Verifying Systems with Integer Constraints and Boolean Predicates: A Composite Approach

Tevfik Bultan Richard Gerber Christopher League

Department of Computer Science
University of Maryland
College Park, MD 20742, USA
{bultan,rich,league}@cs.umd.edu

Abstract

Symbolic model checking has proved highly successful for large finite-state systems, in which states can be compactly encoded using binary decision diagrams (BDDs) or their variants. The inherent limitation of this approach is that it cannot be applied to systems with an infinite number of states – even those with a single unbounded integer.

Alternatively, we recently proposed a model checker for integer-based systems that uses Presburger 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 possess no natural mapping to the Euclidean coordinate space.

In this paper we describe a model checker which combines the strengths of both approaches. We use a composite model, in which a formula's valuations are encoded in a mixed BDD-Presburger form, depending on the variables used. We demonstrate our technique's effectiveness on a nontrivial requirements specification, which includes a mixture of Booleans, integers and enumerated types.

1 Introduction

Symbolic model checking has proved highly successful for verifying large finite-state systems [12, 20]. This success is partially due to the advent of innovative data structures like Binary Decision Diagrams (or BDDs), which can encode huge sets of bit-vector states in a highly compact format [9]. A fortunate consequence of the BDD structure is that it immediately supports efficient Boolean-algebraic operators (e.g., conjunction, negation, etc.) – which also happen to be the main operators used in symbolic model checking.

This research is supported in part by ONR grant N00014-94-10228 and NSF Young Investigator Award CCR-9357850.

Unfortunately, none of the BDD-based techniques can be used for potentially 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.

Recently, we proposed a model checker for general integer based systems, which uses Presburger constraints as its underlying state representation [11]. As with BDDs for Boolean arrays, Presburger constraints can compactly represent huge (even unbounded) sets of integer states over multiple dimensions. Specifically, our model checker represents sets of state-valuations using unions of convex polytopes, each of which is formed by affine 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 [19, 21] for this purpose).

While many model checking queries are undecidable for general infinite-state theories, we often appeal to conservative approximation techniques, many of which guarantee convergence by allowing false negatives. With this approach we were able to easily verify some nontrivial infinite-state programs from the concurrency literature. These programs (and their associated formulas) are the type usually analyzed with hand proofs, due to the subtle way their infinite-state variables influence their control flow.

However, our Presburger technique proved ill-suited for handling Boolean and (unordered) enumerated types. When all state sets are represented as Presburger constraint expressions, all Boolean variables end up getting 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 affine 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 $\{red, green, blue\}$. Unfortunately, these are exactly the datatypes often used for describing modes in requirements specifications.

In this paper we describe our solution to this problem, in the form of a model checker which combines the relative strengths of both BDDs and Presburger formulas. Specifically, we use a composite model to represent a formula's valuations – which are encoded using tuples of BDDs and affine constraints, depending on the variables appearing in the sub-formulas.

The key to our framework rests on some fundamental observations. Given a system whose state transformations (and requirements) exclude (1) arbitrary functions over mixed types, and (2) type-coersions (e.g., to allow using Booleans in arithmetic operators), we (a) orthogonally partition the system's state-space into sets of conjuncts, each of which contain a Boolean part and an integer part; (b) manipulate a Boolean state and its transformation independently, without interference from the integer part (and vice versa); and also (c) use semantically sound rules for handling the logical connectives that involve multiple types.

In formal terms, this partitioning allows pre-image computations to distribute across the conjunction operator. We exploit this property in our implementation, which is structured in a layered class hierarchy. The foundation is composed of two large libraries for encoding type-specific constraints, and their associated set-theoretic operations. While both share a similar API, one of them (our BDD implementation) is used exclusively for Boolean and enumerated types, while the other is the Omega library, used for integer-valued variables and their constraints. At the next level is our composite-model library, which handles operations over mixed-type constraints (e.g., set-inclusion, intersection, etc.); in turn, these operations invoke their relevant type-specific counterparts to help carry out the desired effect. At the topmost level is the model checker's class library, which imports the composite-model operations, and exports functions for parsing, composition, reachability analysis, and liveness tests. Our approach to mixed constraints – and their orthogonal implementations – will hopefully allow us to expand to additional datatypes in the future.

In the sequel, we demonstrate our results using an "enhanced" version of a known SCR specification, which states the requirements for a reactor's water pressure system [7, 16, 18]. 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:

$$\begin{aligned}&(\mathrm{temp} > \mathrm{High} \longleftrightarrow \mathrm{Alarm})\\ &\quad \rightarrow (\exists\ \mathrm{x} < 0 :: \mathrm{temp} + \mathrm{x} = \mathrm{High} \rightarrow \mathrm{Alarm}).\end{aligned}$$

With this capability, we need not map SCR integer predicates to Boolean literals (e.g., as in [3, 4, 5, 13]); 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").

There has been significant work in using model checking to verify tabular-style SCR requirements. In [5] Atlee and Gannon mapped queries about SCR mode-transition tables to the MCB model checker, which uses explicit state enumeration as its underlying representation. Later, Atlee and Buckley improved this process, by using SMV [4] for the same purpose. Since SMV 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 was also used to analyze parts of the RSML specification of the TCAS II system [3]. The main difficulty in using SMV for checking requirements specifications seems to be that every variable gets represented in the same symbolic format, namely BDDs. This can easily result in inefficient encodings of totally-ordered variables; moreover, the resulting number of BDD nodes (not to mention their inherent finiteness) often makes verification untenable, at least without human-guided abstractions.

