Composite Model Checking with Type Specific Symbolic Encodings*

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

Abstract

We present a new symbolic model checking technique, which analyzes temporal properties in multi-typed transition systems. Specifically, the method uses multiple type-specific data encodings to represent system states, and it carries out fixpoint computations via the corresponding type-specific symbolic operations. In essence, different symbolic encodings are unified into one composite model checker. Any type-specific language can be included in this framework – provided that the language is closed under Boolean connectives, propositions can be checked for satisfiability, and relational images can be computed. Our technique relies on conjunctive partitioning of transition relations of atomic events based on variable types involved, which allows independent computation of one-step pre- and post-conditions for each variable type.

In this paper we demonstrate the effectiveness of our method on a nontrivial data-transfer protocol, which contains a mixture of integer and Boolean-valued variables. The protocol operates over an unreliable channel that can lose, duplicate or reorder messages. Moreover, the protocol's send and receive window sizes are not specified in advance; rather, they are represented as symbolic constants. The resulting system was automatically verified using our composite model checking approach, in concert with a conservative approximation technique.

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

1 Introduction

Symbolic model checking has been quite successful in hardware verification [6, 7, 14]. This success is generally due to efficiency of encodings like BDDs, which can represent huge sets of bit-vector states in a highly compact format [3]. However, one shortcoming of the BDD representation is its inability to handle unbounded variables (like integers).

Alternatively, we recently proposed a model checker for general integer based systems, which uses Presburger constraints as its underlying state representation [5]. 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.

While our Presburger technique easily verified some interesting integer-valued problems, it was inefficient 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.

In this paper, we present a general framework for combining multiple type-specific encodings, which allows us to exploit the strengths of both BDDs and Presburger formulas in analyzing systems with a variety of variable types. Also, our strategy can easily include other symbolic encodings as well – provided that the language is closed under Boolean connectives, propositions can be checked for satisfiability, and relational images can be computed. In [13], such an encoding is called an *adequate language*.

The concepts in this paper rest on a large body of work done by us and others. Obviously, we rely heavily on the previous work on symbolic model-checking [6, 7, 14]. As for the definition of an adequate language – and the requisite operations contained in one – the concept was nicely reported by [13] at CAV '97. At the same conference, we described our prototype Presburger-based model-checker [5], and showed how it could be used in concert with a multi-polytope widening operator. Recently, we reported some preliminary progress with exact BDD/Presburger model-checking, where the objective was verifying requirements of a sensor-control system [4].

In this paper we extend these results, by (1) presenting a model checker that accepts *any* adequate language; (2) showing how it works on a TCP-like data-transfer protocol; and (3) integrating conservative approximation techniques with composite model-checking – where the goal is to accelerate fixpoint computations over infinite variable domains. Specifically, we show how we use type-specific approximation operators (such as widening) over multiple-types.

The example protocol is similar to TCP; however, abstractions and simplifications have been made. Yet, much of TCP's complexity remains: the protocol is assumed to operate over an unreliable channel, which can lose, duplicate or reorder messages (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 paper, we show how the system was automatically verified using our composite model checking approach, in concert with a conservative approximation technique.

In [11], a different version of a sliding-window protocol 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. But essentially, while examining a similar problem, [11] and this paper illustrate two different (but related) techniques. In [11], 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.

In recent years various symbolic encodings have been proposed which are efficient for certain variable types. For example model checkers for hybrid systems encode real variables using affine constraints [1]. Affine constraint encoding of real variables is similar to the way we encode integer variables using Presburger formulas.

In [13], Kesten *et al.* use rich assertional languages that can encode infinite-state systems for symbolic model checking. In particular, they use regular sets and tree regular sets as symbolic encodings.

Queue content Decision Diagrams (QDDs) are proposed to encode sets of queue-configurations [2], where the prime use is to carry out reachability queries on communicating state machines. Queue configurations are modeled via deterministic finite-state automata structures, where the language accepted by the automata is equal to the set of queue-contents. QBDDs extend QDDs, by combining QDD representation with BDD encoding [10]. QBDDs have limited expressiveness for infinite sets, they are more appropriate for encoding bounded queues.

