Using canonical representations of solutions to speed up infinite-state model checking

Tatiana Rybina and Andrei Voronkov

University of Manchester
{rybina,voronkov}@cs.man.ac.uk

Abstract. In this paper we discuss reachability analysis for infinite-state systems in which states can be represented by a vector of integers. We propose a new algorithm for verifying reachability properties based on canonical representations of solutions instead of decision procedures for integer or real arithmetic. Experimental results demonstrate that problems in protocol verification which are beyond the reach of other existing systems can be solved completely automatically.

1 Introduction

Reachability properties arise is many applications of verification. In this paper we discuss reachability analysis for infinite-state systems in which state can be represented by vectors of integers. We propose a new algorithm for verifying reachability properties based on canonical representations of solutions instead of decision procedures for integer or real arithmetic. Experiments carried out with our infinite-state model checker BRAIN demonstrate that hard problems in protocol verification which are beyond the reach of other existing systems can be solved completely automatically.

This paper is organized as follows. In Section 2 we introduce our model of transition systems over integers and their symbolic representation. We define the reachability problem and a special kind of transition systems used in BRAIN, called the guarded assignment systems. In Section 3 we introduce the so-called local backward reachability algorithm. In Section 4 we define the notion of basis of a constraint which is used in all main operations on constraints in BRAIN. In Section 5 we explain how, using bases, we implement all these operations, for example, entailment-checking for constraints. In Section 6 we give experimental evidence that our approach results in an efficient model-checker. In Section 7 we discuss related and future work.

The system BRAIN and all the examples of this paper can be found on the Web page `www.cs.man.ac.uk/~voronkov/BRAIN/`.

2 Preliminaries

We use a formal model of transition systems presented in [17]. In this model symbolic representations of transition system are formulas interpreted over a first-order structure, whose domain is the set of values for the state variables of the transition system. In this section we recall the main definitions of [17] specialized to the domain of integers and a particular first-order structure with this domain. We also define the reachability problem

and assignment-based systems used in our system BRAIN. We assume knowledge of standard model-theoretic definitions, such as first-order formulas, structure and truth, which can be found in any standard textbook on logic or model theory.

Denote by I the set of integers. We will formalize transition systems as follows. A transition system has a finite number of integer-valued variables. A state is a mapping from variables to integers. Transitions may change values of variables. A symbolic representation of such a system uses first-order formulas interpreted in a structure with the domain I.

We call an *integer transition system* a pair $\mathbb{S} = (\mathcal{V}, \mathcal{T})$, where $\mathcal{V}$ is a finite set of *state variables*. A *state* of the transition system $\mathbb{S}$ is a function $s : \mathcal{V} \to I$. We define the second component of transition systems as follows: $\mathcal{T}$ is a set of pairs of states, called the *transition relation* of $\mathbb{S}$. In the sequel we assume a fixed integer transition system $\mathbb{S} = (\mathcal{V}, \mathcal{T})$. We call a *transition* any set of pairs of states.

Consider the structure $\mathbb{I} = (I, >, <, \geq, \leq, +, -, 0, 1, 2, \ldots)$, where all the function and predicate symbols (for example $>$) have their standard interpretation over the integers.

A *valuation* for a set of variables V in $\mathbb{I}$ is any mapping $s : V \to I$. We will use the standard model-theoretic notation $\mathbb{I}, s \vDash A$ to denote that the formula A is true in the structure $\mathbb{I}$ under a valuation s. When we use this notation, we assume that s is defined on all free variables of A. A formula A with free variables V is said to be *satisfiable* (respectively, *valid*) in $\mathbb{I}$ if there exists a valuation s for V in $\mathbb{I}$ such that $\mathbb{I}, s \vDash A$.

A formula A is called *quantifier-free* if A contains no quantifiers. A formula A is called a *simple constraint* if A is a conjunction of atomic formulas.

We will often use the following simple property of $\mathbb{I}$.

LEMMA 1 *In $\mathbb{I}$ every quantifier-free formula A is equivalent to a disjunction of simple constraints.* ❑

In addition to the set of state variables $\mathcal{V}$, we also introduce a set $\mathcal{V}'$ of *next state variables* of the same cardinality as $\mathcal{V}$. We fix a bijection $' : \mathcal{V} \to \mathcal{V}'$ such that $v \in \mathcal{V}$ we have $v' \in \mathcal{V}'$. We can treat the variables in $\mathcal{V} \cup \mathcal{V}'$ also as logical variables. Then any mapping $s : \mathcal{V} \to I$ can be considered as both a state of the transition system $\mathbb{S}$ and a valuation for $\mathcal{V}$, and similarly for $s' : \mathcal{V}' \to I$.