In [13], 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 [4, 5]). 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 [13] that this restriction is partly motivated by the semantics of RSML, and it allows modeling of a significant portion of TCAS II system.

In [7] 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.

Other state-encodings have been explored for model checking in various domains, and we note some of these efforts here. For example Alur *et al.* used arithmetic constraints on real variables to check properties of hybrid systems [1, 2]. Other recent results deal with analyzing particular systems, e.g., for systems with FIFO queues (Queue Decision Diagrams [8]), for simultaneous control/datapath verification in VLSI domains (Binary Moment Diagrams [10] and Hybrid Decision Diagrams [15]), etc. Again, our preliminary results suggest that we could potentially combine all these symbolic representations in one composite representation.

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`)

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 1: SCR Specification of the Safety Injection System.

The remainder of this paper is organized as follows. First, we introduce our motivating SCR example, and explain how we enhanced its behavior to make the verification problem more challenging. Then, we discuss our composite model representation, and show how we translate the SCR requirements into our underlying transition language. Next, we show how our symbolic model checker works in general, and also the results it achieved for the SCR specification. We conclude with some remarks on the results, and outline our future research directions.

2 An Example: Safety Injection System

As an example, we analyze the requirements for a reactor's cooling system; this example was adapted from previous specifications in [7, 16, 18] – and in fact, we take a superset of these requirements, as well as add a few of our own. The target application is called an "Engineered Safety Feature Actuation System," for a PWR Nuclear Power Plant. It basically functions as a feedback loop: its sensors monitor the coolant system's water pressure via three redundant channels. When the pressure is determined to fall below a certain threshold, the processor conveys signals to pressure-control actuators with the objective of increasing it (this is called "safety injection"). However, the operator may use a manual control (called the "block" switch) to override safety injection during normal start-up or cool-down phases. Moreover (as in [7]), the operator also has a "reset switch," which results in the controller program clearing any state triggered by outstanding "block" signals.

In [16] and [7] this specification is rendered using the SCR tabular notation – although the system given in [7] uses only one pressure sensor. Here, we adapt the original three-sensor system to the style of requirements in [7], assuming that an action is taken when two out of three sensors agree on a condition.

Additionally, we have complicated the system somewhat, by adding an actuator called "damp," which is similar to "safety inection" – 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 behaviorially, in a table denoting current-state/next-state transitions, labelled by conditions that trigger the state-change. A *term* is any function of modes, vari-

Constants: $min, max, low, high, toohigh, bound$: integer $\quad min < low < high < toohigh < max$
Variables: $wp1, wp2, wp3$: integer
$Block, Reset, Inject, Damp, Over, TLow, Low, High, THigh$: boolean
Initial Condition: $low \leq wp1, wp2, wp3 < high \wedge \neg Block \wedge \neg Reset \wedge \neg Inject \wedge \neg Damp \wedge \neg Over$
Events:

$$e_{TLow}: \; RTLow \wedge TLow' \wedge \neg Low' \wedge \neg High' \wedge \neg THigh' \wedge Feasible \wedge FOver \wedge FInject \wedge FInject' \wedge FDamp \wedge FDamp'$$

$$e_{Low}: \; RLow \wedge Low' \wedge \neg TLow' \wedge \neg High' \wedge \neg THigh' \wedge Feasible \wedge FOver \wedge FInject \wedge FInject' \wedge FDamp \wedge FDamp'$$

$$e_{High}: \; RHigh \wedge High' \wedge \neg TLow' \wedge \neg Low' \wedge \neg THigh' \wedge Feasible \wedge FOver \wedge FInject \wedge FInject' \wedge FDamp \wedge FDamp'$$

$$e_{THigh}: \; RTHigh \wedge THigh' \wedge \neg TLow' \wedge \neg Low' \wedge \neg High' \wedge Feasible \wedge FOver \wedge FInject \wedge FInject' \wedge FDamp \wedge FDamp'$$

$$e_{NoChange}: \; ((CTLow' \wedge CTLow) \vee (CLow' \wedge CLow) \vee (CHigh' \wedge CHigh) \vee (CTHigh' \wedge CTHigh)) \wedge Feasible \wedge FOver \wedge FInject \wedge FInject' \wedge FDamp \wedge FDamp'$$

$$e_{BlockOrReset}: \; ((Block' = \neg Block \wedge Reset' = Reset) \vee (Reset' = \neg Reset \wedge Block' = Block)) \wedge FOver \wedge FInject \wedge FInject' \wedge FDamp \wedge FDamp'$$