Several symbolic representations have been proposed for modeling functions over Boolean variables with non-Boolean ranges, including Multi-Terminal Binary Decision Diagrams (MTBDDs), Binary Moment Diagrams (BMDs), and their generalization Hybrid Decision Diagrams (HDDs) [9]. These are especially useful for datapath circuit verification, since they can encode functions that map Boolean vectors to integers.

Each of these representations can provide an efficient encoding for a specific variable type. However, we believe our techniques for combining different symbolic encodings could, with little effort, be extended to all abovementioned representations – with the result being a general-purpose model checker.

2 Composite Model

We represent a concurrent system $C = (V, I, E)$ by (1) a finite set of 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. A system state is determined by the values of its variables. Each event defines a transformation on the variables of the program.

Given a system $C = (V, I, E)$, we model it as an infinite transition system $M = (S, I, X, VF)$, 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 $VF : S \times SF \to \{\mathbf{true}, \mathbf{false}\}$ 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. The global transition relation is $X \equiv \bigvee_{e \in E} X_e$. 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.

2.1 Symbolic Representations

Given a concurrent system $C = (V, I, E)$, each variable $v \in V$ has a type $t_v \in T$ where T denotes the set of variable types. We assume that for each type $t \in T$, there is a symbolic assertional language L^t, which encodes formulas over t-typed variables and constants. For the sake of carrying out model checking, we assume that any type language L^t has the following properties:

- L^t is effectively closed under the Boolean connectives negation, conjunction and disjunction.
- Satisfiability is decidable for assertions over L^t, and we have an algorithm to carry out the procedure.
- There is also an algorithm to compute binary relational images over L^t. That is, given R of type $L^t \to L^t$, and $Q \in L^t$, we can compute $R[Q]$, which is defined by restricting the domain of R to set Q,

and returning the range of the result.

This definition was borrowed (with some modifications) from the concept of an *adequate language* described in [13]. We use the term "adequate language" to describe any type signature (with a corresponding implementation) which adheres to these three properties. Specifically we have implemented a model checker using two adequate languages – Presburger arithmetic (implemented with the Omega Library [12]), and BDDs (implemented in our own class library).

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:

$$f ::= (f) \mid f \ \vee \ f \mid f \ \wedge \ f \mid \neg f \mid f^t$$

where $t \in T$ and $f^t \in L^t$.

We use this composite language to represent the state formulas SF, i.e., $f \in SF$ only if $f \in L^c$. We also assume that the set of initial states I can be represented as a composite formula, i.e., $I \in L^c$.

As with single-type symbolic model checking, formulas in L^c can symbolically encode composite 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.

2.2 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:

$$P \vee Q \;\equiv\; (\bigvee_{i=1}^{n_P} \bigwedge_{t \in T} p_i^t) \;\vee\; (\bigvee_{i=1}^{n_Q} \bigwedge_{t \in T} q_i^t) \qquad\qquad P \wedge Q \;\equiv\; \bigvee_{i=1,j=1}^{n_P,n_Q} \bigwedge_{t \in T} p_i^t \wedge q_j^t$$

$$\neg Q \;\equiv\; \bigvee \bigwedge_{j=1}^{n_Q} \neg q_j \quad \text{where} \quad q_j = q_j^t \;\; t \in T$$

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, most type-specific symbolic libraries contain their 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 use.) 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:

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

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.

2.3 Composite Transitions

Given a system $C = (V, I, E)$ we assume that every atomic event $e \in E$ can be conjunctively partitioned based on the variable types. Any system which does not allow type-casting or multi-typed functions would satisfy this requirement. Then, for an event e, we can represent its transition relation X_e as $X_e \equiv \bigwedge_{t\in T} X_e^t$, where each $X_e^t \in L^t$. Hence, we can symbolically encode the entire transition relation X as:

$$X \equiv \bigvee_{e\in E} X_e \equiv \bigvee_{e\in E} \bigwedge_{t\in T} X_e^t$$