Let S be a set of states and A be a formula with free variables in $\mathcal{V}$. We say that A *symbolically represents*, or simply *represents* S if for every valuation s for $\mathcal{V}$ we have $s \in S \Leftrightarrow \mathbb{I}, s \vDash A$. Likewise, we say that a formula B with free variables in $\mathcal{V} \cup \mathcal{V}'$ (symbolically) represents a transition T if for every pair of valuations (s, s') for $\mathcal{V}$ and $\mathcal{V}'$ respectively we have $(s, s') \in T \Leftrightarrow \mathbb{I}, s, s' \vDash B$.

In the sequel we will follow the following convention. We will often identify a symbolic representation of a transition with the transition itself. For example, when T is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$ we can refer to T as a transition.

Let us introduce an important special case of transition systems, called the *guarded assignment systems*. The current version of BRAIN works with quantifier-free guarded assignment systems. To define guarded assignment systems, let us first introduce a syntax sugar for representing transitions. We assume that the set of state variables of the transition system is $\mathcal{V}$.

$$
\begin{array}{rcl}
drinks > 0 \wedge customers > 0 \Rightarrow drinks & := & drinks - 1 \quad \texttt{(* dispense-drink *)} \\
true \Rightarrow drinks & := & drinks + 64 \quad \texttt{(* recharge *)} \\
true \Rightarrow customers & := & customers + 1 \quad \texttt{(* customer-coming *)} \\
customers > 0 \Rightarrow customers & := & customers - 1 \quad \texttt{(* customer-going *)}
\end{array}
$$

Fig. 1. Guarded assignment system

We call a *guarded assignment* any formula (or transition) of the form

$$P \wedge v'_1 = t_1 \wedge \ldots \wedge v'_n = t_n \wedge \bigwedge_{v \in \mathcal{V} - \{v_1, \ldots, v_n\}} v' = v,$$

where P is a formula with free variables $\mathcal{V}$, $\{v_1, \ldots, v_n\} \subseteq \mathcal{V}$, and $t_1, \ldots, t_n$ are terms with variables in $\mathcal{V}$. We will write guarded assignments as

$$P \;\Rightarrow\; v_1 \;:=\; t_1, \ldots, v_n \;:=\; t_n. \tag{1}$$

The formula P is called the *guard* of this guarded assignment.

A guarded assignment is *quantifier-free* if so is its guard. A guarded assignment is called *simple* if its guard is a simple constraint.

Formula (1) represents a transition which applies to the states satisfying P and changes the values of variables v_i to the values of the terms t_i before the transition. Note that a guarded assignment T is a deterministic transition: for every state s there exists at most one state s' such that $(s, s') \in T$. Moreover, such a state s' exists if and only if the guard of this guarded assignment is true at s, i.e., $\mathbb{I}, s \vDash P$.

A transition system is called a *guarded assignment system*, or simply *GAS* if its transition relation is a union of a finite number of guarded assignments. A GAS is called *quantifier-free* (respectively *simple*) if every guarded assignment in it is also quantifier-free (respectively simple). An example integer guarded assignment system for modelling a drink dispenser is given in Figure 1. This GAS is simple.

Note that a guarded assignment system may represent a non-deterministic transition system because several guards may be true in the same state.

THEOREM 2 *One can effectively transform every quantifier-free guarded assignment over $\mathbb{I}$ into an equivalent union of simple guarded assignments. Hence, Every integer quantifier-free GAS is also a simple GAS.*

The notion of a guarded assignment system is not very restrictive. Indeed, broadcast protocols [12] and Petri nets can be represented as integer simple guarded assignment systems. All transition systems for cache coherence protocols described in [10] are integer simple GAS.

We say that a state s_n is *reachable* from a state s_0 w.r.t. a transition $\mathcal{T}$ if there exists a sequence of states $s_1, \ldots, s_{n-1}$ such that for all $i \in \{0, \ldots, n-1\}$ we have $(s_i, s_{i+1}) \in \mathcal{T}$. In this case we also say that s_n is reachable from s_0 *in n steps* and that s_0 is *backward reachable* from s_n in n steps. When $\mathcal{T}$ is clear from the context we will simply write "reachable".

The reachability problem can now be defined as follows. We call the (integer) *reachability problem* the following decision problem. Given formulas In, Fin, and Tr such that (i) In represents a set of states, called the set of *initial states*; (ii) Fin represents a set of states, called the set of *final states*; and (iii) Tr represents the transition relation of a transition system $\mathbb{S}$, do there exist an initial state s_1 and a final state s_2 such that s_2 is reachable from s_1 w.r.t. Tr?