$$CTLow \stackrel{\text{def}}{=} wp1, wp2 < low \vee wp1, wp3 < low \vee wp2, wp3 < low$$
$$CLow \stackrel{\text{def}}{=} low \leq wp1, wp2 < high \vee low \leq wp1, wp3 < high \vee low \leq wp2, wp3 < high$$
$$CHigh \stackrel{\text{def}}{=} high \leq wp1, wp2 < toohigh \vee high \leq wp1, wp3 < toohigh \vee high \leq wp2, wp3 < toohigh$$
$$CTHigh \stackrel{\text{def}}{=} toohigh \leq wp1, wp2 \vee toohigh \leq wp1, wp3 \vee toohigh \leq wp2, wp3$$
$$CTLow' \stackrel{\text{def}}{=} wp1', wp2' < low \vee wp1', wp3' < low \vee wp2', wp3' < low$$
$$CLow' \stackrel{\text{def}}{=} low \leq wp1', wp2' < high \vee low \leq wp1', wp3' < high \vee low \leq wp2', wp3' < high$$
$$CHigh' \stackrel{\text{def}}{=} high \leq wp1', wp2' < toohigh \vee high \leq wp1', wp3' < toohigh \vee high \leq wp2', wp3' < toohigh$$
$$CTHigh' \stackrel{\text{def}}{=} toohigh \leq wp1', wp2' \vee toohigh \leq wp1', wp3' \vee toohigh \leq wp2', wp3'$$
$$RTLow \stackrel{\text{def}}{=} CTLow' \wedge \neg CTLow \quad RLow \stackrel{\text{def}}{=} CLow' \wedge \neg CLow$$
$$RHigh \stackrel{\text{def}}{=} CHigh' \wedge \neg CHigh \quad RTHigh \stackrel{\text{def}}{=} CTHigh' \wedge \neg CTHigh$$
$$Feasible \stackrel{\text{def}}{=} (wp1 - bound \leq wp1' \leq wp1 + bound) \wedge (wp2 - bound \leq wp2' \leq wp2 + bound) \wedge (wp3 - bound \leq wp3' \leq wp3 + bound) \wedge (min \leq wp1', wp2', wp3' \leq max)$$
$$\begin{aligned} FOver \stackrel{\text{def}}{=} &(Over' \wedge Block' \wedge Block \wedge \neg Reset \wedge (TLow \vee Low \vee THigh)) \vee \\ &(\neg Over' \wedge ((Reset' \wedge Reset) \vee (TLow' \wedge\ TLow) \vee (Low' \wedge \neg Low) \vee (High' \wedge \neg High) \vee (TooHigh' \wedge\ TooHigh))) \vee \\ &(Over' = Over \wedge (\neg(Block' \wedge Block \wedge \neg Reset \wedge (TLow \vee Low \vee High \vee THigh)) \vee \\ &\neg((Reset' \wedge Reset) \vee (TLow' \wedge\ TLow) \vee (Low' \wedge \neg Low) \vee (High' \wedge \neg High) \vee (THigh' \wedge\ THigh)))) \end{aligned}$$
$$FInject \stackrel{\text{def}}{=} (\neg Inject \wedge (TLow \wedge Over \vee Low \vee High \vee THigh)) \vee (Inject \wedge (TLow \wedge \neg Over))$$
$$FDamp \stackrel{\text{def}}{=} (\neg Damp \wedge (TLow \vee Low \vee High \vee THigh \wedge \neg Over)) \vee (Damp \wedge (THigh \wedge \neg Over))$$

Figure 2: Event-Action Language Representation of the Safety Injection System Requirements Specifications.

ables, 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 1. 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 out 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 implicity, 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 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 overriden. The conditions for `Damp` are analogous.

The semantics of SCR is defined in [18]. 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 1 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 [7]. Here, we ensure that readings can change within a certain range, ± `bound`.

Note that in the specification given in Figure 1 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.

3 Models and Properties

We use an event-action language as our syntax for concurrent systems, with a semantics defined in terms of states and transition relations. A concurrent system $C = (V, I, E)$ is represented by (1) a finite set of data and control variables V; (2) an initial condition I, which specifies the starting states of the program; and (3) a finite set of events E, where each event is considered atomic [22]. A system state is determined by the values of its data and control variables, where we assume that the domain of each variable is a countable set. Each event defines a transformation on the variables of the program.

Given a system $C = (V, I, E)$ in our event action language, we model it as an infinite transition system $M = (S, I, X, L)$, where S is the set of states, I is the set of initial states, $X \subseteq S \times S$ is the transition relation (derived from the set of events E), and $L : S \times SF \rightarrow \{true, false\}$ is the valuation function for state formulas over the program's variables. (We define the set of state formulas SF below.) The set of states S is obtained by taking Cartesian product of domains of all program variables; hence, each state corresponds to a valuation of all the variables of the program.

Every event $e \in E$ defines a binary relation on the program's states, $X_e \subseteq S \times S$, such that when $(s, s') \in X_e$, s and s' denote program's states before and after the execution of event e, respectively. We use $\mathbf{domain}(e)$ and $\mathbf{range}(e)$ to denote the domain and range of event e, i.e., $\mathbf{domain}(e) = \{s : \exists(s, s')[(s, s') \in X_e]\}$ and $\mathbf{range}(e) = \{s' : \exists(s, s')[(s, s') \in X_e]\}$. The global transition relation is $X = \bigvee_{e \in E} X_e$. Note that we use an interleaving model, where each transition represents execution of a single event, i.e., only one event can occur at a time.

See Figure 2, in which we give the representation of the safety injection system in our event-action language. We do not view this notation as a source language – rather, it is a low-level description of a set of transformations between states. After compilation, our analyzer uses these transformations to evaluate validity of temporal expressions. Note that if a variable v is not mentioned in the action of an event, then that event does not change its value, i.e., $v' = v$.

Using the formal semantics of SCR requirements specifications given in [18], 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`.

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 $\pm bound$. (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 $e_{NoChange}$ represents the transitions where water pressure conditions remain static. Finally, event $e_{BlockOrReset}$ defines the changes in the system entities when one of the variables $Block$ or $Reset$ changes.

