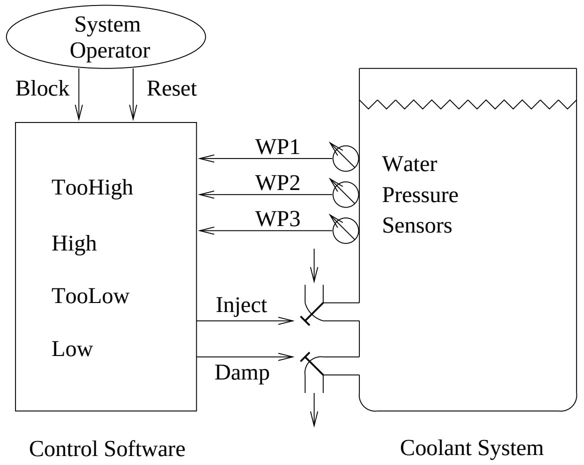


Figure 2: A Nuclear Reactor's Water Coolant System

via three sensors with active redundancy. An action is taken when two of the three sensors agree on a condition. The control software has four modes stored by an enumerated variable `Pressure` : { `TooLow`, `Low`, `High`, `TooHigh` }. When the water pressure is determined to fall below a certain threshold (based on the values of the monitored variables `WP1, WP2, WP3`: integer), the control software goes to `TooLow` mode and conveys signals to pressure-control actuators with the objective of increasing it (by setting the controlled variable `Inject` : { `On, Off` } to `On`). This is called safety injection. We also added another controlled variable `Damp` : { `On, Off` }, to be used for reducing the water pressure. The operator may use a manual control (called the `Block` : { `On, Off` } switch) to override safety injection or damp signals during normal start-up or cool-down phases. Moreover, the operator also has a `Reset` : { `On, Off` } switch, which results in the controller program clearing any state triggered by outstanding block signals.

A possible transition of the safety injection system is enabled by the following condition:

$$(\mathtt{WP1} < \mathtt{low} \land \mathtt{WP2} < \mathtt{low}) \ \lor \ (\mathtt{WP1} < \mathtt{low} \land \mathtt{WP3} < \mathtt{low}) \ \lor \ (\mathtt{WP2} < \mathtt{low} \land \mathtt{WP3} < \mathtt{low})$$

i.e., any two of the water pressure sensors show that the water pressure is below `low`. When this condition becomes true, the enumerated variable `Pressure` should be set to `TooLow`. To model this transition with a BDD based model checker one can either 1) abstract the water pressure sensor values and represent the above condition and similar conditions using one boolean variable for each condition; or 2) represent the integer variables `WP1`, `WP2` and `WP3` using a binary encoding by restricting them to a fixed finite domain. Both of these approaches have certain shortcomings.

For example, if we take the first approach we will lose the information that: If any two of the sensors show that the pressure is less than *low* it is not possible for any two of the sensors to show that pressure is between *low* and *high*, i.e.,

$$\begin{aligned}
&((\mathtt{WP1} < \mathtt{low} \land \mathtt{WP2} < \mathtt{low}) \ \lor \ (\mathtt{WP1} < \mathtt{low} \land \mathtt{WP3} < \mathtt{low}) \ \lor \ (\mathtt{WP2} < \mathtt{low} \land \mathtt{WP3} < \mathtt{low})) \Longrightarrow \\
&\neg((\mathtt{low} \leq \mathtt{WP1} < \mathtt{high} \land \mathtt{low} \leq \mathtt{WP2} < \mathtt{high}) \ \lor \ (\mathtt{low} \leq \mathtt{WP1} < \mathtt{high} \land \mathtt{low} \leq \mathtt{WP3} < \mathtt{high}) \lor \\
&(\mathtt{low} \leq \mathtt{WP3} < \mathtt{high} \land \mathtt{low} \leq \mathtt{WP3} < \mathtt{high}))
\end{aligned}$$

If we abstract away the variables `WP1`, `WP2`, `WP3` as described above, this property may not hold for the abstract transition system the model checker is analyzing. Hence, the model checker may generate spurious behaviors. It is possible to explicitly state that the boolean variables representing the above predicates cannot be true at the same time. But it is not trivial to decide how many such conditions should be stated explicitly in analyzing the system. They can be introduced incrementally to remove spurious behaviors reported by the model checker. Either way more analysis would be required. Of coarse it is possible for the initial abstraction to be strong enough to prove the properties of interest.

In the second approach we have to pick a fixed domain for the integer variables such as $0 \leq \texttt{WP1} \leq 100$. One difficulty is determining this constant bound. Note that if we check the procedure for $0 \leq \texttt{WP1} \leq 100$, and if there is a transition in the system which is enabled for $\texttt{WP1} = 110$, it will never be considered by the model checker even if it is part of a feasible behavior. Another problem is that as we increase the sizes of the variable domains we will require more boolean variables to encode it. Also BDDs are not efficient in encoding arithmetic constraints. Hence, the model checker may run out of memory if the domains of the variables are too large.

Our composite approach allows us to represent integer variables symbolically using arithmetic constraints without any abstraction. However we restrict our arithmetic constraint representation to Presburger formulas, i.e., formulas composed of logical operations $\wedge$, $\vee$, $\neg$; quantifiers $\exists$, $\forall$; arithmetic operations $+$, $-$, multiplication by a constant; and arithmetic predicates $\leq$, $<$, $=$ $\geq$, and $>$. By excluding multiplication operation, we guarantee that our symbolic representation supports the functionality required for model checking. If we include the multiplication operation, relational image computations (i.e., existential quantifier elimination) becomes uncomputable.

If we restrict our symbolic representation only to Presburger formulas, then we have to map all variables (including `Inject`, `Damp`, `Block`, `Reset`, `Pressure`) to integers, or eliminate the boolean and enumerated variables by partitioning the state-space based on their valuations [19, 20]. Because of the high complexity of manipulating Presburger constraints mapping all variables to integers does not scale to large systems. Also, for large numbers of boolean and enumerated variables partitioning generates too many partition classes and makes model checking infeasible. In our composite model we use BDD and Presburger representations together. We encode integer variables `WP1`, `WP2` and `WP3` using Presburger formulas and enumerated and boolean variables using BDDs. Hence, the composite model checker uses efficient representations for each variable type and does not report any spurious behaviors. Using this approach we can also verify systems with unbounded integer variables or with variables which do not have fixed bounds. For example, for the safety injection system, we can either specify that the domains of pressure sensors are infinite $0 \leq \texttt{WP1}, \texttt{WP2}, \texttt{WP3}$, or fixed but not specified $0 \leq \texttt{WP1}, \texttt{WP2}, \texttt{WP3} \leq \texttt{bound}$ where `bound` is an unspecified constant. Similarly the constants such as `low` and `high` can be left unspecified.

## 4.1 Composite Representations

We assume that each variable $v \in V$ has an encoding type $t_v \in T$ where $T$ denotes the set of symbolic representations available in the composite model checker. In our composite model we represent the transition relation $R$ as a disjunction of a set of atomic events:

$$R \equiv \bigvee_{e \in E} R_e,$$

where the transition relation of each atomic event is conjunctively partitioned based on symbolic encoding types as:

$$R_e \equiv \bigwedge_{t \in T} R_e^t.$$

We call $L^t$ the symbolic representation language of type $t \in T$. For example for BDD based model checkers there is only one encoding type (boolean) and the language $L^t$ of that type is boolean logic. Our prototype composite model checker has an additional type (integer) and the language $L^t$ for that type is Presburger arithmetic formulas. For the sake of carrying out model checking, we assume that any type language $L^t$ is an adequate language, i.e., supports the functionality defined in Section 3.2. In our prototype model checker this functionality is implemented by the Omega Library [48] for the integer type, and by our own BDD library for the boolean type.

Consider the following (simplified) event of the safety injection system:

$$\begin{aligned} R_e \;\equiv\; & (\texttt{WP1} < \texttt{low} \wedge \texttt{WP2} < \texttt{low}) \;\vee\; (\texttt{WP1} < \texttt{low} \wedge \texttt{WP3} < \texttt{low}) \;\vee\; (\texttt{WP2} < \texttt{low} \wedge \texttt{WP3} < \texttt{low}) \\ & \wedge \neg(\texttt{Pressure} = \texttt{TooLow}) \;\wedge\; \texttt{Pressure}' = \texttt{TooLow} \end{aligned}$$

where unprimed variables indicate the current state values and the primed variables indicate the next state values. This event states that if two of the water pressure sensors show that the water pressure is below `low` and `Pressure` is not in the mode `TooLow` then in the next state `Pressure` should be in the mode `TooLow`. In our composite model this transition will be represented as:

$$\begin{aligned} R_e \;&\equiv\; R_e^I \;\wedge\; R_e^B \\ R_e^I \;&\equiv\; (\texttt{WP1} < \texttt{low} \wedge \texttt{WP2} < \texttt{low}) \;\vee\; (\texttt{WP1} < \texttt{low} \wedge \texttt{WP3} < \texttt{low}) \;\vee\; (\texttt{WP2} < \texttt{low} \wedge \texttt{WP3} < \texttt{low}) \\ R_e^B \;&\equiv\; \neg(\texttt{Pressure} = \texttt{TooLow}) \;\wedge\; \texttt{Pressure}' = \texttt{TooLow} \end{aligned}$$

where $I$ denotes the integer encoding type (i.e., Presburger formulas) and $B$ denotes the boolean encoding type (i.e., BDDs). Note that $R_e^I$ can be represented as a Presburger arithmetic formula and $R_e^B$ can be represented as a boolean logic formula (stored as a BDD) using a binary encoding of the variable `Pressure`. In Section 7 we show how the whole SCR specification for the safety injection system can be represented using a set of events expressed as conjunctions of boolean logic and Presburger arithmetic formulas.