When we discuss instances of the reachability problem, we will call the formulas In and Fin the *initial* and *final conditions*, respectively, and Tr the *transition formula*.

The integer reachability problem for infinite-state systems is, in general, undecidable, even for simple GAS with three variables (see, e.g., [17]). Various results on reachability are discussed in many papers, including [12, 1, 15].

3 Local backward reachability algorithm

There are various (semi-decision) procedures for checking reachability. For example, a classification of reachability algorithms in undertaken in [17]. The current version of BRAIN uses an algorithm called the *local backward reachability algorithm*.

Before defining these algorithms, let us discuss symbolic representations of reachable states. We assume fixed initial and final conditions $In(\mathcal{V})$, $Fin(\mathcal{V})$ and the transition formula $Tr(\mathcal{V}, \mathcal{V}')$.

Let $A(\mathcal{V})$ be a formula which represents a set of states S. It is not hard to argue that the set of states reachable in one step from a state in S can be represented by the formula $\exists \mathcal{V}_1 (A(\mathcal{V}_1) \wedge Tr(\mathcal{V}_1, \mathcal{V}))$.

Likewise, the set of states backward reachable in one step from a state in S is represented by the formula $\exists \mathcal{V}_1 (A(\mathcal{V}_1) \wedge Tr(\mathcal{V}, \mathcal{V}_1))$. The last formula can be considerably simplified when the transition Tr is a guarded assignment. Let u be a guarded assignment of the form $P(v_1, \ldots, v_n) \;\Rightarrow\; v_1 := t_1, \ldots, v_n := t_n$. For simplicity we assume that $\mathcal{V} = \{v_1, \ldots, v_n\}$. This can be achieved by adding "dummy" assignments $v := v$ for every variable $v \in \mathcal{V} - \{v_1, \ldots, v_n\}$. Let also $A(v_1, \ldots, v_n)$ be a formula whose free variables are in $\mathcal{V}$. For every term t denote by t' the term obtained from t by replacing every occurrence of every state variable v_i by v'_i.

Define the following formulas:

$$A^u(v_1, \ldots, v_n) \stackrel{\text{def}}{=} \exists \mathcal{V}' (A(v'_1, \ldots, v'_n) \wedge P(v'_1, \ldots, v'_n) \wedge v_1 = t'_1 \wedge \ldots \wedge v_n = t'_n);$$
$$A^{-u}(v_1, \ldots, v_n) \stackrel{\text{def}}{=} P(v_1, \ldots, v_n) \wedge A(t_1, \ldots, t_n).$$

LEMMA 3 *Let $A(v_1, \ldots, v_n)$ represent a set of states S. Then (i) the formula $A^u(v_1, \ldots, v_n)$ represents the set of states reachable in one step from S using u; (ii) the formula $A^{-u}(v_1, \ldots, v_n)$ represents the set of states backward reachable in one step from S using u.* ❑

The *local backward reachability algorithm* LocalBackward is given in Figure 2. It is parametrized by a function select which selects a simple constraint in a set of simple constraints. The function $\mathsf{pdnf}(A)$ returns a set S of simple constraints such that A is equivalent to $\bigvee_{C \in S} C$ in $\mathbb{I}$. To apply this algorithm to a quantifier-free GAS, we first transform it to a simple GAS using Theorem 2.

procedure LocalBackward
input: quantifier-free formulas In, Fin,
finite set of simple guarded assignments U
output: "reachable" or "unreachable"
begin
IS := pdnf(In); FS := pdnf(Fin)
if there exist $I \in IS, F \in FS$ such that $\mathbb{I} \models \exists \mathcal{V}(I \wedge F)$ **then return** "reachable"
$unused$:= FS; $used$:= $\emptyset$
while $unused \neq \emptyset$
S := select$(unused)$
$used$:= $used \cup \{S\}$; $unused$:= $unused - \{S\}$
forall $u \in U$
(* backward application of u *)
N := S^{-u}
(* satisfiability-check for the new constraint N *)
if $\mathbb{I} \models \exists \mathcal{V}(N)$ **then**
(* intersection-checks *)
if there exists $I \in IS$ such that $\mathbb{I} \models \exists \mathcal{V}(N \wedge I)$ **then return** "reachable"
(* entailment-checks *)
if for all $C \in used \cup unused$ we have $\mathbb{I} \not\models \forall \mathcal{V}(N \rightarrow C)$ **then**
$unused = unused \cup \{N\}$
forall $C' \in used \cup unused$
(* more entailment-checks *)
if $\mathbb{I} \models \forall \mathcal{V}(C' \rightarrow N)$ **then** remove C' from $used$ or $unused$
return "unreachable"
end

Fig. 2. Local backward reachability algorithm used in BRAIN

THEOREM 4 (Soundness and Semi-Completeness) LocalBackward *has the following properties:*

1. *there is a final state reachable from an initial state if and only if the algorithm returns "reachable";*
2. *if the algorithm returns "unreachable", then there is no final state reachable from an initial state.* ❑

On some inputs the algorithms do not terminate. In this case we say that the reachability analysis *diverges*.