3.1 Composite Formulas

We now define the set of *composite formulas*, which serve as the basic building blocks of our logic. Composite formulas are defined by the following grammar:

$$
\begin{array}{lll}
F & ::= & (F) \mid F \wedge F \mid \neg F \mid true \mid false \mid F^B \mid F^I \\
F^B & ::= & \mathbf{boolvar} \\
F^I & ::= & E^I \leq E^I \mid \exists \mathbf{intvar}\ F^I \\
E^I & ::= & (E^I) \mid E^I + E^I \mid \mathbf{intvar} \mid \mathbf{intcons}
\end{array}
$$

Here, the terminals **boolvar** and **intvar** represent Boolean and integer variables respectively – while the terminal **intcons** denotes an arbitrary integer constant. Using this base language, we can easily represent formulas including $<$, $=$, $\vee$, $\forall$, as well as multiplication by a constant.

We reason about a program C by using the composite formulas which range over C's program variables. We call these C's **state-formulas**, or SF. For example, one of the properties that the safety injection system should satisfy is:

> Whenever `Pressure` is `TooLow` then either `WP1`, `WP2` < `low`, or `WP1`, `WP3` < `low`, or `WP2`, `WP3` < `low`.

We can represent this in our model by asserting that the following formula should stay invariant over all executions:

$$
\begin{array}{ll}
TLow \rightarrow & (wp1, wp2 < low \vee wp1, wp3 < low \\
 & \vee\ wp2, wp3 < low).
\end{array}
$$

This composite formula uses terms from the safety injection system given in Figure 2.

Three crucial restrictions are placed on how composite terms can be used: (1) Booleans cannot be coerced to integers, and used in arithmetic operations (i.e., $+$ and $\leq$); (2) likewise, integer variables cannot be coerced into Boolean variables; and (3) the only function symbol allowed is the additive operator.

The importance of (1)-(2) will be understood shortly. Stipulation (3) means that the set of closed formulas is expressively equivalent to the *Presburger arithmetic* – a first-order theory for which validity and satisfiability is decidable in finite time. Hence, given a state formula $f \in SF$, and a program state s we can decide if $s \models f$ – by simply substituting the free variables in f by their values in s, and checking the result with a Presburger decision procedure. Note that while the worst-case time bound can be prohibitive for deciding general Presburger formulas, those that arise in our problem domain are relatively inexpensive to solve – since they typically possess a small number of constraints, and do not contain multiple levels of alternating quantifiers. We have found that the Omega library [19, 21] can easily handle these types of constraints, when purely integer-valued expressions are involved.

3.2 Temporal Properties

We use four CTL-style modal operators as the basis for our temporal logic – the "quantified-next-state" operators ($\exists\bigcirc$ and $\forall\bigcirc$), and "quantified-eventuality" operators ($\exists\Diamond$ and $\forall\Diamond$). Thus, the logic we use to reason about a program is generated over the set

$$\{f \in SF,\ \exists\bigcirc,\ \forall\bigcirc,\ \exists\Diamond,\ \forall\Diamond,\ \wedge,\ \vee,\ \neg\}.$$

As usual, quantified-invariant operators can easily be represented as $\exists\Box f = \neg\forall\Diamond\neg f$, and $\forall\Box f = \neg\exists\Diamond\neg f$, respectively.

The semantics of a temporal formula is defined on the paths of a program's transition system, $M = (S, I, X, L)$. A path $(s_0, s_1, s_2, \ldots)$ is a (finite or infinite) sequence of states, such that for each successive pair of states $(s_i, s_{i+1}) \in X$. Unlike Clarke *et al.* [14], we do not require the transition relation X to be total. Rather, the semantics is defined using maximal paths [6] (as opposed to infinite paths). A maximal path is one which is either infinite, or it ends with a state that has no successors. The semantics of the temporal operators can then be defined on a program's transition system $M = (S, I, X, L)$, as shown in Table 1.

If all the initial states of a program satisfy a temporal property, then we say that the program itself satisfies the property. Formally, given a temporal formula f and transition system $M = (S, I, X, L)$, $M \models f$ if and only if $\forall s \in I[s \models f]$.

Using our temporal logic, we can specify the property of the safety injection system mentioned above as follows:

$$\forall\Box(TLow \rightarrow \quad (wp1, wp2 < low \vee wp1, wp3 < low \\ \vee\ wp2, wp3 < low)).$$

Some other properties of the system are:

$$\forall\Box((Reset \wedge \neg High) \rightarrow \neg Over)$$
$$\forall\Box((Reset \wedge TLow) \rightarrow Inject)$$

$s \models f$	iff	$L(s, f) = true$, where $f \in SF$
$s \models \neg f$	iff	$s \not\models f$
$s \models f \wedge g$	iff	$s \models f$ and $s \models g$
$s \models f \vee g$	iff	$s \models f$ or $s \models g$
$s_0 \models \forall\bigcirc f$	iff	for all maximal paths $(s_0, s_1, s_2, \ldots)$, with length ≥ 2, $s_1 \models f$
$s_0 \models \exists\bigcirc f$	iff	for some maximal path $(s_0, s_1, s_2, \ldots)$, with length ≥ 2, $s_1 \models f$
$s_0 \models \forall\Diamond f$	iff	for all maximal paths $(s_0, s_1, s_2, \ldots)$, there exists an i, $s_i \models f$
$s_0 \models \exists\Diamond f$	iff	for some maximal path $(s_0, s_1, s_2, \ldots)$, there exists an i, $s_i \models f$

Table 1: Semantics of Temporal Operators.