More generally, given a set of variable types $t \in T$ and their associated adequate languages $L^t$, we define a *composite language* $L^c$ to be generated by the following grammar:

$$\mathrm{CF} \;::=\; \mathrm{CF} \;\vee\; \mathrm{CF} \mid \mathrm{CF} \;\wedge\; \mathrm{CF} \mid \neg\mathrm{CF} \mid \mathrm{form}^t$$

where $t \in T$ and terminal $\mathrm{form}^t$ denotes a formula in language $L^t$, and nonterminal CF denotes a *composite formula* in $L^c$. We use this composite language to represent the basic properties $BP$, and we also assume that the set of initial states $I$ can be represented as a composite formula, i.e., $I \in L^c$.

For example, one of the properties that the safety injection system should satisfy is:

> Whenever `Pressure` = `TooLow` then two of the water pressure sensors show that the pressure is below `low`, and whenever two of the water pressure sensors show that the pressure is below `low` then `Pressure` = `TooLow`.

We can represent this property as an invariant as follows:

$$\mathrm{AG}(\texttt{Pressure} = \texttt{TooLow} \longleftrightarrow (\texttt{WP1} < \texttt{low} \wedge \texttt{WP2} < \texttt{low} \;\vee\; \texttt{WP1} < \texttt{low} \wedge \texttt{WP3} < \texttt{low} \;\vee\; \texttt{WP2} < \texttt{low} \wedge \texttt{WP3} < \texttt{low})$$

Note that the assertion inside the temporal operator AG is a basic property represented as a composite formula which involves both boolean and integer variables.

As with single-type symbolic model checking, we use the formulas in $L^c$ to symbolically encode state sets. We convert any composite state formula $q \in L^c, q \subseteq S$ to a disjunctive form, and represent it as follows:

$$q \;\equiv\; \bigvee_{i=1}^{n_q} \bigwedge_{t \in T} q_i^t$$

where each $q_i^t \in L^t$, and $n_q$ denotes the number of disjuncts needed. Such a disjunctive form can be obtained for any composite term, since we do not allow functions (or predicates) with arguments which have different types. For example the property

$$q \;\equiv\; \texttt{Pressure} = \texttt{TooLow} \longleftrightarrow (\texttt{WP1} < \texttt{low} \wedge \texttt{WP2} < \texttt{low} \;\vee\; \texttt{WP1} < \texttt{low} \wedge \texttt{WP3} < \texttt{low} \;\vee\; \texttt{WP2} < \texttt{low} \wedge \texttt{WP3} < \texttt{low})$$

can be represented in this disjunctive form as follows:

$$q \equiv \bigvee_{i=1}^{2} \bigwedge_{t\in\{B,I\}} q_i^t \equiv q_1^B \wedge q_1^I \vee q_2^B \wedge q_2^I$$

where

$$\begin{array}{lcl} q_1^B & \equiv & \texttt{Pressure} = \texttt{TooLow} \\ q_1^I & \equiv & \texttt{WP1} < \texttt{low} \wedge \texttt{WP2} < \texttt{low} \vee \texttt{WP1} < \texttt{low} \wedge \texttt{WP3} < \texttt{low} \vee \texttt{WP2} < \texttt{low} \wedge \texttt{WP3} < \texttt{low} \\ q_2^B & \equiv & \texttt{Pressure} \neq \texttt{TooLow} \\ q_2^I & \equiv & \texttt{WP1} \geq \texttt{low} \wedge \texttt{WP2} \geq \texttt{low} \vee \texttt{WP1} \geq \texttt{low} \wedge \texttt{WP3} \geq \texttt{low} \vee \texttt{WP2} \geq \texttt{low} \wedge \texttt{WP3} \geq \texttt{low} \end{array}$$

## 4.2 Logical Operations on Composite Representations

Assume that we have two state sets $p$ and $q$ represented symbolically as

$$p \equiv \bigvee_{i=1}^{n_p} \bigwedge_{t\in T} p_i^t \quad \text{and} \quad q \equiv \bigvee_{i=1}^{n_q} \bigwedge_{t\in T} q_i^t$$

where each $p_i^t, q_i^t \in L^t$. Now we show how to compute logical operators disjunction, conjunction and negation on $p$ and $q$:

$$p \vee q \equiv (\bigvee_{i=1}^{n_p} \bigwedge_{t\in T} p_i^t) \vee (\bigvee_{i=1}^{n_q} \bigwedge_{t\in T} q_i^t) \qquad p \wedge q \equiv \bigvee_{i=1,j=1}^{n_p,n_q} \bigwedge_{t\in T} p_i^t \wedge q_j^t$$

$$\neg q \equiv \bigvee \bigwedge_{j=1}^{n_q} \neg q_j \quad \text{where} \quad q_j = q_j^t \quad t \in T$$

In other words, disjunction just requires appending the two disjuncts together; the result will be in our composite symbolic form. Conjunction is computationally more expensive. Using the distributive properties of boolean algebra, we can compute all the pertinent disjuncts – yet we may end up with $n_p \times n_q$ disjuncts, which we have to compute by traversing the disjunctive representations of $p$ and $q$.

Finally, taking a complement is more expensive; indeed, complementation of a set $q$ with $n_q$ disjuncts may, in fact, create $|T|^{n_q}$ disjuncts in the worst case – however it is very likely that most of these will be empty. Hence, we build $\neg q$ in an incremental manner so that we try to minimize the number of disjuncts generated. We do this by testing for emptiness on the fly, while we are computing the conjunctions.

During model checking operations, the number of disjuncts in a composite formula can easily increase. As we showed above, applying the disjunction operation is relatively cheap – yet it can still linearly increase a formula's complexity. And this problem gets worse when applying conjunction (with quadratic growth) and even more so with negation (and its worst-case exponential growth). In practice, however, each symbolic library will contain its own simplification procedures, which can minimize the number of formulas used to represent a set of valuations. (This is true for both the BDD and the Presburger libraries we used in out prototype.) So, for composite models, the challenge lies in merging as many terms as possible into a single-type format, and still retaining the semantics of the original formula. To do this we use some simple reduction rules. Given a composite formula with two disjuncts $q \equiv (\bigwedge_{t\in T} q_1^t) \vee (\bigwedge_{t\in T} q_2^t)$ we have the following properties:

$$\forall s \in T \quad s \neq t \quad q_1^s \equiv q_2^s \quad \rightarrow \quad q \equiv (\bigwedge_{s\in T, s\neq t} q_1^s) \wedge (q_1^t \vee q_2^t)$$

$$\forall t \in T \quad q_1^t \subseteq q_2^t \quad \rightarrow \quad q \equiv \bigwedge_{t\in T} q_2^t$$

Note that in both cases we can reduce the formula from two disjuncts to one. Hence, to simplify a general composite formula we (1) check all pairs for the conditions listed above, and (2) merge the appropriate disjuncts when a condition is satisfied.

### 4.3 Composite Image Computations

The fundamental property which enables us to manipulate each symbolic encoding type individually is the fact that relational image computations distribute over the symbolic encoding types:

$$R_e \ \ (\bigwedge_{t\in T} q_i^t) \ \equiv \ (\bigwedge_{t\in T} R_e^t) \ \ (\bigwedge_{t\in T} q_i^t) \ \equiv \ \bigwedge_{t\in T}(R_e^t \ \ (\bigwedge_{t\in T} q_i^t)) \ \equiv \ \bigwedge_{t\in T}(R_e^t \ \ q_i^t)$$

The first step follows from the fact that each $R_e^t$ only references variables of type $t$, i.e., we can push the existential quantification for the relational image computation inside the first conjunction. The second step follows from the fact that each $q_i^t$ only references variables of type $t$, i.e., we can push the existential quantification for the relational image computation inside the second conjunction. Taken as a whole, the property shows that the relational image computation for different symbolic encoding types are orthogonal, and hence they can be computed separately.

A composite model checker for CTL can be constructed using exactly the same procedure given in Figure 1, except it uses composite representations as its symbolic representation. The functional EX is implemented with respect to the system's event decomposition, and the composite representation of $q \equiv \bigvee_{i=1}^{n_q} \bigwedge_{t\in T} q_i^t$:

$$\text{EX } q \ \equiv \ \bigvee_{e\in E} R_e^{-1} \ q \ \equiv \ \bigvee_{e\in E} \bigvee_{i=1}^{n_q} \bigwedge_{t\in T}((R_e^t)^{-1} \ \ q_i^t)$$

which follows from the results above.

In the following sections we demonstrate the effectiveness of our composite model checker on two example systems, a requirements specification for a nuclear reactor's water pressure system, and a transport protocol. It was not possible to analyze these systems using a BDD or Presburger based model checker. However using our composite approach we were able to analyze them successfully.

If all variable domains are finite, the composite CTL model checker will always converge. However, if integers (or other unbounded types) are involved, the procedure in Figure 1 may be a partial function, as it is with our prototype tool – which uses BDDs and Presburger arithmetic over unbounded transition systems. The problem of checking if a CTL property holds for an infinite transition system is an undecidable problem. However using conservative analysis techniques it is possible to analyze a wide range of systems, below we discuss this approach.

## 5 Conservative Analysis

Model checking is undecidable for infinite-state systems. This implies that the fixpoint computations in our composite model checker are not guaranteed to terminate if we use symbolic representations such as Presburger arithmetic. However, we do not always need to compute the fixpoints exactly to prove or disprove temporal properties. We can use conservative analysis techniques, i.e., given a temporal property, we either 1) prove the property, or 2) disprove the property and generate a counter-example, or 3) report that the analysis is inconclusive. There are different ways to implement conservative analysis:

- Generating lower or upper bounds to least fixpoints using approximation techniques such as widening.
- Computing fixpoints on abstract transition systems which are generated by abstracting away selected variables or by restricting their domains.