Note that termination of the local algorithms may depend on a the selection function select. Let us call the selection function *fair* if no formula remains in $unused$ forever.

THEOREM 5 *If the local forward (respectively backward) algorithm terminates for some selection function, then it terminates for every fair selection function.* ❑

To implement the LocalBackward one has to implement procedures for the following problems:

1. *Backward application of guarded assignments*: given a simple constraint S and guarded assignment u, compute S^{-u}.
2. *Satisfiability of simple constraints*: given a simple constraint C, is C or satisfiable in $\mathbb{I}$?
3. *Entailment of simple constraints:* given simple constraints N and C, is the formula $N \to C$ valid in $\mathbb{I}$?

To implement the reachability algorithm efficiently, one has to implement efficiently these three procedures. As our experimental data show, for hard problems the number of entailment-checks is considerably larger than the number transition applications and satisfiability-checks. Therefore, entailment-checking should be implemented especially efficiently.

A simple constraint over $\mathbb{I}$ can be considered as a system of linear equations and inequations over integers with variables in $\mathcal{V}$. Satisfiability of simple constraints is known to be NP-complete and entailment coNP-complete. If one uses relaxation (see [10] or [17]) to use real numbers instead of integers, then both problems can be solved in polynomial time.

4 Hilbert's basis

In this section we present some properties of the set of non-negative solutions to a simple constraint. Proofs can be found in, e.g., [18]. In this section we consider $\mathcal{V}$ as a vector of variables rather than a set and restrict ourselves by simple constraints with the variables $\mathcal{V}$. Denote by $\mathbf{0}$ a vector of 0's. Every simple constraint C with variables in $\mathcal{V}$ can be written as a system of linear equations and inequations with integer coefficients:

$$E\mathcal{V} + e = \mathbf{0}, \quad L\mathcal{V} + l \leq \mathbf{0}. \tag{2}$$

where E, L are matrices with integer coefficients and e, l are integer vectors. We call a *solution* to such a system any vector V of non-negative integers which satisfies all equations and inequations in the system. Let us emphasize that in this section we will only consider non-negative solutions. We will show below in Section 5 how to treat arbitrary integer solutions. For every system C of the form (2) denote by C^{hom} the corresponding system of homogeneous linear Diophantine equations

$$E\mathcal{V} = \mathbf{0}, \quad L\mathcal{V} = \mathbf{0}. \tag{3}$$

We call a solution v to (2) *non-decomposable* if it cannot be represented in the form $v_1 + v_2$, where v_1 is a solution to (2) and v_2 is a non-zero solution to (3). Likewise, we call a solution to (3) non-decomposable if and only if it cannot be represented as a sum of two non-zero solutions to (3). It is easy to see that for systems of homogeneous linear Diophantine equations a solution is non-decomposable if and only if it is minimal.

A pair of sets of vectors (N, H) is called a *basis* for a simple constraint C if the following conditions hold.

1. Every vector in N is a non-decomposable solution to C.
2. Every vector in H is a non-zero non-decomposable solution to C^{hom}.

3. Every solution v to C can be represented as a linear combination $v = w + \sum_{i=1\ldots k} m_i w_i$, where $w \in N$, $k \geq 0$ and for all $i = 1 \ldots k$ m_i is a non-negative integer and $w_i \in H$.
4. Every solution v to C^{hom} can be represented as a linear combination $v = \sum_{i=1\ldots k} m_i w_i$, where $k \geq 0$ and for all $i = 1 \ldots k$ m_i is a non-negative integer and $w_i \in H$. ❑

This definition is a modification of the standard definition of Hilbert's basis [14] for the case of systems of linear inequations.

THEOREM 6 *Every simple constraint has a basis, and this basis is unique.* ❑

Algorithms for finding the basis of systems of linear Diophantine equations and inequations are described in, e.g., [9, 2, 19]. BRAIN uses an original algorithm [20]. In general, it is more difficult to find the basis of a simple constraint than to check its solvability.[1] The solvability problem is NP-complete, but the number of vectors in the basis can be exponential in the size of the system. Nevertheless, we will show that the construction of the basis may speed up reachability-checking.