Symbolic Operations	
$F \wedge G$	conjunction of Presburger/BDD/Composite formulas
$F \vee G$	disjunction of Presburger/BDD/Composite formulas
$\neg F$	negation of Presburger/BDD/Composite formulas
F^{-1}	inverse of Presburger/BDD/Composite relation F
$F[G]$	restrict domain of Presburger/BDD/Composite relation F to Presburger/BDD/Composite formula G and return the range of the result

Figure 3: Symbolic Presburger/BDD/Composite Operations.

which basically state that the following are invariants of the system:

1: $(\texttt{Reset} = \texttt{On} \wedge \texttt{Pressure} \neq \texttt{High})$
$\quad\rightarrow \neg\ \texttt{Overridden}$
2: $(\texttt{Reset} = \texttt{On} \wedge \texttt{Pressure} = \texttt{TooLow})$
$\quad\rightarrow \texttt{Inject} = \texttt{On}$

4 Symbolic Representations

Composite formulas – and their corresponding set-theoretic interpretations – give us a convenient way to symbolically encode sets of program states. We can also use this encoding to represent the program's underlying transition relation. If we assume that all events are representable as composite formulas (which prevents us, for example, from defining multiplication within a single event), then X_e (the transition relation of event e) is representable as a composite formula. This results in $|E|$ formulas, which together symbolically encode the transition relation X.

Recall that composite state-sets and transition relations usually posses both Boolean and integer parts – which we encode separately, and whose set-theoretic and Boolean operations are carried out in a type-specific manner. Hence, we use specialized procedures to (1) decompose sets of states (and transitions); (2) process them by their respective libraries; and to (3) assemble the results, and give the right semantic interpretation to them.

In carrying out these operations, the composite-model library interacts with two underlying libraries, whose key exported operations are defined in Figure 3. We refer interested readers to [19] for relevant background on the Presburger solver, and to [9, 20] for details on how general BDD formulas are manipulated (however the BDD library we use is our own). The composite library exports the same functions for composite formulas, and these, in turn, are used by the model checker's algorithms.

4.1 Composite State Representations

Given a system $C = (V, I, E)$, we partition the set of variables V into two classes $V = V^I \cup V^B$, with $V^I \cap V^B = \emptyset$, where V^I is the set of integer variables and V^B is the set of Boolean variables. Constraints strictly over variables in V^I are encoded using Presburger formulas – whereas constraints formed exclusively over variables in V^B are encoded using BDDs. (Currently we treat ordered enumerated types as integers, and unordered enumerated types as Boolean vectors.)

Let the Boolean variables $V^B = \{v_1^B, \dots, v_m^B\}$, and the integer variables $V^I = \{v_1^I, \dots, v_n^I\}$. Consider the state $s \in S$, such that

$$s = (\bigwedge_{j=1}^{m} v_j^B = s_j^B) \bigwedge (\bigwedge_{j=1}^{n} v_j^I = s_j^I)$$

where all the s_j^B's are Boolean constants, and the s_j^I's are integer constants. We use the notation $s \restriction V^B$ to denote the restriction of s to the variables in V^B; similarly, $s \restriction V^I$ denotes the restriction of s to V^I:

$$s \restriction V^B = \bigwedge_{j=1}^{m} v_j^B = s_j^B \qquad s \restriction V^I = \bigwedge_{j=1}^{n} v_j^I = s_j^I$$

Note that $s \restriction V^B$ and $s \restriction V^I$ implicitly define sets of states – since the variables removed are now considered "don't cares."

For example, two states for the safety injection system from Figure 2 are:

$$\begin{aligned} s_1 &= wp1 = 950 \ \wedge \ wp2 = 930 \ \wedge wp3 = 890 \ \wedge \\ &\quad \neg Block \ \wedge \ \neg Reset \ \wedge \ \neg Inject \ \wedge \ \neg Damp \ \wedge \\ &\quad \neg Over \wedge \neg TLow \ \wedge \ Low \ \wedge \ \neg High \ \wedge \ \neg THigh \\ s_2 &= wp1 = 910 \ \wedge \ wp2 = 850 \ \wedge \ wp3 = 930 \ \wedge \\ &\quad Block \ \wedge \ \neg Reset \ \wedge \ \neg Inject \ \wedge \ \neg Damp \ \wedge \\ &\quad \neg Over \ \wedge \ \neg TLow \ \wedge \ Low \ \wedge \ \neg High \ \wedge \ \neg THigh \end{aligned}$$

where as usual, $\neg v$ denotes $v = false$, and v means $v = true$. We partition the set of variables for the safety injection system as follows:

$$\begin{aligned} V^I &= \{wp1, wp2, wp3\} \\ V^B &= \{Block, Reset, Inject, Damp, Over, TLow, \\ &\quad Low, High, THigh\} \end{aligned}$$

where two example restricted states are:

$$\begin{aligned} s_1 \restriction V^I &= s_1 \restriction \{wp1, wp2, wp3\} \\ &= wp1 = 950 \ \wedge \ wp2 = 930 \ \wedge \ wp3 = 890 \\ s_2 \restriction V^B &= Block \ \wedge \ \neg Reset \ \wedge \ \neg Inject \ \wedge \ \neg Damp \ \wedge \\ &\quad \neg Over \ \wedge \ \neg TLow \ \wedge \ Low \ \wedge \ \neg High \ \wedge \\ &\quad \neg THigh. \end{aligned}$$