Now we state the fundamental property which enables us to manipulate the type specific symbolic encodings independently:

$$(\bigwedge_{t\in T} X_e^t)[\bigwedge_{t\in T} q_i^t] \equiv \bigwedge_{t\in T} X_e^t[\bigwedge_{t\in T} q_i^t] \equiv \bigwedge_{t\in T} X_e^t[q_i^t]$$

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

3 Composite Model Checker

Now we define a *precondition operator*, **pre** : $2^S \rightarrow 2^S$, 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):

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

By definition, we have that $\mathbf{pre}(Q) \equiv X^{-1}[Q]$. More practically, however, we can symbolically compute **pre** with respect to the system's event decomposition, and the symbolic representation of $Q \equiv \bigvee_{i=1}^{n_Q} \bigwedge_{t\in T} q_i^t$:

$$\mathbf{pre}(Q) \equiv \bigvee_{e\in E} X_e^{-1}[Q] \equiv \bigvee_{e\in E} (\bigwedge_{t\in T} (X_e^t)^{-1})[\bigvee_{i=1}^{n_Q} \bigwedge_{t\in T} q_i^t] \equiv \bigvee_{e\in E} \bigvee_{i=1}^{n_Q} \bigwedge_{t\in T} (X_e^t)^{-1}[q_i^t]$$

$f = \exists \bigcirc f_1 : \textsc{Return}(\mathbf{pre}(f_1))$	$f = \forall \bigcirc f_1 : \textsc{Return}(\neg\mathbf{pre}(\neg f_1))$
$f = \exists\Diamond f_1 \quad : Q_0 = f_1$ $Q_{i+1} \equiv Q_i \vee \mathbf{pre}(Q_i)$ $\textsc{Return}(Q_n)$ when $Q_n \equiv Q_{n+1}$	$f = \forall\Diamond f_1 \quad : Q_0 \equiv f_1$ $Q_{i+1} \equiv Q_i \vee (\mathbf{pre}(Q_i) \wedge (\neg\mathbf{pre}(\neg Q_i)))$ $\textsc{Return}(Q_n)$ when $Q_n \equiv Q_{n+1}$

Figure 1: Computation of Temporal Properties.

which follows from results developed in the previous section.

Now, using the **pre**, as well as the symbolic operations $\wedge$, $\vee$ and $\neg$, we can construct computation procedures for basic CTL properties, as shown in Figure 1; the full composite model-checker follows from the Boolean operations described in the previous section. If all types are finite, the algorithm will always converge – after all, Figure 1 is basically the core part of any symbolic model-checker. However, if integers (or other unbounded types) are involved, this procedure may well be a partial-function, as it is with our prototype tool – which uses BDDs and Presburger arithmetic over unbounded transition systems. Hence, we often appeal to conservative techniques, which can help when exact results are unobtainable.

3.1 Approximate fixpoint computations

As reported in [5], if we cannot directly compute a property f for a program C, we generate a lower-bound for f, denoted f^-, such that $f^- \subseteq f$. Then we check if $I \subseteq f^-$; if so, we conclude that $I \subseteq f$. However, if $I \not\subseteq f^-$, we cannot conclude anything, because it may be a *false negative*. In that case, we 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*. If both cases fail, i.e., both $I \not\subseteq f^-$ and $I \cap (\neg f)^- \equiv \emptyset$, then the model checker can not report a definite answer.

To compute a lower bound to a property like $g \equiv \neg h$, we first need to compute an *upper* approximation h^+ for the subformula h, and then let $g^- \equiv S - h^+$. Thus, we need algorithms to compute both lower and upper bounds of temporal formulas.

Since all operators in our temporal logic other than "$\neg$" are monotonic, we can compute a lower/upper approximation for a negation free formula using the corresponding lower/upper approximations for its subformulas. As for handling arbitrary levels of negation, we can easily generalize the abovementioned method for outermost negation operators and determine what type of approximation we need for each subformula [5].

Note that each iteration of the exact fixpoint computations (Figure 1) will yield a lower a bound for $\exists\Diamond f_1$ and $\forall\Diamond f_1$. 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 [8], but over multiple convex regions, and 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 "∇^t" is a type-specific widening operator such that:

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

i.e., $q_1^t \nabla^t q_2^t$ is an upper bound for the union computation. (Note that we do not have the guaranteed convergence requirement given in [8], i.e., our approximate fixpoint computations are not guaranteed to converge.)