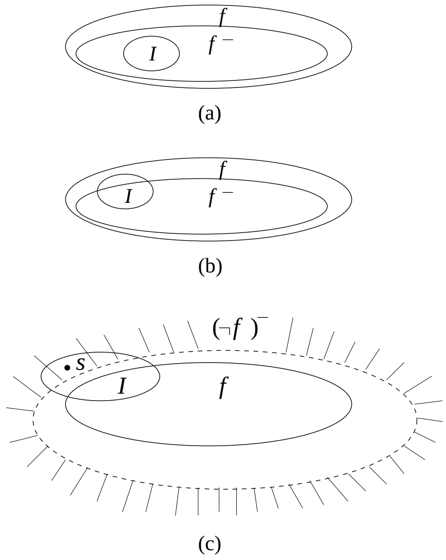


Figure 3: Lower Bounds for the Truth Set of a Temporal Property and Its Negation

- Using compositional approaches which analyze selected concurrent components before composing them.

In this paper we use the first approach, i.e., we present techniques for approximating fixpoint computations using composite representations. However it is possible to integrate all these approaches to our conservative analysis framework. This would make it possible to analyze a system incrementally instead of attempting to verify the whole system in one shot. We think that this approach will be key to analyzing large systems. Since our approach is conservative, the proofs and the counter-examples are always exact, i.e., there are no spurious results. Below we describe our conservative analysis framework which was initially presented in [19].

If we cannot directly compute the truth set of a property $f$ for a transition system $TS = (S, I, R)$, the next best thing is to generate a *lower bound* for $f$, denoted $f^-$, such that $f^- \subseteq f$. Then if we determine that the set of initial states is included in this lower bound (i.e., $I \subseteq f^-$), we have also achieved our objective – that $I \subseteq f$, i.e., we proved that $TS \models f$ (Figure 3 (a)). However, if $I \not\subseteq f^-$, we cannot conclude anything because it can be a *false negative* (Figure 3 (b)). In that case we can compute a lower bound for the negated property: $(\neg f)^-$. If we can find a state $s$ such that $s \in I \cap (\neg f)^-$, then we can generate a counter example which would be a *true negative* (Figure 3 (c)). If both cases fail, i.e., both $I \not\subseteq f^-$ and $I \cap (\neg f)^- \equiv \emptyset$, then the analyzer cannot report a definite answer.

Since model checking procedures are recursive (as in Figure 1), we have to compute an approximation to a formula by first computing approximations for its subformulas. Hence, to compute a lower bound to a property like $g = \neg h$, we first need to compute an *upper* approximation $h^+$ for the subformula $h$ where $h^+ \supseteq h$, and then let $g^- \equiv S - h^+$. Similarly we can compute an upper bound for $g$ using a lower bound for $h$. This follows directly from set theory, since $S - f^+ \subseteq \neg f$ and $S - f^- \supseteq \neg f$ where $f^- \subseteq f \subseteq f^+$. Thus, we need algorithms to compute both lower and upper bounds of temporal formulas.

When analyzing a negation-free formula, the compositionality of an approximation follows directly from the fact that all operators other than "$\neg$" are monotonic. This means that any lower/upper approximation for a negation free formula can be computed using the corresponding lower/upper approximation for its subformulas. As for handling arbitrary levels of negation, we can easily generalize the above mentioned method for outermost negation operators [19].

As we discussed earlier the CTL model checking procedure (Figure 1) recursively computes least fixpoints to generate truth sets of temporal formulas. Note that each iteration of the exact fixpoint computations will yield a lower a bound for $f_1 \mathrm{EU} f_2$ and $f_1 \mathrm{AU} f_2$. So, to obtain a lower approximation for the purposes of analysis, we need only stop after a finite number of iterations; in this manner we are guaranteed to have a conservative approximation. In most cases, deciding when to stop is usually a matter of computing resources, time constraints, and human patience. However, if a result obtained is not precise enough to prove the property of interest, it can be cached away and improved later, by running more fixpoint iterations.

As for upper bounds, we use a technique similar to widening [32], but over multiple types. Assume that $q_1^t$ and $q_2^t$ are two symbolic sets formed over a single type $t \in T$, and that "$\bigtriangledown^t$" is a type-specific widening operator such that:

$$q_1^t \cup q_2^t \subseteq q_1^t \bigtriangledown^t q_2^t$$

i.e., $q_1^t \bigtriangledown^t q_2^t$ is an upper bound for the union computation. (Note that we do not have the guaranteed convergence requirement given in [32], i.e., our approximate fixpoint computations are not guaranteed to converge.)

Obviously there are many choices for operators which majorize binary union. However, if $\bigtriangledown^t$ is to be useful, it should – in many settings – be able to "guess" the direction of growth in the fixpoint iterates, and to extend the successive iterates in these directions. This will accelerate the fixpoint computation by generating a majorizing sequence to the exact fixpoint iterates. We show in [19] how this is done for integer domains, over multiple convex regions.

We extend the widening idea to composite models as follows. Assume that we have two composite representations

$$p \equiv \bigvee_{i=1}^{n_p} p_i \equiv \bigvee_{i=1}^{n_p} \bigwedge_{t \in T} p_i^t \quad \text{and} \quad q \equiv \bigvee_{i=1}^{n_q} q_i \equiv \bigvee_{i=1}^{n_q} \bigwedge_{t \in T} q_i^t$$

such that $p \subseteq q$. Let $d_p$ denote the following subset of disjuncts forming $p$: For each $p_i \in d_p$, there exists some $q_j$ such that $p_i \subseteq q_j$. Likewise, let $d_q$ denote the set of disjuncts in $q$ for which $q_j \in d_q$ means there is a $p_i$ such that $p_i \subseteq q_j$. Then we define $p \bigtriangledown q$ as

$$p \bigtriangledown q \equiv \bigvee_{p_i \notin d_p} p_i \vee \bigvee_{q_i \notin d_q} q_i \vee \bigvee_{p_i \subseteq q_j} \bigwedge_{t \in T} (p_i^t \bigtriangledown^t q_j^t)$$

Note that the composite widening operator $\bigtriangledown$ uses type specific widening operators $\bigtriangledown^t$ to implement the widening operation on composite representations. Using this composite widening operator we can generate a majorizing sequence for the least fixpoint computations used in Figure 1 as follows:

$$\begin{array}{lcl|lcl}
\hat{q}_0 & \equiv & \textbf{false} & \hat{q}_0 & \equiv & \textbf{false} \\
\hat{q}_{i+1} & \equiv & \hat{q}_i \bigtriangledown (f_2 \vee (f_1 \wedge \mathrm{EX}\ \hat{q}_i)) & \hat{q}_{i+1} & \equiv & \hat{q}_i \bigtriangledown (f_2 \vee (f_1 \wedge \mathrm{EX}\ \hat{q}_i \wedge \mathrm{AX}\ \hat{q}_i)) \\
(f_1 \mathrm{EU} f_2)^+ & \equiv & \hat{q}_n \ \text{ when } \ \hat{q}_n \equiv \hat{q}_{n+1} & (f_1 \mathrm{AU} f_2)^+ & \equiv & \hat{q}_n \ \text{ when } \ \hat{q}_n \equiv \hat{q}_{n+1}
\end{array}$$

As mentioned above in our prototype implementation we have two symbolic representation types: boolean (encoded as BDDs), and integer (encoded as Presburger arithmetic formulas). In our prototype composite model checker we implemented the widening operator for the integer type using our multi-polyhedra widening operator [19]. For the variables encoded as BDDs (i.e., the boolean type), we simply used union as the widening operator.

## 6 Composite Model Checking Toolset

We implemented a prototype toolset that implements the methods described above. The software architecture of the proposed tool is shown in Figure 4. Its layered structure is designed to facilitate future extensions.

The front end compiler translates the software specifications to the composite symbolic representations. We use a simple event based language to specify the input systems, and CTL to specify temporal properties

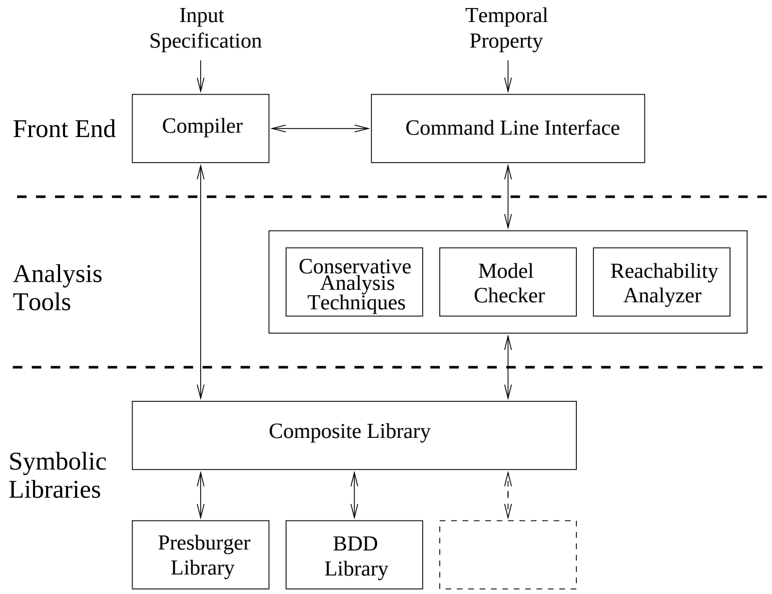


Figure 4: Software Architecture of the Prototype Tool

of the input systems.

In the lower level, we have forward, backward, exact, and approximate fixpoint computations for computing truth sets of temporal properties. The forward fixpoint computations can be used for approximate reachability analysis which can increase the convergence rate of the fixpoint computations [20]. We also support fixpoint computations over partitioned state spaces since partitioning could be crucial in reducing the size of symbolic representations [21, 22]. Using our conservative analysis framework we integrate approximate fixpoint computations to our composite model checker.

The lowest level of our layered architecture consists of libraries implementing symbolic representations such as BDDs and Presburger formulas. These libraries share the same API. At the next level we have our composite-model library, which handles operations over composite representations (e.g., equivalence check, intersection, etc.). The composite library calls the procedures in the lower level to manipulate the composite representations as described in the previous sections.