BRAIN uses an *incremental algorithm* for building the basis. We call an *incremental basis-finding function* any function ibff of two arguments, such that for every pair of simple constraints (C_1, C_2), if B is the basis for C_1, then $\mathsf{ibff}(B, C_2)$ is the basis of $C_1 \wedge C_2$. Essentially, an incremental basis-funding function uses a basis computed previously for C_1 to find a basis for $C_1 \wedge C_2$.

5 The BRAIN

In this section we explain how BRAIN implements the three important procedures used in the local backward reachability algorithm: backward application of guarded assignments, satisfiability and entailment. All three algorithms are implemented using repeated calls to an incremental basis-finding function ibff. In order to use the basis incrementally, BRAIN stores the basis together with every computed simple constraint. We assume that all variables range over non-negative integers and show how to handle arbitrary integers later. We call an *augmented constraint* a pair (C, B) consisting of a simple constraint C and its basis B.

Entailment-checking. The algorithm for entailment-checking in BRAIN is based on the following theorem.

THEOREM 7 *Let $(C_1, (N_1, H_1))$ be an augmented constraint and C_2 be a simple constraint. Then $\mathbb{I} \models \forall \mathcal{V}(C_1 \rightarrow C_2)$ if and only if the following two conditions hold: (i) every vector $v \in N_1$ is a solution to C_2; (ii) every vector $w \in H_1$ is a solution to C_2^{hom}.* ❑

[1] In fact, our experiments have shown that the existing algorithms for building the basis often outperform some well-known algorithms for checking solvability.

This theorem gives us an algorithm for entailment-checking: to check the entailment problem for augmented constraints, one has to check that the vectors of the basis of C_1 are solutions to C_2 or to the corresponding homogeneous system C_2^{hom}. Checking that a particular vector is a solution to a system can obviously be solved in time polynomial in the size of the vector and the system. As a consequence, we obtain the following theorem.

THEOREM 8 *Entailment of augmented constraints can be solved in polynomial time.*

The algorithm implicit in Theorem 7 is used in BRAIN to check the entailment.

Satisfiability-checking. Evidently, a simple constraint C is satisfiable if for its basis (N, H) we have $N \neq \emptyset$. So satisfiability-checking for augmented constraints is trivial. Note that the reachability algorithm makes two kinds of satisfiability-checks:

1. checking whether the new formula N (i.e., S^{-u}) is satisfiable;
2. *intersection-checks*, when we check satisfiability of the formula $N \wedge I$.

The latter kind of satisfiability-checking can be performed by any satisfiability-checking procedure. But the first kind of satisfiability checks in BRAIN is combined with the backward applications of transitions for the reasons mentioned below.

Backward application of transitions. Repeated backward applications of transitions in the reachability algorithm may create too large constraints. To explain this, let us consider the formula for computing the set of states backward reachable from the set states presented by a simple constraint $C_1(v_1, \ldots, v_n)$. If the guarded assignment u has the form $C_2 \;\Rightarrow\; v_1 \;:=\; t_1, \ldots, v_n \;:=\; t_n$, then by Lemma 3 the formula C_1^{-u} is $C_2 \wedge C_1(t_1, \ldots, t_n)$. The number of atomic formulas in this simple constraint is the number of atomic formulas in C_1 plus the number of atomic formulas in C_2. Every iteration of the reachability algorithm yields longer constraints in which, for hard examples described below in Section 6, the number of atoms may be over a hundred. It is often the case that a large number of these atoms are a consequence of the remaining atoms in the constraint and can be safely removed (in our hardest examples the number of non-redundant atoms usually does not exceed ten). Redundant atoms in constraints do not change the basis, but they slow down entailment, since our algorithm for checking validity of $(C_1 \rightarrow C_2)$ is, roughly speaking, linear in the number of atoms in C_2.

We can get rid of redundant constraints in $C_2 \wedge C_1(t_1, \ldots, t_n)$ together with checking satisfiability of this constraint and building a basis for it using an incremental basis-finding function. The procedure for this is given in Figure 3. The input to this procedure is the sequence of atoms in $C_2 \wedge C_1(t_1, \ldots, t_n)$ in any order. When a new atom A_i should be added to the constraint, it is first check if the addition of this atom changes the basis. If it does not, then the atom is redundant.