We extend variable restriction over (symbolic) sets of states and relations as follows. Let Q be a set of states in S, and let R be any relation over $S \times S$. Then

$$Q \restriction V^B \stackrel{\text{def}}{=} \bigvee_{s \in Q} s \restriction V^B$$

$$R \restriction V^B \stackrel{\text{def}}{=} \bigvee_{(s,s') \in R} (s \restriction V^B \wedge s' \restriction V^B)$$

The integer restrictions, $Q \restriction V^I$ and $R \restriction V^I$, are defined similarly.

We use restriction to manipulate composite formulas with their suitable type-specific functions, as defined in Figure 3. To accomplish this, we convert all composite state formulas $Q \subseteq S$ to a type-specific disjunctive form, as follows:

$$Q = \bigvee_{i=1}^{n_Q} (q_i^I \wedge q_i^B)$$

where n_Q denotes the number of disjuncts needed. By definition, we have

$$\begin{array}{rclrcl} q_i^I \restriction V^I &=& q_i^I & q_i^B \restriction V^B &=& q_i^B \\ q_i^I \restriction V^B &=& true & q_i^B \restriction V^I &=& true \end{array}$$
$$q_i^I \wedge q_i^B = (q_i^I \restriction V^I) \wedge (q_i^B \restriction V^B)$$

Hence, we get

$$\begin{array}{rcl} s \in q_i^I & \Longleftrightarrow & s \restriction V^I \in q_i^I \\ s \in q_i^B & \Longleftrightarrow & s \restriction V^B \in q_i^B \end{array} \qquad (1)$$

These properties are satisfied by having the q_i^B's formed exclusively over Boolean variables and logical connectives, and the q_i^I's containing integer variables and constants, arithmetic operators and inequalities – as well as logical connectives.

Recall that our logic does *not* allow functions (or predicates) with *both* Boolean and integer arguments. Hence, such a disjunctive form can be obtained for any composite term, and in fact there may be many ways of decomposing a formula Q into the various disjuncts. We are currently investigating methods to find the most efficient representation.

4.2 Logical Operations on Composite Representations

Assume that we have two state sets P and Q represented symbolically as

$$P = \bigvee_{i=1}^{n_P} p_i^I \wedge p_i^B \quad \text{and} \quad Q = \bigvee_{i=1}^{n_Q} q_i^I \wedge q_i^B$$

where each p_i^B and q_i^B is represented in a BDD format, while each p_i^I and q_i^I is represented in Presburger form. Now we explain how to combine P and Q using logical connectives.

The simplest composite operation is disjunction:

$$P \vee Q = (\bigvee_{i=1}^{n_P} p_i^I \wedge p_i^B) \ \vee (\bigvee_{i=1}^{n_Q} q_i^I \wedge q_i^B)$$

Note that right hand side is in the symbolic form we want, so we do not have to do any processing, i.e., we just append the two disjunction representations, which is our composite symbolic form.

Conjunction is computationally more expensive:

$$P \wedge Q = \bigvee_{i=1,j=1}^{n_P,n_Q} (p_i^I \wedge q_j^I) \wedge (p_i^B \wedge q_j^B)$$

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, complement is the most expensive operation

$$\neg Q = \bigvee \bigwedge_{j=1}^{n_Q} \neg q_j \quad \text{where} \quad q_j = q_j^I \text{ or } q_j = q_j^B$$

Note that we can arrange the terms in the right hand side so that the result will be in our symbolic form. Complementation of a set Q with n_Q disjuncts may, in fact, create a set with 2^{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). We note, however, that each of the constituent datatype libraries – both for BDDs and for Presburger arithmetic – are quite adept at simplifying constraints in their own formats. (We use several known algorithms for reducing integer constraints, and for minimizing the complexity of BDD representations). 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

$$Q = (q_1^I \wedge q_1^B) \vee (q_2^I \wedge q_2^B)$$

we have the following properties:

$$\begin{array}{rcl} q_1^I = q_2^I & \rightarrow & Q = q_1^I \wedge (q_1^B \vee q_2^B) \\ q_1^B = q_2^B & \rightarrow & Q = (q_1^I \vee q_2^I) \wedge q_1^B \\ q_1^I \subseteq q_2^I \text{ and } q_1^I \subseteq q_2^I & \rightarrow & Q = q_2^I \wedge q_2^B \end{array}$$

Note that in all three 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 three conditions listed above, and (2) merge the appropriate disjuncts when a condition is satisfied.

4.3 Composite Transitions

The syntax of a program's events can be written in an arbitrary way – as long as it adheres to the rules of our composite logic. However when our pre-processor compiles a program, its events get decomposed into a disjunctive form – i.e., they are represented exactly like composite state representations.