Our prototype implementation uses our own BDD library and the Omega Library (used for manipulating Presburger formulas) [17, 18] at the lowest level. Our composite approach will allow us to expand to additional symbolic representations. Note that given an adequate symbolic representation, we can easily integrate its implementation to our software architecture (the dashed box in Figure 4). We are interested in expanding our composite model checker by adding other symbolic representations such as automata based representations [11, 12, 8, 10, 37, 41, 50], real arithmetic [2, 1], etc. This will allow us to encode variable types like reals and queues, and to investigate the scalability of our approach to several data types.

# 7 The Safety Injection System

In this section we check the requirements specification for the control software of a nuclear reactor's cooling system (Figure 2). It was adapted from previous specifications in [9, 31, 40] – and in fact, we take a superset of these requirements, as well as add a few of our own. In [31] and [9] this specification is rendered using the SCR tabular notation – although the system given in [9] uses only one pressure sensor. Here, we adapt the original three-sensor system to the style of requirements in [9], assuming that an action is taken when two of three sensors agree on a condition.

Additionally, we have complicated the system somewhat, by adding an actuator called "damp," which is

| | |
|---|---|
| **Constants:** | `min`, `low`, `high`, `toohigh`, `max` : integer |
| **Monitored Variables:** | `WP1, WP2, WP3`: integer; `Block, Reset` : { `On, Off` } |
| **Controlled variables:** | `Inject, Damp` : { `On, Off` } |
| **Terms:** | `Overridden` : boolean |
| **Mode Class:** | `Pressure` : { `TooLow`, `Low`, `High`, `TooHigh` }; |
| **Initial Conditions:** | `Block, Reset, Inject, Damp` = `Off`; |
| | `Overridden` = `False`; `Pressure` = `Low`; |
| | `low` < `WP1`, `WP2`, `WP3` < `high`; |
| | `min` < `low` < `high` < `toohigh` < `max`; |

| | | |
|---|---|---|
| CTLow | $\stackrel{\text{def}}{=}$ | (`WP1`, `WP2` < `low`) OR (`WP1`, `WP3` < `low`) OR (`WP2`, `WP3` < `low`) |
| CLow | $\stackrel{\text{def}}{=}$ | (`low` ≤ `WP1`, `WP2` < `high`) OR (`low` ≤ `WP1`, `WP3` < `high`) OR (`low` ≤ `WP2`, `WP3` < `high`) |
| CHigh | $\stackrel{\text{def}}{=}$ | (`high` ≤ `WP1`, `WP2` < `toohigh`) OR (`high` ≤ `WP1`, `WP3` < `toohigh`) OR (`high` ≤ `WP2`, `WP3` < `toohigh`) |
| CTHigh | $\stackrel{\text{def}}{=}$ | (`toohigh` ≤ `WP1`, `WP2`) OR (`toohigh` ≤ `WP1`, `WP3`) OR (`toohigh` ≤ `WP2`, `WP3`) |

Figure 5: SCR Specification of the Safety Injection System (Declarations)

similar to “safety injection” – but conveys the opposite meaning. When a “damp” signal is sent, it means that incoming water pressure is getting too high, and that the coolant system should start reducing it. Again, the “block” signal can disable this condition (which the reset button can also clear).

The SCR requirements notation is used to specify plant-controller systems; it gives the choice of specifying a state-change explicitly (via events), or implicitly (via current valuations of of other conditions). In SCR, a system’s environment is abstracted as a set of monitored and controlled state variables, corresponding to the sensors and actuators in a control-theoretic setting. Based on the values conveyed on the monitored variables, the system can change its internal state – and send out signals on the controlled variables.

SCR uses three basic constructs to represent software behavior: modes, terms and conditions. A *mode class* is just an enumerated type, denoting an internal state of the controller. Each class has an exclusivity relation between its values – but many classes can be active simultaneously. Modes are usually described behaviorally, in a table denoting current-state/next-state transitions, labeled by conditions that trigger the state-change. A *term* is any function of modes, variables, constants, operators, etc. – i.e., a function built up over the other symbols in an SCR specification. A *condition* is just a term which evaluates to True or False, based on the present state’s valuation.

The SCR specification of our safety injection system is given in Figure 5 and Figure 6. The monitored variables `WP1, WP2, WP3` model the readings from three water pressure sensors. The mode class `Pressure` denotes the controller’s state, dependent on the other conditions of the system. Note that it ranges over `TooLow, Low, High, TooHigh` – since we also include an “unsafe” mode for overly-high pressure.

Sensor values change between two constants `min` and `max`. Other constants – `low`, `high`, `toohigh` – indicate the critical pressure levels to which the system reacts. Water pressure is assumed to be dangerously low when it is between `min` and `low`, and too high when it ranges between `toohigh` and `max`. The objective is to maintain the pressure between `low` and `high`.

If at least two of three sensors detect a drop in the water pressure below the constant `low`, this causes the system to enter the mode `TooLow` – and to start safety injection (if it is not overridden). The analogous logic holds for when pressure gets too high.

We show the explicit transition tables for mode `Pressure` and term `Overridden`. As for the controlled variables `Inject` and `Damp`, their state-changes are stated implicitly, via condition tables. In these tables, `@T(InMode)` denotes that the system enters the corresponding mode (i.e., the mode shown on the left-hand-side of the corresponding column).

Note that the system is considered `Overridden` when it is blocked in particular modes (as well as not

| Old Mode | Event | New Mode |
|---|---|---|
| – | @T(CTLow) | TooLow |
| – | @T(CLow) | Low |
| – | @T(CHigh) | High |
| – | @T(CTHigh) | TooHigh |

| Mode | Events | |
|---|---|---|
| High | False | @T(InMode) |
| TooLow, Low, TooHigh | @T(Block=On) WHEN Reset=Off | @T(InMode) OR @T(Reset=On) |
| Overridden | True | False |

| Mode | Conditions | |
|---|---|---|
| Low, High, TooHigh | True | False |
| TooLow | Overridden | NOT Overridden |
| Inject | Off | On |

| Mode | Conditions | |
|---|---|---|
| TooLow, Low, High | True | False |
| TooHigh | Overridden | NOT Overridden |
| Damp | Off | On |

Figure 6: SCR Specification of the Safety Injection System (Tables)

reset). As for `Inject`, the condition table states that it can only be actuated when the system is in mode `TooLow` and it is not overridden. The conditions for `Damp` are analogous.

The semantics of SCR is defined in [40]. An important restriction of the model is the One Input Assumption, which states that only one monitored variable can change at a time. For the SCR specification given in Figure 5 and Figure 6 we assume that at any given time, either `Block` or `Reset` can toggle, or the water pressure readings change. However, we assume the pressure sensors are read in on a vector – hence, we treat them as one input.

We also place environmental constraints on the fluctuations of the pressure readings, as in [9]. Here, we ensure that readings can change within a certain range, ± `bound`.

Note that in the specification given in Figure 5 and Figure 6 values of the constants `min`, `max`, `low`, `high`, `toohigh`, and `bound` are unspecified. These constants can take any integer value as long as they satisfy the ordering `min` < `low` < `high` < `toohigh` < `max`. Our representation of the system, which is described below, leaves these constants as unspecified. Hence, any property we verify is valid for any possible interpretation of these constants.

## 7.1 Event-Action Language Specification of the Safety Injection System

Figures 7, 8, and 9 show the representation of the safety injection system in our event-action language. Using the formal semantics of SCR requirements specifications given in [40], any specification in SCR notation can be automatically converted to our event-action language – and any theorems we prove valid in our model will be true for the original SCR requirements.

We use boolean variables to encode unordered enumerated SCR variables. Note that we could actually encode `Pressure` using two boolean variables, but for clarity of presentation we use four. We also define several formulas as abbreviations of complicated expressions – for example, to change conditions, to evaluate voting of sensors, etc. Note that formulas $FInject$ and $FDamp$ define the semantics of condition tables for `Inject` and `Damp`, respectively; similarly $FOver$ defines the semantics of the event table for `Overridden`.

PROGRAM Safety Injection System
CONSTANTS
$min, max, low, high, toohigh, bound$: integer $min < low < high < toohigh < max$
VARIABLES
$wp1, wp2, wp3$: integer
$Block, Reset, Inject, Damp, Over, TLow, Low, High, THigh$: boolean
INITIAL CONDITION
$low \leq wp1, wp2, wp3 < high \land \neg Block \land \neg Reset \land \neg Inject \land \neg Damp \land \neg Over$
EVENTS

$$
\begin{array}{rl}
e_{TLow}: & RTLow \land TLow' \land \neg Low' \land \neg High' \land \neg THigh' \land Feasible \land \\
& FOver \land FInject \land FInject' \land FDamp \land FDamp' \\
e_{Low}: & RLow \land Low' \land \neg TLow' \land \neg High' \land \neg THigh' \land Feasible \land \\
& FOver \land FInject \land FInject' \land FDamp \land FDamp' \\
e_{High}: & RHigh \land High' \land \neg TLow' \land \neg Low' \land \neg THigh' \land Feasible \land \\
& FOver \land FInject \land FInject' \land FDamp \land FDamp' \\
e_{THigh}: & RTHigh \land THigh' \land \neg TLow' \land \neg Low' \land \neg High' \land Feasible \land \\
& FOver \land FInject \land FInject' \land FDamp \land FDamp' \\
e_{Same}: & ((CTLow' \land CTLow) \lor (CLow' \land CLow) \lor (CHigh' \land CHigh) \lor \\
& (CTHigh' \land CTHigh)) \land Feasible \land FOver \land FInject \land \\
& FInject' \land FDamp \land FDamp' \\
e_{BOrR}: & ((Block' = \neg Block \land Reset' = Reset) \lor \\
& (Reset' = \neg Reset \land Block' = Block)) \land \\
& FOver \land FInject \land FInject' \land FDamp \land FDamp'
\end{array}
$$