Obviously there are many choices for operators which majorize binary union. However, if ∇^t is to be useful, it should – in many settings – be able to "guess" the direction of growth in the fixpoint iterates,

Constants:

snd_wnd: positive integer // sender window
rcv_wnd: positive integer // receiver window

Variables:

$sender$: $\{\mathtt{closed}, \mathtt{syn\_sent}, \mathtt{established}, \mathtt{fin\_wait}\}$ // sender state
$receiver$: $\{\mathtt{listen}, \mathtt{established}\}$ // 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:

$snd\_una = snd\_nxt = rcv\_nxt = 0 \ \wedge \ sender = \mathtt{closed} \ \wedge \ receiver = \mathtt{listen}$
$\wedge \ syn = ack = snt = finS = finR = \textbf{false}$

Events:

// SENDER EVENTS

event e_{S1} : // sender goes from closed to syn_sent
$sender = \mathtt{closed} \ \wedge \ sender' = \mathtt{syn\_sent} \ \wedge \ syn' = \textbf{true} \ \wedge \ finS' = \textbf{false}$

event e_{S2} **:** // sender receives ack and goes from syn_sent to established
$ack = \textbf{true} \ \wedge \ sender = \mathtt{syn\_sent} \ \wedge \ sender' = \mathtt{established} \wedge \ syn' = \textbf{false}$
$\wedge \ snd\_una' = snd\_nxt' = 0$

event e_{S3} **:** // sender sends data in established
$sender = \mathtt{established} \ \wedge \ snt' = \textbf{true}$
$\wedge \ (\exists seg\_len \ : \ 0 < seg\_len \ \wedge \ snd\_nxt + seg\_len \leq snd\_una + snd\_wnd$
$\wedge \ snd\_nxt' = snd\_nxt + seg\_len) \ \vee \ snd\_nxt' = snd\_nxt$

event e_{S4} **:** // sender receives ack in established
$sender = \mathtt{established}$
$\wedge \ (\exists seg\_ack : 0 \leq seg\_ack \ \wedge \ seg\_ack \leq rcv\_nxt \ \wedge \ snd\_una < seg\_ack \leq snd\_nxt$
$\wedge \ snd\_una' = seg\_ack) \ \vee \ snd\_una' = snd\_una$

event e_{S5} **:** // sender goes from established to fin_wait
$sender = \mathtt{established} \wedge \ sender' = \mathtt{fin\_wait} \ \wedge \ finS' = \textbf{true}$

event e_{S6} **:** // sender goes from fin_wait to closed after receiving fin
$finR = \textbf{true} \ \wedge \ sender = \mathtt{fin\_wait} \ \wedge \ sender' = \mathtt{closed} \ \wedge \ syn' = finS' = snt' = \textbf{false}$

// RECEIVER EVENTS

event e_{R1} **:** // receiver goes from listen to established after receiving syn
$syn = \textbf{true} \ \wedge \ receiver = \mathtt{listen} \ \wedge \ receiver' = \mathtt{established} \ \wedge \ ack' = \textbf{true} \ \wedge \ finR' = \textbf{false}$
$\wedge \ rcv\_nxt' = 0$

event e_{R2} **:** // receiver receives data in established
$snt = \textbf{true} \ \wedge \ receiver = \mathtt{established} \wedge$
$(\exists seg\_seq, seg\_len \ : \ 0 \leq seg\_seq \ \wedge \ 0 < seg\_len \ \wedge \ rcv\_wnd > 0 \ \wedge \ (seg\_seq + seg\_len \leq snd\_nxt)$
$\wedge \ (rcv\_nxt \leq seg\_seq < rcv\_nxt + rcv\_wnd \ \vee \ rcv\_nxt \leq seg\_seq + seg\_len - 1 < rcv\_nxt + rcv\_wnd)$
$\wedge \ ((seg\_seq + seg\_len > rcv\_nxt + rcv\_wnd \ \wedge \ rcv\_nxt' = rcv\_nxt + rcv\_wnd)$
$\vee \ (seg\_seq + seg\_len \leq rcv\_nxt + rcv\_wnd \ \wedge \ rcv\_nxt' = seg\_seq + seg\_len))) \ \vee \ rcv\_nxt' = rcv\_nxt$

event e_{R3} **:** // receiver goes from established to listen after receiving fin
$finS = \textbf{true} \ \wedge \ receiver = \mathtt{established} \wedge \ receiver' = \mathtt{listen} \ \wedge \ finR' = \textbf{true} \ \wedge \ ack' = \textbf{false}$