In the sequel we assume that all events are *syntactically* represented in disjunctive form, since they are eventually compiled into that form. Hence, for every event e, we assume that we can write X_e as $X_e = X_e^I \cap X_e^B$, where X_e^I is representable as a Presburger formula on variables V^I, and where X_e^B is representable as a BDD formula on variables V^B. Then we can symbolically encode the transition relation X as:

$$X = \bigvee_{e \in E} X_e = \bigvee_{e \in E} (X_e^I \wedge X_e^B).$$

where, similar to composite state representations we have

$$\begin{array}{rclrcl} X_e^I \restriction V^I & = & X_e^I & X_e^B \restriction V^B & = & X_e^B \\ X_e^I \restriction V^B & = & true & X_e^B \restriction V^I & = & true \end{array}$$
$$X_e^I \wedge X_e^B = X_e^I \restriction V^I \wedge X_e^B \restriction V^B.$$

This means that similar to the property (1) for the state representations, for all states s and s', and transition relations X_e we have:

$$\begin{array}{rcl} (s\restriction V^I, s'\restriction V^I) \in X_e^I \restriction V^I & \Longleftrightarrow & (s, s') \in X_e^I \\ (s\restriction V^B, s'\restriction V^B) \in X_e^B \restriction V^B & \Longleftrightarrow & (s, s') \in X_e^B \end{array} \quad (2)$$

Now we state the fundamental property which enables us to manipulate integer and Boolean parts separately in our model checker:

$$\begin{array}{lll} (X_e^B \wedge X_e^I)[q^B \wedge q^I] & & \\ = \bigvee_{s^B \wedge s^I \in q^B \wedge q^I} (X_e^B \wedge X_e^I)[s^B \wedge s^I] & & (3) \\ = \bigvee_{s^B \wedge s^I \in q^B \wedge q^I} X_e^B[s^B \wedge s^I] \wedge X_e^I[s^B \wedge s^I] & & (4) \\ = \bigvee_{s^B \wedge s^I \in q^B \wedge q^I} X_e^B[s^B] \wedge X_e^I[s^I] & & (5) \\ = \bigvee_{s^B \in q^B \wedge s^I \in q^I} X_e^B[s^B] \wedge X_e^I[s^I] & & (6) \\ = \bigvee_{s^B \in q^B} (\bigvee_{s^I \in q^I} X_e^B[s^B] \wedge X_e^I[s^I]) & & (7) \\ = (\bigvee_{s^B \in q^B} X_e^B[s^B]) \wedge (\bigvee_{s^I \in q^I} X_e^I[s^I]) & & (8) \\ = X_e^B[q^B] \wedge X_e^I[q^I] & & (9) \end{array}$$

Step (3) follows from the definition of relational function application. Step (4) holds because of the fact that $s^B \wedge s^I$ is a single state; hence this is just conjuncting two functions on a single element. Steps (5) and (6) follow from properties (2) and (1), respectively. Step (8) basically pushes through the existential quantification for both the Boolean and the integer parts. This is sound since we know that X_e^B is a formula constructed only on Boolean variables, and the analogous property holds for X_e^I.

This property basically states that the image computation for the integer and Boolean parts are orthogonal, and hence they can be computed separately.

$$
\begin{array}{ll}
f \in SF & : \textsc{Return}(f) \\
f = \neg f_1 & : \textsc{Return}(\neg f_1) \\
f = f_1 \wedge f_2 & : \textsc{Return}(f_1 \wedge f_2) \\
f = f_1 \vee f_2 & : \textsc{Return}(f_1 \vee f_2) \\
f = \exists \bigcirc f_1 & : \textsc{Return}(\mathbf{pre}(f_1)) \\
f = \forall \bigcirc f_1 & : \textsc{Return}(\neg \mathbf{pre}(\neg f_1)) \\
f = \exists \Diamond f_1 & : Q_0 = f_1 \\
& \quad Q_{i+1} = Q_i \vee \mathbf{pre}(Q_i) \\
& \quad \textsc{Return}(Q_n) \text{ when } Q_n = Q_{n+1} \\
f = \forall \Diamond f_1 & : Q_0 = f_1 \\
\quad Q_{i+1} = & \quad Q_i \vee (\mathbf{pre}(Q_i) \wedge (\neg \mathbf{pre}(\neg Q_i))) \\
& \quad \textsc{Return}(Q_n) \text{ when } Q_n = Q_{n+1}
\end{array}
$$

Figure 4: Composite Model Checker.

5 Composite Model Checker

To symbolically compute the temporal operators, we define a function $\mathbf{pre} : 2^S \to 2^S$, called the *precondition function*, which, given a set of states, returns all the states that can reach this set in one step (i.e. after execution of a single event):

$$\mathbf{pre}(Q) \stackrel{\text{def}}{=} \{s : \exists s'[s' \in Q \wedge (s, s') \in X]\}.$$

Using the symbolic operations in Figure 3 we have $\mathbf{pre}(Q) = X^{-1}[Q]$, assuming that we have a symbolic representation for the overall transition relation X. Moreover, we can symbolically compute **pre** with respect to our event decomposition and the symbolic representation of $Q = \bigvee_{i=1}^{n_Q} q_i^I \wedge q_i^B$, as follows. These inferences make use of the results developed in the previous section.

$$
\begin{aligned}
\mathbf{pre}(Q) &= \bigvee_{e \in E} X_e^{-1}[Q] = \bigvee_{e \in E} X_e^{-1}[\bigvee_{i=1}^{n_Q} q_i^I \wedge q_i^B] \\
&= \bigvee_{e \in E} \bigvee_{i=1}^{n_Q} X_e^{-1}[q_i^I \wedge q_i^B] \\
&= \bigvee_{e \in E} \bigvee_{i=1}^{n_Q} ((X_e^I)^{-1} \wedge (X_e^B)^{-1})[q_i^I \wedge q_i^B] \\
&= \bigvee_{e \in E} \bigvee_{i=1}^{n_Q} (X_e^I)^{-1}[q_i^I] \wedge (X_e^B)^{-1}[q_i^B]
\end{aligned}
$$