Figure 7: Event-Action Language Representation of the Safety Injection System Requirements Specifications

$$
\begin{array}{l}
CTLow \stackrel{\text{def}}{=} wp1, wp2 < low \lor wp1, wp3 < low \lor wp2, wp3 < low \\
CLow \stackrel{\text{def}}{=} low \leq wp1, wp2 < high \lor low \leq wp1, wp3 < high \lor low \leq wp2, wp3 < high \\
CHigh \stackrel{\text{def}}{=} high \leq wp1, wp2 < toohigh \lor high \leq wp1, wp3 < toohigh \lor \\
\quad high \leq wp2, wp3 < toohigh \\
CTHigh \stackrel{\text{def}}{=} toohigh \leq wp1, wp2 \lor toohigh \leq wp1, wp3 \lor toohigh \leq wp2, wp3 \\
CTLow' \stackrel{\text{def}}{=} wp1', wp2' < low \lor wp1', wp3' < low \lor wp2', wp3' < low \\
CLow' \stackrel{\text{def}}{=} low \leq wp1', wp2' < high \lor low \leq wp1', wp3' < high \lor \\
\quad low \leq wp2', wp3' < high \\
CHigh' \stackrel{\text{def}}{=} high \leq wp1', wp2' < toohigh \lor high \leq wp1', wp3' < toohigh \lor \\
\quad high \leq wp2', wp3' < toohigh \\
CTHigh' \stackrel{\text{def}}{=} toohigh \leq wp1', wp2' \lor toohigh \leq wp1', wp3' \lor toohigh \leq wp2', wp3' \\
RTLow \stackrel{\text{def}}{=} CTLow' \land \neg CTLow \qquad RLow \stackrel{\text{def}}{=} CLow' \land \neg CLow \\
RHigh \stackrel{\text{def}}{=} CHigh' \land \neg CHigh \qquad RTHigh \stackrel{\text{def}}{=} CTHigh' \land \neg CTHigh \\
Feasible \stackrel{\text{def}}{=} (wp1 - bound \leq wp1' \leq wp1 + bound) \land \\
\quad (wp2 - bound \leq wp2' \leq wp2 + bound) \land \\
\quad (wp3 - bound \leq wp3' \leq wp3 + bound) \land (min \leq wp1', wp2', wp3' \leq max)
\end{array}
$$

Figure 8: Abbreviations for Conditions on Water Pressure Sensor Readings of the Safety Injection System

$$
\begin{array}{lcl}
\mathit{FOver} & \stackrel{\mathrm{def}}{=} & (\mathit{Over}' \wedge \mathit{Block}' \wedge \mathit{Block} \wedge \neg \mathit{Reset} \wedge (\mathit{TLow} \vee \mathit{Low} \vee \mathit{THigh})) \\
 & & \vee (\neg \mathit{Over}' \wedge (\mathit{Reset}' \wedge \mathit{Reset} \vee \mathit{TLow}' \wedge \mathit{TLow} \vee \\
 & & \mathit{Low}' \wedge \neg \mathit{Low} \vee \mathit{High}' \wedge \neg \mathit{High} \vee \mathit{TooHigh}' \wedge \mathit{TooHigh})) \vee \\
 & & (\mathit{Over}' = \mathit{Over} \wedge (\neg(\mathit{Block}' \wedge \mathit{Block} \wedge \neg \mathit{Reset} \wedge \\
 & & (\mathit{TLow} \vee \mathit{Low} \vee \mathit{High} \vee \mathit{THigh})) \vee \\
 & & \neg(\mathit{Reset}' \wedge \mathit{Reset} \vee \mathit{TLow}' \wedge \mathit{TLow} \vee \mathit{Low}' \wedge \neg \mathit{Low} \vee \\
 & & \mathit{High}' \wedge \neg \mathit{High} \vee \mathit{THigh}' \wedge \mathit{THigh}))) \\
\mathit{FInject} & \stackrel{\mathrm{def}}{=} & (\neg \mathit{Inject} \wedge (\mathit{TLow} \wedge \mathit{Over} \vee \mathit{Low} \vee \mathit{High} \vee \mathit{THigh})) \vee \\
 & & (\mathit{Inject} \wedge \mathit{TLow} \wedge \neg \mathit{Over}) \\
\mathit{FDamp} & \stackrel{\mathrm{def}}{=} & (\neg \mathit{Damp} \wedge (\mathit{TLow} \vee \mathit{Low} \vee \mathit{High} \vee \mathit{THigh} \wedge \neg \mathit{Over})) \vee \\
 & & (\mathit{Damp} \wedge \mathit{THigh} \wedge \neg \mathit{Over})
\end{array}
$$

Figure 9: Abbreviations for Conditions on Boolean Variables of the Safety Injection System

As in the SCR requirements we use the One Input Assumption, which yields 6 events specifying the behavior of the system. At any time only one of the following can occur: *Block* or *Reset* may toggle; or values of $wp1, wp2, wp3$ may change within a range $\pm \mathit{bound}$. (Note however that this range is not specified in advance.) Again, we assume that all three pressure readings are updated at the same time.

In event $e_{TLow}$, variables $wp1, wp2, wp3$ may change values, and this may cause a change in system state. Specifically, if the previous voting outcome did not detect a pressure reading of `TooLow` – and now it does – then the event can fire, causing changes in other variables too. This corresponds to the action of SCR event `@T(CTLow)`; as in the original specification, the value of the mode class `Pressure` changes from `Low` to `TooLow`. Events $e_{Low} - e_{THigh}$ behave similarly, whereas $e_{Same}$ represents the transitions where water pressure conditions remain static. Finally, event $e_{BOrR}$ defines the changes in the system entities when one of the variables *Block* or *Reset* changes.

## 7.2 Automated Analysis of the Safety Injection System

We analyzed the safety injection system using our composite model checking technique. We only used exact fixpoint computations in this analysis, and we did not partition the state space (i.e., each fixpoint iterate was stored as a single composite representation). We disjunctively decomposed the transition relation using the event syntax.

Two properties of the safety injection system verified in [9] are:

$$
\begin{array}{ll}
\text{(SIS1)} & \mathrm{AG}((\mathit{Reset} \wedge \neg \mathit{High}) \rightarrow \neg \mathit{Over}) \\
\text{(SIS2)} & \mathrm{AG}((\mathit{Reset} \wedge \mathit{TLow}) \rightarrow \mathit{Inject})
\end{array}
$$

We verified these properties on the system model presented in Figures 7, 8, and 9. (SIS1) and (SIS2) required 2.71 seconds and 2.58 seconds, respectively, as run on a Sun Ultra. Note that, our safety injection system is significantly more complicated than that in [9]; in fact, it models the three-way voting scheme as originally specified in [31]. This complicates the transition system considerably, since the actions are taken by a majority vote on three different readings – which can range over the entire space of integers. Indeed, note that the system we check (Figure 7) is unbounded in most dimensions, since the limit constants $\{\mathit{min}, \mathit{low}, \mathit{high}, \mathit{toohigh}, \mathit{max}\}$ remain unspecified. Hence, any property we check is proved for any concrete values, provided they satisfy the ordering $\mathit{min} < \mathit{low} < \mathit{high} < \mathit{toohigh} < \mathit{max}$. One cannot check such a system with a finite-state model checker like SMV or SPIN, without using some abstraction techniques.

We also wanted to determine whether mode class `Pressure` is a correct abstraction of the water pressure

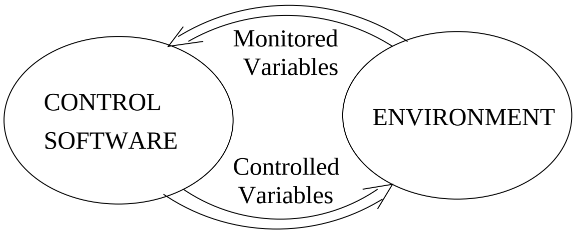


Figure 10: Feedback Between the Control Software and the Environment

readings. This can be shown by checking the following four properties:

(SIS3) $\mathrm{AG}(TLow \longleftrightarrow wp1, wp2 < low \;\;\vee\;\; wp1, wp3 < low \;\;\vee\;\; wp2, wp3 < low)$
(SIS4) $\mathrm{AG}(Low \longleftrightarrow low \leq wp1, wp2 < high \;\;\vee\;\; low \leq wp1, wp3 < high \;\;\vee\;\; low \leq wp2, wp3 < high)$
(SIS5) $\mathrm{AG}(High \longleftrightarrow high \leq wp1, wp2 < toohigh \;\;\vee\;\; high \leq wp1, wp3 < toohigh$
$\vee\; high \leq wp2, wp3 < toohigh)$
(SIS6) $\mathrm{AG}(THigh \longleftrightarrow toohigh \leq wp1, wp2 \;\;\vee\;\; toohigh \leq wp1, wp3 \;\;\vee\;\; toohigh \leq wp2, wp3)$

The model checker verified these (SIS3)-(SIS6) in 16.22, 35.54, 35.39 and 15.94 seconds, respectively. Some other properties we tried were

(SIS7) $\mathrm{AG}(Inject \rightarrow TLow)$
(SIS8) $\mathrm{AG}(Damp \rightarrow THigh)$

which were successfully verified in 1.52 and 1.60 seconds, respectively.

Finally, we wanted to check a liveness property, a property which would state that if certain conditions hold system will eventually go to a good state. For example we may want to check if the system will eventually leave the `TooLow` mode when `Inject` stays `On`. This property can be stated in CTL as:

$$\text{(SIS9)} \quad \mathrm{AG}(Inject \;\mathrm{AU}\; \neg TLow)$$

But this property does not hold for the system given in Figure 7. The reason is, there is no causality between safety injection process and water pressure readings. I.e., there is no feedback between the control software and the environment (Figure 10). What we would like to do is, then, to change the environment constraints to reflect the effect of the safety injection process, and check the liveness property under the new environment constraints.