// IDLE EVENT

event e_I **:** // receiver and sender events can be delayed indefinitely
true

Figure 2: A Data Transfer Protocol.

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 [5] how this is done for integer domains, over multiple convex regions.

We have extended the widening idea to composite models. 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 P' denote the following subset of disjuncts forming P: For each $p_i \in P'$, there exists some q_j such that $p_i \subseteq q_j$. Likewise, let Q' denote the set of disjuncts in Q for which $q_j \in Q'$ means there is a p_i such that $p_i \subseteq q_j$. Then we define $P \bigtriangledown Q$ as

$$P \bigtriangledown Q \equiv \bigvee_{p_i \notin P'} p_i \vee \bigvee_{q_i \notin Q'} q_i \vee \bigvee_{p_i \in P', q_j \in Q'} \bigwedge_{t \in T} (p_i^t \bigtriangledown^t q_j^t)$$

Using this operator we can generate a majorizing sequence for our least fixpoint computations as follows:

$\hat{Q}_0$	$\equiv$	f	$\hat{Q}_0$	$\equiv$	f
$\hat{Q}_{i+1}$	$\equiv$	$\hat{Q}_i \bigtriangledown (\hat{Q}_i \vee \mathbf{pre}(\hat{Q}_i))$	$\hat{Q}_{i+1}$	$\equiv$	$\hat{Q}_i \bigtriangledown (\hat{Q}_i \vee (\mathbf{pre}(\hat{Q}_i) \wedge \neg\mathbf{pre}(\neg\hat{Q}_i)))$
$(\exists\Diamond f)^+$	$\equiv$	$\hat{Q}_n$ when $\hat{Q}_n \equiv \hat{Q}_{n+1}$	$(\forall\Diamond f)^+$	$\equiv$	$\hat{Q}_n$ when $\hat{Q}_n \equiv \hat{Q}_{n+1}$

4 A Data Transfer Protocol

In this section we demonstrate the effectiveness of our technique on a one-directional data transfer protocol, abstracted from the TCP specification (RFC 793). We analyze parts of the protocol, specifically: handshaking for call setup and take-down, as well as reliable data-transfer over an unreliable channel, which can delay, duplicate, lose or reorder messages to an unlimited extent. 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 approximation techniques based on their operators.

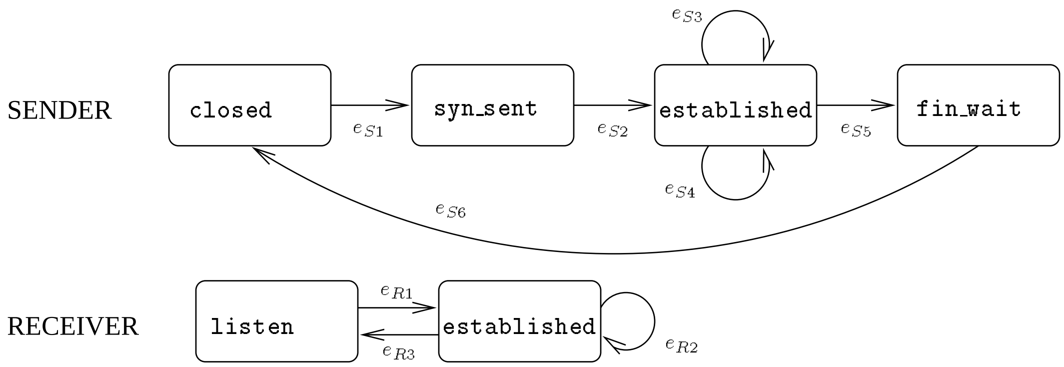

Figure 3: Sender and Receiver States.