The current version of BRAIN only works with variables ranging over non-negative integers. Integers can be implemented using the same technology as follows. If an integer-valued variable is restricted by $v \leq n$ (or respectively by $n \leq v$), then it can be replaced by a variable $w = n - v$ (or respectively by $w = v - n$) ranging over non-negative integers. For every unrestricted integer-valued variable v one can introduce

procedure Basis
input: sequence of atoms $A_1, \ldots, A_n$,
output: pair (C, B), where C is equivalent to $A_1 \wedge \ldots \wedge A_n$,
and B is the basis for $A_1 \wedge \ldots \wedge A_n$
begin
C := $true$; B := the basis of C
for $i = 1 \ldots n$
$B' = \mathsf{ibff}(B, A_i)$
if B' contains no solution **then return** $(false, B')$
if $B' \neq B$ **then**(C, B) := $(C \wedge A_i, B')$
return (C, B)
end

Fig. 3. Incremental building of the basis

two variables w_1, w_2 ranging over non-negative integers and replace all occurrences of v by $w_1 - w_2$.

6 Experiments

In this section we present the results of experiments carried out on a number of benchmarks taken from several Web pages. The examples can be found on the Web page `www.cs.man.ac.uk/~voronkov/BRAIN/`. We compare the performance of BRAIN with that of the following systems: HyTech [13], Action Language Verifier [7], and DMC [11].

All benchmarks were carried out on the same computers (Sparc 300 with 1G of RAM memory). These computers are slow (about 8–10 times slower than the modern PCs), but we did not have access to a network of PCs with large RAM. The systems HyTech, Action Language, and BRAIN are implemented in C++ or C, and were compiled using the same version 2.92 of the GNU C/C++ compiler. DMC is implemented in Sicstus Prolog. In several cases we had to interrupt the systems because they consumed over 1G of memory. In these cases we show the time when they were interrupted and the memory consumed by that moment. DMC has never consumed more than 14M of memory, but was interrupted after several days of running. We were interested in hard benchmarks, but occasionally, for the sake of comparison, included figures for relatively easy benchmarks, because only HyTech and BRAIN could solve some of the hard ones. All runtimes are given in seconds.

Note that HyTech and DMC use relaxation, i.e., they solve real reachability problems instead of integer reachability problems. Therefore, they are correct only when they report non-reachability. Among the systems compared with BRAIN only Action Language checks for integer reachability.

We took most of the benchmarks presented in this paper were from Giorgio Delzanno's Web page `www.disi.unige.it/person/DelzannoG/`. These benchmarks were used to verify cache coherence protocols, properties of Java programs, and some other

problem	BRAIN		HyTech		speedup
	time	memory	time	memory	
incdec-inv	22.9s	7	120.9s	80	5.3
csm5	9.26s	4	40.32s	18	4.35
csm10	457s	28	868s	140	1.9
csm15	119m13s	126	154m32s	509	1.3
csm5-inv	0.37s	1	0.96s	2	2.6
csm10-inv	0.89s	1	0.96s	2	1.07
consprod-inv	1.19s	2	1.2s	4	1.01
consistencyprot	0.96s	1	17.66s	10	18.4
consistencyprot-inv	7.25s	2	0.13s	2	0.02
csm15-inv	1.88s	1	0.96s	2	0.51
bigjava-inv	40m2s	105	32m5s	849	0.80
bigjava	171m31s	245	>531m	>2345	>3.1
bigjava1	189m16s	304	>580m44s	>2480	>3.07
bigjava1-inv	744m14s	217	>271m50s	>2163	>0.36
incdec	170.3s	20	>470m	>1142	>165.6
consprod	173m31s	139	>560m	>1200	>3.22

Table 1. Comparison with HyTech

problem	BRAIN		DMC		speedup
	time	memory	time	memory	
incdec-inv	22.9s	7	96.3s	10	4.2
csm5	9.26s	4	13h15m	11	5165
csm5-inv	0.37s	1	3.2s	10	8.6
csm10-inv	0.89s	1	3.2s	10	3.59
csm15-inv	1.88s	1	3.2s	10	1.7
consprod-inv	1.19s	2	4.9	10	4.12
consistencyprot	0.96s	1	888s	10	925
consistencyprot-inv	7.25s	2	19m52s	11	164.4
bakery-4	0.6s	2	122s	10	203
csm10	457s	28	>150h	14	>1181
csm15	119m13s	126	>150h	14	>75.5
incdec	170.3s	20	>150h	14	>3176
consprod	173m31s	139	>150h	14	>51.9