Using the notation in Figure 7 and Figure 8, we can do this quite easily. We just have to change the constraint $Feasible$. Consider the following redefinition of $Feasible$:

$$\begin{aligned} Feasible \stackrel{\text{def}}{=} \; & (\neg Inject \;\wedge\; wp1 - bound \leq wp1' \leq wp1 + bound \;\wedge\; wp2 - bound \leq wp2' \leq wp2 + bound \;\wedge \\ & wp3 - bound \leq wp3' \leq wp3 + bound \;\wedge\; min \leq wp1', wp2', wp3' \leq max) \;\vee \\ & (Inject \;\wedge\; wp1 < wp1' \leq wp1 + bound \;\wedge\; wp2 < wp2' \leq wp2 + bound \;\wedge \\ & wp3 < wp3' \leq wp3 + bound \;\wedge\; min \leq wp1', wp2', wp3' \leq max) \end{aligned}$$

This constraint basically states that water pressure increases when the safety injection is on. Now, we can try to verify our liveness property using the new $Feasible$ constraint. But, when we run our model checker on this system it does not converge. The reason is that since all the constants in the system are uninterpreted, there is no fix number of steps after which the system will reach $\neg TLow$. Hence, the fixpoint computations do not converge after fixed number of steps. This problem maybe solved using conservative approximation techniques such as widening. Unfortunately, a standard application of the widening technique would not work since we need a lower bound for a least fixpoint. However, an approach which combines automated theorem proving with widening technique to guess and verify least fixpoints may work [42]. Another solution would be to force water pressure to increase with a fixed amount when safety injection is on. For example we can redefine *Feasible* as

$$
\begin{aligned}
Feasible \stackrel{\text{def}}{=}\ & (\neg Inject\ \wedge\ wp1 - bound \le wp1' \le wp1 + bound\ \wedge\ wp2 - bound \le wp2' \le wp2 + bound\ \wedge \\
& wp3 - bound \le wp3' \le wp3 + bound\ \wedge\ min \le wp1', wp2', wp3' \le max) \vee \\
& (Inject\ \wedge\ wp1' = wp1 + (low - min)\ \wedge\ wp2' = wp2 + (low - min) \wedge \\
& wp3' = wp3 + (low - min)\ \wedge\ min \le wp1', wp2', wp3' \le max)
\end{aligned}
$$

This will force the system to get out of `TooLow` mode after one execution step. Using this constraint we tried to check the liveness property (SIS9) again. This time the liveness property was verified in 32.11 seconds.

# 8 A Transport Protocol

In this section we will demonstrate the effectiveness of our methods by analyzing a protocol. The protocol we will present is a one-directional transport protocol, abstracted from the TCP specification (RFC 793). Although abstractions and simplifications have been made, much of TCP's complexity remains. We analyze parts of the protocol, specifically: handshaking for call setup and take-down, as well as reliable data-transfer using sliding-window protocol [55] over an unreliable channel, which can delay, duplicate, lose or reorder messages to an unlimited extent (this is modeled using existential quantification). Moreover, the protocol's send and receive window sizes are not specified in advance; rather, they are represented as symbolic constants. The protocol system contains a mixture of integer and boolean-valued variables. Hence, it was not amenable to verification by a BDD-checker alone; moreover, its size (and the number of boolean variables) made pure-Presburger checking infeasible. In this section, we show how the system was automatically verified using our composite model checking approach, in concert with a conservative approximation technique.

In [46], a different version of the sliding-window protocol [55] was verified, using a compositional preorder approach. In certain respects, this previous study was more broad, e.g., it handled liveness properties for arbitrary channel lengths. In other respects, it was more constrained, e.g., it assumed a fixed window size, whereas we verify the problem for any window size; also sliding-window protocol forms only a part of the system we are analyzing (it is used for data transfer after the connection is established). But essentially, while examining a similar problem, the work in [46] and our approach illustrate two different (but related) techniques. In [46], verification is carried out in a semi-automated fashion, where some key user-generated abstractions were required. Here, the protocol is verified automatically, via composite model checking. In the future, we would expect to see many of these concepts used together, toward solving more complex problems.

Figure 11 shows how synchronization and data-transfer are implemented in our transport protocol, which involves one sender process and one receiver process. A sender can be in one of the following states: `closed`, `syn_sent`, `established` , and `fin_wait`, whereas the receiver process will either be in `listen`, or `established` states.

After the connection is established via hand-shaking, and both parties are in the `established` state, they use a sliding-window protocol to send and receive data, where each message has a unique sequence number. Here, it is sufficient to model the sequence numbers, as well as the send and receive pointers, and to abstract away the message contents.

As specified in TCP, a sender has two integer pointers $snd\_una$ (oldest unacknowledged sequence number), and $snd\_nxt$ (next sequence number to be sent). On the other hand, the receiver maintains a single integer pointer $rcv\_nxt$ (next sequence number to be received). Again, these variables can be unbounded; hence, it is not hard to see how the underlying transition system is an infinite one, and why it is not amenable to known automated techniques.

The integers $snd\_wnd$ and $rcv\_wnd$ denote the sizes of sender and receiver windows, respectively. These

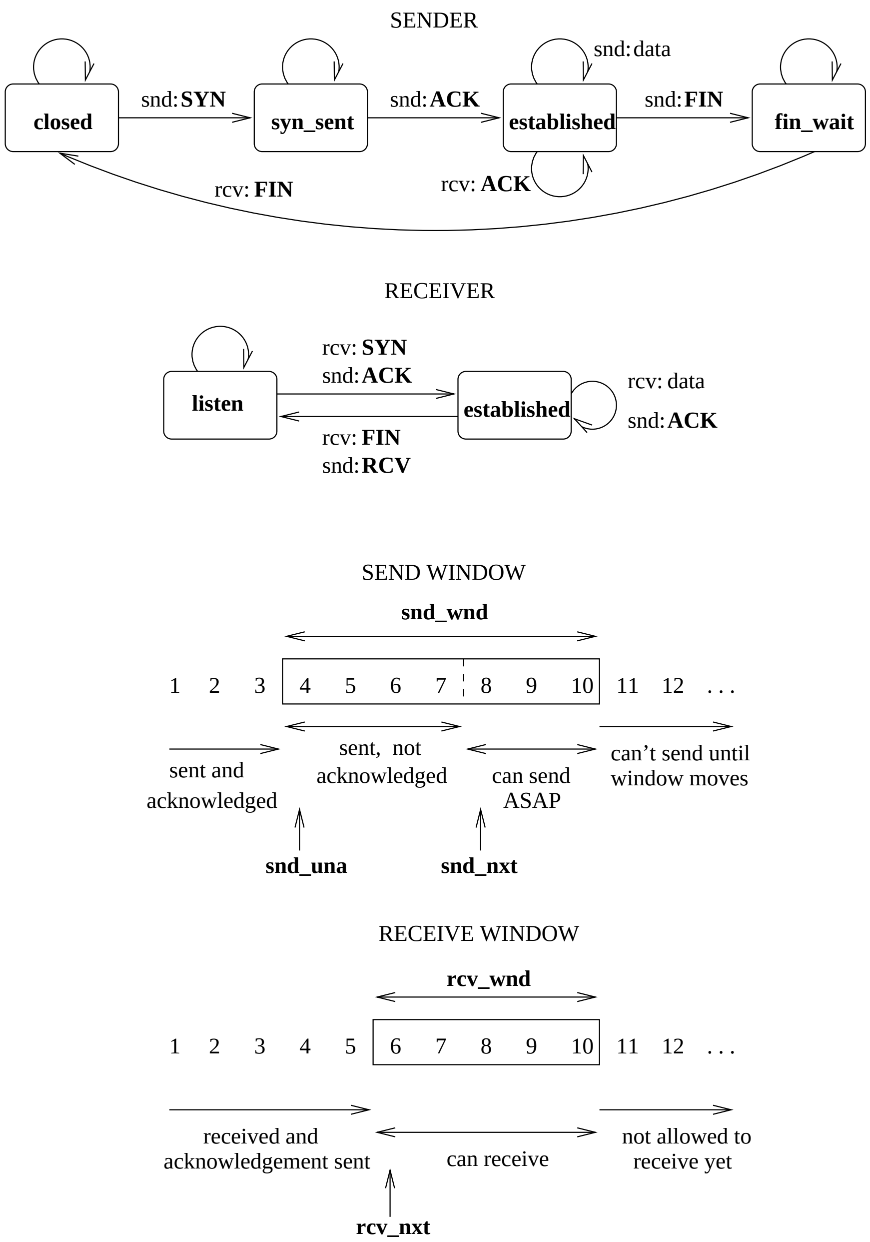


Figure 11: A Transport Protocol

```
typedef enum { closed, syn_sent, established, fin_wait } sender_state;
typedef enum { listen, established } receiver_state;
typedef enum { syn, ack, snt, fin } signal;
typedef enum { send, receive_ack, finish } event;

int snd_una, snd_nxt, rcv_nxt;
int snd_wnd, rcv_wnd;
int seg_seq, seg_len, seg_ack;
sender_state sender;
receiver_state receiver;

signal signal_from_net();
int seq_num_from_net();
int seg_len_from_net();
to_net( signal sent_signal, int seq_num, int seg_len );

event wakeup_event();
```

Figure 12: A Transport Protocol Implementation (Declarations)

are unspecified constants; hence, a property verified for the protocol will hold for any interpretation of these constants. As usual, the windows are used to avoid buffer overflow on the sender and receiver sides. The sender can have *snd_wnd* messages outstanding – a situation maintained by ensuring that $snd\_nxt - snd\_una \leq snd\_wnd$. Also, the receiver can only queue up *rcv_wnd* messages, and additional messages will be lost.

Figures 12, 13 and 14 demonstrate a possible implementation of the protocol.