Figure 2 gives the event-action language description of our protocol, which involves one sender process and one receiver process. (In Figure 2, next value of a variable v is denoted as v'; if v' does not appear in the action of an event then $v' = v$.) As in TCP call-establishment, 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. We show the event transitions in Figure 3 without their enabling conditions – which involve sequence numbers, link-state flags, etc.

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 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.

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., after an instantiation of the protocol, there are no previous *syn*'s hanging around in the network. (There are standard fixes for this used TCP, but for the sake of conciseness, we omit them.)

PROP.	FIXPOINT	CONVERGED IN
(P1)	Exact	4 iterations – 3.11 sec.
	Approximate	4 iterations – 3.42 sec.
(P2)	Exact	did not converge
	Approximate	6 iterations – 14.13 sec.
(P3)	Exact	1 iterations – 1.74 sec.
	Approximate	1 iterations – 1.93 sec.

PROP.	FIXPOINT	CONVERGED IN
(P4)	Exact	1 iterations – 1.73 sec.
	Approximate	1 iterations – 1.71 sec.
(P5)	Exact	7 iterations – 91.97 sec.
	Approximate	4 iterations – 11.52 sec.
(P6)	Exact	did not converge
	Approximate	10 iterations – 377.44 sec.

Figure 4: Verified Properties.

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

- (P1) $\forall\Box(sender = \texttt{established} \rightarrow receiver = \texttt{established})$:

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

- (P2) $\forall\Box(receiver = \texttt{established} \rightarrow \neg(sender = \texttt{closed}))$: If the receiver establishes a connection, then the sender is not closed.

- (P3) $\forall\Box(sender = \texttt{established} \rightarrow snd\_una \leq snd\_nxt \leq snd\_una + snd\_wnd)$: The sender does not overload the send buffer.

- (P4) $\forall\Box((receiver = \texttt{established} \wedge rcv\_nxt = i) \rightarrow \forall \bigcirc (rcv\_nxt \leq i + rcv\_wnd))$: The receiver does not overload the receive buffer.

- (P5) $\forall\Box(sender = receiver = \texttt{established} \rightarrow snd\_una \leq rcv\_nxt)$: All the acknowledged messages are received by the receiver.

- (P6) $\forall\Box(sender = receiver = \texttt{established} \rightarrow rcv\_nxt \leq snd\_nxt)$: All messages received were actually sent by the sender.

Figure 4 shows the results of our experiments. Property (P1) 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 (P2) 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 (P2) using the approximate fixpoint computations.

On the other hand, properties (P3) and (P4) 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 (P5) 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 (P6) 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.

5 Conclusions

We presented a technique to combine symbolic type-specific languages in a single composite model checker, and to maintain their encodings in carrying out fixpoint computations. 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.

Our current model checker uses two class libraries for encoding type-specific constraints. While both share a similar API, one of them (a BDD implementation) is used exclusively for Boolean types, and the other is the Omega library, used for integer-valued variables and their constraints. Then a composite-model library 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, which imports the composite-model routines.

Using this model-checker, we were able to analyze the sliding-window semantics of a data-transfer protocol, in addition to its handshaking involved in connection-setup and tear-down. The properties were verified automatically (with conservative approximations in some cases). Moreover, they were proved for unbounded integer variables, over an unbounded state space, as *theorems* intrinsic to the underlying protocol – and not specific to arbitrary bounds for integers.

Our approach to mixed constraints – and their orthogonal implementations – will allow expanding to additional datatypes in the future. To this end, we plan to incorporate QDDs as well, to help model communication channels more realistically.

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] 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).*

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

[4] T. Bultan, R. Gerber, and C. League. "Verifying Systems with Integer Constraints and Boolean Predicates: A Composite Approach." To appear in *Proceedings of the 1998 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98).*

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] R. Kaivola. Using Compositional Preorders in the Verification of Sliding Window Protocol. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, LNCS 1254, pages 48–59.

[12] 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>.

[13] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, LNCS 1254, pages 424–435.

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