Table 2. Comparison with DMC

problem	BRAIN		AL		speedup
	time	memory	time	memory	
incdec-inv	22.9s	7	10h20m	846	1625
csm5	9.26s	4	87m32s	297	567
csm5-inv	0.37s	1	2.87s	28	7.77
csm10-inv	0.89s	1	2.87s	28	3.22
csm15-inv	1.88s	1	2.87s	28	1.52
consistencyprot-inv	7.25s	2	164.8s	73	22.7
consprod-inv	1.19s	2	25.87s	68	21.7
bakery-4	0.6s	2	330.5s	294	550
consprod	173m31s	139	>295m	>1789	>1.7
csm10	457s	28	>666m	>1272	>87
csm15	119m13s	126	154m32s	>1525	>13
incdec	170.3s	20	>179m	>2269	>63
bigjava	171m31s	245	>118m	>2499	>0.69
bigjava-inv	40m2s	105	>610m	>1552	>15
bigjava1	189m16s	304	>217m	>2568	>1.15
bigjava1-inv	744m14s	217	>197m	>2234	>0.26

Table 3. Comparison with Action Language

problem	variables	time	memory	inter-section	entailment	transition applications
csm5	13	9.26s	4	3,313	790,119	3,576
csm10	13	457s	28	27,308	60,803,697	29,736
csm15	13	119m13s	126	107,503	990,874,884	117,496
csm5-inv	13	0.37s	1	106	835	152
csm10-inv	13	0.89s	1	106	835	152
csm15-inv	13	1.88s	1	106	835	152
consistencyprot	12	0.96s	1	813	30,557	880
consistencyprot-inv	12	7.25s	2	813	29,803	880
consprod	18	173m31s	139	162,817	698,478,060	181,650
consprod-inv	18	1.19s	2	187	5,126	742
bakery-4	8	0.6s	2	597	4,838	2,500
incdec	32	170.3s	20	41,971	12,762,257	42,252
incdec-inv	32	22.9s	7	873	54,824	4,004
bigjava	44	171m31s	245	122,516	93,410,447	134,828
bigjava1	44	189m16s	304	127,185	95,800,396	139,688
bigjava-inv	44	40m2s	105	7,581	3,378,979	45,103
bigjava1-inv	44	744m14s	217	49,531	48,384,856	104,538

Table 4. Statistics about particular operations

applications. The results are presented in Tables 1 (comparison with HyTech), 2 (comparison with DMC), and 3 (comparison with the Action Language). In each table we present the runtimes and memory consumption (in megabytes). We write > when the compared system could not solve the problem because of the time or memory limit. In this case we present the measurements to the moment when the system was aborted.

Table 1 shows that BRAIN is normally faster than HyTech, and sometimes considerably faster. It also consumes less memory. There are three problems (with the suffix `-inv` in the name) on which HyTech was faster. We make comments on the problems with the suffix `-inv`, called *problems with invariants*, below.

Table 1 shows that BRAIN is normally faster than HyTech, and sometimes considerably faster. It also consumes less memory. There are three problems (with the suffix `-inv` in the name) on which HyTech was faster. We will comment on these problems below.

Table 2 shows that BRAIN is normally several hundred times faster than DMC, except for problems with invariants, where the difference is not so high.

Table 3 shows that BRAIN is normally at least 500 times faster than Action Language, except for problems with invariants, where the difference is not so high. BRAIN also uses less memory.

The problems with invariants were obtained from the original problems by adding *invariants*: some simple properties obtained by forward reachability analysis. A typical invariant has the form $v_1 + \ldots + v_k < m$, where m is a natural number. In fact, it bounds the variables $v_1, \ldots, v_k$ to a finite region. Such invariants cause a problem to BRAIN, because the Basis for problems with such an invariant usually contains all, or a large proportion, of the points in this region explicitly. We believe that this problem is not essential for the approach, but is rather particular to the current implementation of BRAIN in which the basis is stored explicitly, point-wise. A symbolic representation of this finite region, or the use of suitable datastructures for presenting finite-domain variables should solve this problem.

There are several problems which could only be solved by BRAIN, but not by any of the other systems. However, we would like to note that all of these systems are on some benchmarks more powerful than BRAIN since they can use techniques such as widening or transitive closure which the current version of BRAIN does not have. Examples are some versions of the ticket protocol.

To give the reader an idea of the complexity of the problems solved by BRAIN, we present statistics about the number of operations such an entailment-checks performed by BRAIN during each run. This statistics shows why DMC is hopelessly slow on some of these problems: for example, in the case of `csm15` one can hardly check almost 10^9 entailment problems in reasonable time using general-purpose constraint-solving tools. BRAIN solves them in less than 2 hours (on a fast PC with Intel this time would be less than 15 minutes). The table shows that, for most of the benchmarks, entailment seems to be the most important operation. It also demonstrates slowdown of BRAIN on the problems with invariants: indeed the number of operations per second in these problems is much smaller than that for their original formulations without the invariants.

7 Related and future work

In this section we briefly overview related work on systems for infinite-state model checking and algorithms. We do not overview numerous papers related to reachability analysis.