## 8.1 Event-Action Language Specification of the Transport Protocol

Figures 15, 16 and 17 show our event-action language description of the protocol. This is the model we analyze with our composite model checker. Below we describe the behavior of the protocol by explaining the events individually.

Upon start-up, the sender forwards a *syn* message to the receiver, and then it goes into the `syn_sent` state (event $e_{S1}$). When the receiver gets the *syn*, it acknowledges via an *ack*; then it goes into the `established` state (event $e_{R1}$). When the send party receives the *ack*, it likewise transitions into the `established` state ($e_{S2}$). Note that both parties initialize their sequence number pointers to 0 before starting data-transfer, which works in a "stop-and-wait" manner: The sender repeatedly sends messages within its current window (event $e_{S3}$), and updates *snd_una* when acknowledgments are received ($e_{S4}$). The receiver, on the other hand, attempts to receive new messages within its own window (event $e_{R2}$); when the input queue bound is exceeded, the excess is dropped.

To close a connection, the sender sends a $finS$ message, and goes to the `fin_wait` state (event $e_{S5}$). When receiver receives this signal, it responds with a $finR$, and then it goes back to `listen`. Finally, when sender receives this signal, it goes back to `closed` state, and the hand-shake is complete.

We assume that the communication channel is unreliable, i.e., it can delay, lose, duplicate or reorder outstanding messages. To model this, we use shared variables and existential quantification, which include all the possible behaviors such a channel would generate. Also, note the presence of an idle event – which is used to delay messages for an arbitrary number of transitions before being received (if ever). We do assume, however, that there are no delayed duplicates in the network – i.e., when a new session of the protocol is instantiated, there are no previous *syn*, $fin$, *snt*, and *ack* signals hanging around in the network.

```
sender_process()
{
  switch ( sender )
  {
    case closed:
      to_net( syn, dummy, dummy );
      sender = syn_sent;
      break;
    case syn_sent:
      if ( signal_from_net() == ack )
      {
        sender = established;
        snd_una = snd_nxt = 0;
      }
      break;
    case established:
      switch  ( wakeup_event() )
      {
        case send:
          if ( snd_nxt + seg_len <= snd_una + snd_wnd )
          {
            to_net( snt, snd_nxt, seg_len );
            snd_nxt = snd_nxt + seg_len;
          }
          else
            print_error( "message length exceeds send window" );
          break;
        case receive_ack:
          seg_ack = seq_num_from_net();
          snd_una = seg_ack;
          break;
        case finish:
          to_net( fin, dummy, dummy );
          sender = fin_wait;
      }
      break;
    case fin_wait:
      if ( signal_from_net() == fin )
        sender = closed;
      break;
  }
}
```

Figure 13: A Transport Protocol Implementation (Sender Process)

```
receiver_process()
{
  switch ( receiver )
  {
    case listen:
      if ( signal_from_net() == syn )
      {
        receiver = established;
        rcv_nxt = 0;
        to_net( ack, dummy, dummy );
      }
      break;
    case established:
      if ( signal_from_net() == snt )
      {
        seg_seq = seq_num_from_net();
        seg_len = seg_len_from_net();

        if ( (rcv_nxt <= seg_seq < rcv_nxt + rcv_wd) ||
             (rcv_nxt <= seg_seq + seg_len - 1 < rcv_nxt + rcv_wnd) )
        {
          if ( seg_seq + seg_len > rcv_nxt + rcv_wnd )
            rcv_nxt = rcv_nxt + rcv_wnd;
          if ( seg_seq + seg_len <= rcv_nxt + rcv_wnd )
            rcv_nxt = seg_seq + seg_len;
          to_net( ack, rcv_nxt, dummy );
        }
        else
          print_error( "message is outside of the receive window" );
      }
      else if ( signal_from_net() == fin )
      {
        receiver = listen;
      }
      break;
  }
}
```

Figure 14: A Transport Protocol Implementation (Receiver Process)

PROGRAM A Transport Protocol
CONSTANTS
  $snd\_wnd$: positive integer // sender window
  $rcv\_wnd$: positive integer // receiver window
VARIABLES
  $sender$ : $\{\texttt{closed}, \texttt{syn\_sent}, \texttt{established}, \texttt{fin\_wait}\}$ // sender state
  $receiver$ : $\{\texttt{listen}, \texttt{established}\}$ // receiver state
  $snd\_una$: positive integer // oldest unacknowledged sequence number
  $snd\_nxt$: positive integer // next sequence number to be sent
  $rcv\_nxt$: positive integer // next sequence number to be received
  $syn$: boolean // if true, then sender has sent a syn
  $ack$: boolean // if true, then receiver has sent an ack
  $snt$: boolean // if true, then receiver has sent data
  $finS$: boolean // if true, then sender has sent a fin
  $finR$: boolean // if true, then receiver has sent a fin
INITIAL CONDITION
  $snd\_una = snd\_nxt = rcv\_nxt = 0 \ \wedge\ sender = \texttt{closed} \ \wedge\ receiver = \texttt{listen}$
  $\wedge\ syn = ack = snt = finS = finR = \textbf{false}$

Figure 15: Variable Declarations for the Transport Protocol

EVENTS
// SENDER EVENTS
$e_{S1}$ : // sender goes from closed to syn_sent
$sender = \texttt{closed} \ \wedge \ sender' = \texttt{syn\_sent} \ \wedge \ syn' = \textbf{true} \ \wedge \ finS' = \textbf{false}$
$e_{S2}$ : // sender receives ack and goes from syn_sent to established
$ack = \textbf{true} \ \wedge \ sender = \texttt{syn\_sent} \ \wedge \ sender' = \texttt{established} \ \wedge \ syn' = \textbf{false}$
$\wedge \ snd\_una' = snd\_nxt' = 0$
$e_{S3}$ : // sender sends data in established
$sender = \texttt{established} \ \wedge \ snt' = \textbf{true} \ \wedge$
$(\exists seg\_len \ : \ 0 < seg\_len \ \wedge \ snd\_nxt + seg\_len \leq snd\_una + snd\_wnd$
$\wedge \ snd\_nxt' = snd\_nxt + seg\_len) \ \vee \ snd\_nxt' = snd\_nxt$
$e_{S4}$ : // sender receives ack in established
$sender = \texttt{established} \ \wedge$
$(\exists seg\_ack : 0 \leq seg\_ack \ \wedge \ seg\_ack \leq rcv\_nxt \ \wedge \ snd\_una < seg\_ack \leq snd\_nxt$
$\wedge \ snd\_una' = seg\_ack) \ \vee \ snd\_una' = snd\_una$
$e_{S5}$ : // sender goes from established to fin_wait
$sender = \texttt{established} \ \wedge \ sender' = \texttt{fin\_wait} \ \wedge \ finS' = \textbf{true}$
$e_{S6}$ : // sender goes from fin_wait to closed after receiving fin
$finR = \textbf{true} \ \wedge \ sender = \texttt{fin\_wait} \ \wedge \ sender' = \texttt{closed} \ \wedge$
$syn' = finS' = snt' = \textbf{false}$

Figure 16: Sender Events for the Transport Protocol

EVENTS
// RECEIVER EVENTS
$e_{R1}$ : // receiver goes from listen to established after receiving syn
$syn = \textbf{true} \ \wedge \ receiver = \texttt{listen} \ \wedge \ receiver' = \texttt{established} \ \wedge$
$ack' = \textbf{true} \ \wedge \ finR' = \textbf{false} \ \wedge \ rcv\_nxt' = 0$
$e_{R2}$ : // receiver receives data in established
$snt = \textbf{true} \ \wedge \ receiver = \texttt{established} \ \wedge$
$(\exists seg\_seq, seg\_len \ : \ 0 \leq seg\_seq \ \wedge \ 0 < seg\_len \ \wedge \ rcv\_wnd > 0 \ \wedge$
$(seg\_seq + seg\_len \leq snd\_nxt) \ \wedge \ (rcv\_nxt \leq seg\_seq < rcv\_nxt + rcv\_wnd \ \vee$
$rcv\_nxt \leq seg\_seq + seg\_len - 1 < rcv\_nxt + rcv\_wnd) \ \wedge$
$((seg\_seq + seg\_len > rcv\_nxt + rcv\_wnd \ \wedge \ rcv\_nxt' = rcv\_nxt + rcv\_wnd) \ \vee$
$(seg\_seq + seg\_len \leq rcv\_nxt + rcv\_wnd \ \wedge \ rcv\_nxt' = seg\_seq + seg\_len))) \ \vee$
$\vee \ rcv\_nxt' = rcv\_nxt$
$e_{R3}$ : // receiver goes from established to listen after receiving fin
$finS = \textbf{true} \ \wedge \ receiver = \texttt{established} \ \wedge \ receiver' = \texttt{listen} \ \wedge$
$finR' = \textbf{true} \ \wedge \ ack' = \textbf{false}$
// IDLE EVENT
$e_I$ : // receiver and sender events can be delayed indefinitely
**true**

Figure 17: Receiver and Idle Events for the Transport Protocol

| PROP. | FIXPOINT | CONVERGED IN |
|---|---|---|
| (TP1) | Exact | 4 iterations – 3.11 sec. |
| | Approximate | 4 iterations – 3.42 sec. |
| (TP2) | Exact | did not converge |
| | Approximate | 6 iterations – 14.13 sec. |
| (TP3) | Exact | 1 iterations – 1.74 sec. |
| | Approximate | 1 iterations – 1.93 sec. |
| (TP4) | Exact | 1 iterations – 1.73 sec. |
| | Approximate | 1 iterations – 1.74 sec. |
| (TP5) | Exact | 7 iterations – 91.97 sec. |
| | Approximate | 4 iterations – 11.52 sec. |
| (TP6) | Exact | did not converge |
| | Approximate | 10 iterations – 377.44 sec. |

Table 1: Verified Properties for the Transport Protocol

## 8.2 Automated Analysis of the Protocol

We analyzed the transport protocol presented above using the composite model checking technique. We used both exact and approximate fixpoint computations. We applied the widening technique to composite representations as described in Section 5. We disjunctively decomposed the transition relation using the event syntax.