Now, using the function **pre** and symbolic operations $\wedge$, $\vee$ and $\neg$, we construct a model checking procedure for our temporal logic, as shown in Figure 4. Given a program and a temporal logic formula, the model checker will (attempt to) symbolically compute the set of program states that satisfy the input formula – and the procedure will yield an exact answer if it converges. Note that this procedure is a partial-function, i.e., it is not guaranteed to terminate. The convergence depends on the structure of the program and the formula – which was fortunately the case for our SCR requirements specification.

However, we have also developed some conservative techniques which often work when exact results are unobtainable. For lack of space we do not present them here; instead the reader is referred to [11].

6 Experimental Results

Two properties of the safety injection system verified in [7] are:

$$
\begin{array}{ll}
(P1) & \forall \Box((Reset \wedge \neg High) \to \neg Over) \\
(P2) & \forall \Box((Reset \wedge TLow) \to Inject)
\end{array}
$$

We verified these properties on the system model presented in Figures 1 and 2. (P1) and (P2) 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 [7]; in fact, it models the three-way voting scheme as originally specified in [16]. 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 2) is unbounded in most dimensions, since the limit constants $\{min, low, high, toohigh, max\}$ remain unspecified. Hence, any property we check is proved for *any* concrete values, provided they satisfy the ordering $min < low < high < toohigh < max$. 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 readings. This can be shown by checking the following four properties:

$$
\begin{array}{ll}
(P3) & \forall \Box(CTLow \iff TLow) \\
(P4) & \forall \Box(CLow \iff Low) \\
(P5) & \forall \Box(CHigh \iff High) \\
(P6) & \forall \Box(CTHigh \iff THigh)
\end{array}
$$

where the voting technique introduces the following constraints:

$$
\begin{array}{lll}
CTLow & \stackrel{\text{def}}{=} & wp1, wp2 < low \ \vee wp1, wp3 < low \ \vee \\
& & wp2, wp3 < low \\
CLow & \stackrel{\text{def}}{=} & low \le wp1, wp2 < high \ \vee \\
& & low \le wp1, wp3 < high \ \vee \\
& & low \le wp2, wp3 < high \\
CHigh & \stackrel{\text{def}}{=} & high \le wp1, wp2 < toohigh \ \vee \\
& & high \le wp1, wp3 < toohigh \ \vee \\
& & high \le wp2, wp3 < toohigh \\
CTHigh & \stackrel{\text{def}}{=} & toohigh \le wp1, wp2 \ \vee toohigh \le wp1, wp3 \ \vee \\
& & toohigh \le wp2, wp3
\end{array}
$$

The model checker verified these (P3)-(P6) in 16.22, 35.54, 35.39 and 15.94 seconds, respectively. Some other properties we tried were

$$
\begin{array}{ll}
(P7) & \forall \Box(Inject \to TLow) \\
(P8) & \forall \Box(Damp \to THigh)
\end{array}
$$

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

7 Conclusions

We presented a composite model checker which combines the relative strengths of two different symbolic representations: BDDs and Presburger formulas. We applied this technique to a non-trivial SCR requirements specification, which contains 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 this same SCR specification – when all the booleans were mapped to integer variables. On the other hand, it would be impossible to validate our system model in a finite-state model checker, without either bounding the pressure-reading domains, or using some other abstraction.

The SCR example we selected was an extended version of those reported in [7, 16] and [18], and we were able to check various properties of the specification. We verified several of system's intrinsic invariants (which had been previously checked on abstractions of the model); and we verified several new properties as well, having to do with additional features we added. However, we believe what distinguishes our results is not necessarily the properties themselves – but rather, that they were proved for unbounded integer variables, over an unbounded state space. In other words, the invariants were proved as *theorems* intrinsic to the basic SCR model itself, without being constrained by abstractions. This is unlike most previous efforts involving model checking of requirements specifications.

We are extending this work in various directions. First, we plan to include other symbolic representations as well, including real variables, queues, and the like. We believe the decomposition methods we described here will generalize to these other datatypes.

Also, note that the model checker presented is a semi-decision procedure, and is not guaranteed to converge. However, for systems ranging over pure integer domains, we have already developed techniques which make automatic conservative approximations, and are guaranteed to converge (but may report false negatives). We are now applying these techniques to the composite models as well.

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

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[2] 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.

[3] 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.

[4] J. M. Atlee, and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA '96)* pages 280-292.

[5] 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.

[6] A. Arnold. Finite transition systems: semantics of communicating Systems. New Jersey, 1994, Prentice Hall.

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

[8] 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 (CAV '96).*

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

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

[11] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, LNCS 1254, pages 400–411.

[12] 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 *Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[13] 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 *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, LNCS 1254, pages 316–327.

[14] 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.

[15] E. Clarke, 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.

[16] 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.

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

[18] 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.

[19] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman and D. Wonnacott. The Omega Library (version 1.00) interface guide. Available at <http://www.cs.umd.edu/projects/omega>.

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

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

[22] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.