One can implement satisfiability- and entailment-checking using the decision procedure for the first order theory of $\mathbb{I}$. In some cases, one can even use off-the-shelf decision procedures or libraries. For example, [8] use the Omega Library [16] for deciding Presburger arithmetic, [7] use the Composite Library [21], [11] use the $\mathrm{CLP}(R)$ library of Sicstus Prolog. The use of decision procedures for Presburger arithmetic has several advantages, since formulas more general than simple constraints can be handled. As a consequence, one can use non-local reachability algorithms (see [17]), forward reachability, and apply techniques such as widening and transitive closure which cannot be handled by the current version of BRAIN. However, the use of general-purpose algo-

rithms to decide specific classes of formulas may be inefficient, which is confirmed by our experiments.

Other systems for infinite-state model checking not mentioned here are described in [5, 22, 4, 3]. In particular, [5] can prove more safety properties than those obtained by reachability analysis by automatically finding auxiliary invariants and abstraction.

[6] report that HyTech shows better performance on Petri nets that integer-based systems. Moreover, they prove that for Petri nets using relaxation is exact. As we have shown, BRAIN is usually faster than HyTech on integer problems, even without the use of relaxation.

In the future we are going to develop BRAIN into an advanced infinite-state model-checker based on the implementation method proposed here, which will be both faster and more flexible. In particular, we will include in BRAIN other reachability algorithms and techniques such as widening. This will, however, require an implementation of quantifier elimination and an extension of our method, and especially Theorem 7, to constraints with existentially quantified variables and divisibility constraints.

To cope with the problem of large finite regions, one has to introduce their convenient symbolic representation, which may require reworking of all algorithms. It would also be interesting to apply our method to real-valued systems, or systems with both integer- and real-valued variables.

References

1. P.A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160(1-2):109–127, January 2000.
2. F. Ajili and E. Contejean. Avoiding slack variables in the solving of linear Diophantine equations and inequations. *Theoretical Computer Science*, 173(1):183–208, 1997.
3. T. Amnell, G. Behrmann, J. Bengtsson, P.R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K.G. Larsen, M.O. Möller, P. Pettersson, C. Weise, and W. Yi. UPPAAL—Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000*, volume 2067 of *Lecture Notes in Computer Science*, pages 99–124. Springer Verlag, 2001.
4. A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: a tool for reachability analysis of complex systems. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification, 13th International Conference, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 368–372. Springer Verlag, 2001.
5. S. Bensalem, Y. Lakhnech, and S. Owre. InVeST: a tool for verification of invariants. In A.J. Hu and M.Y. Vardi, editors, *Computer-Aided Verification, 10th International Conference, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 505–510. Springer Verlag, 1998.
6. B. Bérard and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99: Concurrency Theory, 10th International Conference*, volume 1664 of *Lecture Notes in Computer Science*, pages 178–193. Springer Verlag, 1999.
7. T. Bultan. Action Language: a specification language for model checking reactive systems. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 335–344, Limerick, Ireland, 2000. ACM.

8. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.
9. E. Contejean and H. Devie. An efficient incremental algorithm for solving of systems of linear diophantine equations. *Information and Computation*, 113(1):143–172, 1994.
10. G. Delzanno. Automatic verification of parametrized cache coherence protocols. In A.E. Emerson and A.P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Verlag, 2000.
11. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
12. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *14th Annual IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 352–359, Trento, Italy, 1999. IEEE Computer Society.
13. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hy-tech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.
14. D. Hilbert. über die Theorie der algebraischen Formen. *Mathematische Annalen*, 36:473–534, 1890.
15. O. Kupferman and M. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
16. W. Pugh. Counting solutions to Presburger formulas: how and why. *ACM SIGPLAN Notices*, 29(6):121–134, June 1994. Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation (PLDI).
17. T. Rybina and A. Voronkov. A logical reconstruction of reachability. submitted, 2002.
18. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1998.
19. A.P. Tomás and M. Filgueiras. An algorithm for solving systems of linear diophantine equations in naturals. In E. Costa and A. Cardoso, editors, *Progress in Artificial Intelligence, 8th Portugese Conference on Artificial Intelligence, EPIA'97*, volume 1323 of *Lecture Notes in Artificial Intelligence*, pages 73–84, Coimbra, Portugal, 1997. Springer Verlag.
20. A. Voronkov. An incremental algorithm for finding the basis of solutions to systems of linear Diophantine equations and inequations. unpublished, January 2002.
21. T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In T. Margaria, editor, *Tools and Algorithms for Construction and Analysis of Systems, 7th International Conference, TACAS 2001*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–66, Genova, Italy, 2001. Springer Verlag.
22. S. Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.