Some interesting properties of this data transfer protocol are as follows:

- (TP1) $\mathrm{AG}(\mathit{sender} = \texttt{established} \rightarrow \mathit{receiver} = \texttt{established})$ :

  If the sender establishes a connection, then so does the receiver.

- (TP2) $\mathrm{AG}(\mathit{receiver} = \texttt{established} \rightarrow \neg(\mathit{sender} = \texttt{closed}))$ : If the receiver establishes a connection, then the sender is not closed.

- (TP3) $\mathrm{AG}(\mathit{sender} = \texttt{established} \rightarrow \mathit{snd\_una} \leq \mathit{snd\_nxt} \leq \mathit{snd\_una} + \mathit{snd\_wnd})$ : The sender does not overload the send buffer.

- (TP4) $\mathrm{AG}((\mathit{receiver} = \texttt{established} \wedge \mathit{rcv\_nxt} = i) \rightarrow \mathrm{AX}(\mathit{rcv\_nxt} \leq i + \mathit{rcv\_wnd}))$ : The receiver does not overload the receive buffer.

- (TP5) $\mathrm{AG}(\mathit{sender} = \mathit{receiver} = \texttt{established} \rightarrow \mathit{snd\_una} \leq \mathit{rcv\_nxt})$ : All the acknowledged messages are received by the receiver.

- (TP6) $\mathrm{AG}(\mathit{sender} = \mathit{receiver} = \texttt{established} \rightarrow \mathit{rcv\_nxt} \leq \mathit{snd\_nxt})$ : All messages received were actually sent by the sender.

Figure 1 shows the results of our experiments. Property (TP1) converged in 4 iterations, using both exact and approximate fixpoint computations; and due to additional cost of widening, the approximate fixpoint actually used a bit more time. However, property (TP2) does not converge at all using the exact method – even though the property does not use any integer variables. However, this is not too surprising, since within the protocol, almost all state-changes are due to a combination of interacting variables, both booleans and integers. We were able to prove property (TP2) using the approximate fixpoint computations.

On the other hand, properties (TP3) and (TP4) are both built up exclusively using integer variables – yet, both converge exactly in one iteration; indeed they follow from the event specifications of the sender and the receiver.

Property (TP5) establishes the relationship between *snd_una* and *rcv_nxt*. Since *snd_una* is updated by the sender, and since *rcv_nxt* is updated by the receiver, verification of this property involves considering all possible concurrent executions. The model-checker verifies the property in 7 iterations using exact fixpoint computations, whereas the approximate fixpoint computations converges much faster in 4 iterations. This shows that conservative approximations are not only useful for approximating divergent fixpoint computations; they can also be used to get quicker results in general.

Property (TP6) is another property which involves both the sender and the receiver processes, and it shows the relationship between variables *rcv_nxt* and *snd_nxt*. Again *rcv_nxt* is updated by the receiver, and *snd_nxt* is a variable updated by the sender. While exact analysis did not converge for this property, the approximate fixpoint computations converged in 10 iterations.

# 9 Conclusions

We presented a composite model checking approach for combining different type-specific symbolic representations in a single model checker. To do this, we conjunctively partition the atomic events of the system based on the underlying variable types, and then compute each type's pre-image separately. We use some simple rules for handling the logical connectives over multiple types.

We presented a prototype tool which implements these ideas by combining the relative strengths of two different symbolic representations: BDDs and Presburger formulas. We applied this technique to a non-trivial SCR requirements specification for a nuclear reactor's cooling system, and a transport protocol. Both systems contain many boolean and enumerated variables, as well as multiple unbounded integers. In situations like these, the extra overhead involved in processing the composite model is well worth the time spent. In fact, our first integer-oriented tool (exclusively limited to Presburger constraints) quickly ran out of memory when subjected to these specifications – when all the booleans enumerated types were mapped to integer variables. On the other hand, it would be impossible to validate these systems using a finite-state model checker, without either bounding the variable domains, or using some other abstraction.

We are interested in expanding our composite model checker by adding other symbolic representations such as automata based representations [11, 12, 8, 10, 37, 41, 50] and real arithmetic [2, 1]. This will allow us to encode variable types like reals and queues. Our composite model can also be useful in analyzing systems with nonlinear constraints. In our current system, multiplication cannot be expressed in basic properties and atomic event descriptions. Using a composite model we can isolate the parts of a system with nonlinear constraints, and deal with them separately, possibly using approximation techniques. It would be interesting to investigate the scalability of our approach to several symbolic representations.

Also our conservative approximation techniques can be extended. There are two basic ways of approximating truth sets of temporal formulas: (1) Using approximate fixpoint computations (this is the approach we adopted), and (2) using approximate models which over- or under-estimate behaviors of a system [35, 47]. These two techniques can be combined to form a model checker that would be capable of both compositional and incremental analysis. We currently compute the fixpoints over the Cartesian-product of all variable domains. Using the conservative approximation approach, we could integrate compositional strategies developed for finite-state systems [16, 30] in our analysis.

Finally, we would like to test presented techniques on larger systems, and determine their feasibility for checking "industrial-strength" examples. We think that using various abstraction techniques – in conjunction with some human interaction – we should be able to attack these significantly larger systems.

# References


[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, X. Nicollin P. H. Ho, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34,

1995.

[2] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Richel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[3] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[4] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 156–166, October 1996.

[5] A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[6] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 280–292, January 1996.

[7] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[8] D. A. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Proceedings of the 7th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[9] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proceedings of First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.

[10] M. Biehl, N. Klarlund, and T. Rauhe. Mona: Decidable arithmetic in practice. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium*, volume 1135 of *Lecture Notes in Computer Science*. Springer, 1996.

[11] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, August 1996.

[12] B. Boigelot, P. Godefroid, B. Williams, and P. Wolper. The power of QDDs. In *Proceedings of the Fourth Static Analysis Symposium*, September 1997.

[13] J. C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhauser, Boston, 1992.

[14] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[15] R. E. Bryant and Y. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, June 1995.

[16] T. Bultan, J. Fischer, and R. Gerber. Compositional verification by model checking for counter-examples. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 224–238, January 1996.

[17] T. Bultan and R. Gerber. Composite model checking with type specific symbolic encodings. Technical Report CS-TR-3871, Department of Computer Science, University of Maryland, College Park, February 1998.

[18] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, March 1998.

[19] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[20] T. Bultan, R. Gerber, and W. Pugh. Model checking concurrent systems with unbounded integer variables: Symbolic representations, approximations and experimental results. Technical Report CS-TR-3870, Department of Computer Science, University of Maryland, College Park, February 1998.

[21] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on Very Large Scale Integration*, August 1991.

[22] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[23] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, January 1990.

[24] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 316–327. Springer, June 1997.

[25] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[26] E. Clarke, M. Fujita, and X. Zhao. Applications of multi-terminal binary decision diagrams. Technical Report CMU-CS-95-160, School of Computer Science, Carnegie Mellon University, April 1995.

[27] E. Clarke and X. Zhao. Word level symbolic model checking: A new approach for verifying arithmetic circuits. Technical Report CMU-CS-95-161, School of Computer Science, Carnegie Mellon University, May 1995.

[28] 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*, 8(2):244–263, April 1986.

[29] E. M. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. Technical Report CMU-CS-94-204, School of Computer Science, Carnegie Mellon University, October 1994.

[30] E. M. Clarke, D. E. Long, , and K. L. McMillan. Compositional model checking. In *Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science*, pages 464–475, June 1989.

[31] P. J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proceedings of the 15th International Conference on Software Engineering*, pages 315–323, May 1993.

[32] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[33] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming*, pages 84–97, 1978.

[34] R. Cridlig. Semantic analysis of concurrent ML by abstract model-checking. In *Proceedings of INFINITY International workshop on Infinite State Systems, Technical Report MIP-9614*, pages 71–86. University of Passau, 1996.

[35] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[36] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[37] P. Godefroid and D. Long. Symbolic protocol verification with queue BDDs. In *Proceedings of the 11th Symposium on Logic in Computer Science*, pages 198–206, July 1996.

[38] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Proceedings of computer aided verification*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346. Springer-Verlag, 1993.

[39] N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *Proceedings of International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1994.

[40] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[41] J. G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop*, volume 1019 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[42] T. A. Henzinger and V. Rusu. Reachability verification for hybrid systems. In *Proceedings of the International Workshop on Hybrid Systems: Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, April 1998.

[43] D. Jackson. Nitpick: A checkable specification language. In *Proceedings of Workshop on Formal Methods in Software Practice*, January 1996.

[44] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 224–238, January 1996.

[45] D. Jackson, Y. Ng, and J. Wing. A nitpick analysis of mobile ipv6. Technical Report CMU-CS-98-113, School of Computer Science, Carnegie Mellon University, March 1998.

[46] R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 48–59. Springer, June 1997.

[47] P. Kelb, D. Dams, and R. Gerth. Practical symbolic model checking of the full $\mu$-calculus using compositional abstractions. Technical Report 95-31, Department of Computer Science, Eindhoven University of Technology, 1995.

[48] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.

[49] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[50] N. Klarlund. Mona & fido: The logic-automaton connection in practice. In *Proceedings of Computer Science Logic*, 1997.

[51] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.

[52] A. Pardo and G. D. Hachtel. Automatic abstraction techniques for propositional $\mu$-calculus model checking. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 12–23. Springer, June 1997.

[53] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–104, August 1992.

[54] T. Sreemani and J. M. Atlee. Feasibility of model checking software requirements: A case study. In *Proceedings of the 11th Annual Conference on Computer Assurance*, pages 77–88, June 1996.

[55] N. V. Stenning. A data transfer protocol. *Computer Networks*, 1:99–110, 1976.

[56] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[57] J. M. Wing and M. Vaziri-Farahani. Model checking software systems: A case study. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, 1995.