Efficient Temporal-Logic Query Checking for Presburger Systems

Dezhuang Zhang and Rance Cleaveland
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{dezhuang,rance}@cs.sunysb.edu

ABSTRACT

This paper develops a framework for solving *temporal-logic query-checking* problems for a class of infinite-state system models that compute with integer-valued variables (so-called *Presburger* systems, in which Presburger formulas are used to define system behavior). The temporal-logic query-checking problem may be formulated as follows: given a model and a temporal logic formula with *placeholders*, compute a set of assignments of formulas to placeholders such that the resulting temporal formula is satisfied by the given model. Temporal-logic query checking has proved useful as a means for requirements and design understanding; existing work, however, has focused only on propositional temporal logic and finite-state systems.

Our method is based on a symbolic model-checking technique that relies on proof search. The paper first introduces this model-checking approach and then shows how it can be adapted to solving the temporal queries in which formulas may contain integer variables. We also present experimental results showing the computational efficacy of our approach.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods; Model Checking*; D.2.1 [**Software Engineering**]: Requirement / Specification—*Tools*

General Terms

Design, Verification, Experimentation, Performance

Keywords

query checking; Presburger systems; tableau-based symbolic model checking

1. INTRODUCTION

ASE'05, November 7–11, 2005, Long Beach, California, USA.

Temporal-logic query checking [7] has emerged as a useful extension to model checking for supporting requirements and design understanding. The query-checking problem may be formulated as follows: given a model and a temporal logic formula with placeholders (i.e. a *query*), compute a set of assignments of formulas to placeholders such that the resulting temporal formula is satisfied by the given model. For example, solving the CTL query $\mathsf{AG}\ ?_X$, where $?_X$ is a placeholder, for the strongest formula making the query true yields the strongest invariant $(2 \leq x \leq 5) \wedge (3 \leq y \leq 8)$ for the system in Figure 1. Note that x, y are integer variables here and each transition takes a condition and an optional update action. We assume the familiarity of the reader with these intuitive notations.

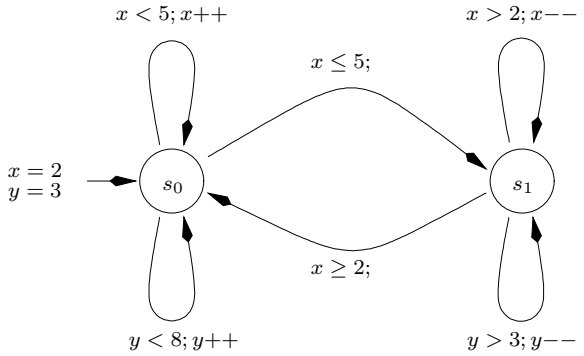

Figure 1: An Intuitive State Machine

In general, temporal-logic query checking is valuable for model understanding. For example, given an early attempt at a specification for a system, one would want to validate some desired temporal-logic properties with a model checker. Some of the properties might fail to hold, in which case one might infer either that the specification requires revision or that the properties are faulty. To determine which is faulty, one can modify formulas into queries in order to retrieve the strongest formulas that makes the query true, and get more diagnostic information to help improve the design. Even if a property is proved to hold in the model, one can still use a query checking to obtain much stronger properties and thus gain more understanding of the system.

As originally proposed by Chan [7], query checking concentrated on valid queries, i.e. queries that always have a unique strongest solution for every system. Recent work has extended this seminal research in several ways. Bruns and

Godefroid [5] studied how to adapt the automata-theoretic model-checking approach to solve the query-checking problem. Gurfinkel *et al.* [11] enriched the query language with multiple placeholders and implemented query checking using a multi-valued model checker. The problem of deciding whether a given query has a unique strongest solution over a given system and how to compute this solution is studied in [13]. The valid-query problem is revisited by [18]. All these works focus on propositional temporal logic and finite-state systems.

The goal of this paper is to develop query-checking techniques for the class of system models that use integer-valued variables (so-called *Presburger* systems, in which Presburger formulas are used to encode both set of states and transitions). Our method is based on a symbolic model-checking technique that relies on proof search. The main data structure is a *predicate equation system*, which is an encoding of the product of a system model and the query formula. Proofs of the validity of the predicate equation corresponding to the start-configuration / query are constructed using proof rules; solutions to a placeholder are inferred at the leaves of a proof tree in order to ensure that the resulting proof is valid.

The principal contributions of our query-checker, which we call CWB-QC(Concurrency Workbench–Query Checking), are the following.

(1) CWB-QC is the first query checker for the class of (infinite-state) Presburger systems. Existing query checkers [11] only deal with finite-state systems. With CWB-QC, formulas and systems can manipulate integer-valued variables and may thus be infinite-state.

(2) Our solution focuses on the *existential* ("find a solution") query-checking problem, as opposed to the *universal* one ("find all solutions"). The latter problem is the usual one studied, but its double exponential time complexity limits its application [5, 11]. However, the applications of query checking that are most often cited [11], existential query checkers can equally well be used, and at much lower computational costs.

(3) Our existential query checker runs as fast as our model checker, and faster and with more precise results than the Action Language Verifier [3], the state-of-the-art model checker for Presburger systems.

The remainder of the paper is organized as follows. Section 2 introduces a symbolic model-checking framework for Presburger systems. The next section shows how the tableau-based symbolic techniques may be adapted to solving temporal-logic queries. We then experimentally illustrate the usage of our query checker with several case studies. The last section gives our conclusions.

2. MODEL CHECKING PRESBURGER SYSTEMS

This section introduces a proof-based symbolic model-checking technique for Presburger systems. This technique will serve as the basis of our query-checking approach.

We begin by introducing some terminology and notation. Throughout let $(x, y, \ldots \in)\mathcal{X}$ be a set of data variables, $(a, b, \ldots \in)\mathbb{N}$ be the set of non-negative integers, and $(t \in)\Pi$ be the set of linear terms of form of $\Sigma_{i=1}^{n} a_i x_i$. Also fix a set of (uninterpreted) actions $(\alpha, \beta, \ldots \in)\mathcal{A}\mathrm{ct}$.

2.1 Presburger Systems

Presburger systems may be thought of as state machines that, in the course of their execution, may modify integer-valued data variables. The tests and modifications to these variables that these state machines may engage in must take the form of so-called Presburger formulas, which represent a restricted subset of logical formulas over integer arithmetic. In this section, we review the definition of Presburger formulas and introduce Presburger systems formally.

Presburger formulas. Presburger formulas are generated by the following BNF grammar, where $x \in \mathcal{X}$ and $t_1, t_2 \in \Pi$.

$$\phi ::= t_1 \leq t_2 \mid \phi \wedge \phi \mid \neg\phi \mid \exists x.\phi$$

We use Φ to represent the set of Presburger formulas in what follows.

Semantically, Presburger formulas are interpreted with respect to *data states* $\rho \in \mathbb{N}^{\mathcal{X}}$ mapping data variables to non-negative integers. We write $\rho \models \phi$ when ρ makes ϕ true; the definition is standard and is omitted. Formula ϕ is called *satisfiable* if there exists ρ such that $\rho \models \phi$.

The satisfiability of Presburger formulas is decidable, although the worst-case time bound is double-exponential in the length of the formula. Efficient procedures [15, 17] do exist to solve the satisfiability problems that arise most often in practice, which typically posses a small number of constraints and do not contain multiple levels of alternating quantifiers [6].

Finally, a Presburger formula ϕ defines a set $[\![\phi]\!]$ of data states in the obvious manner: $[\![\phi]\!] = \{\rho \mid \rho \models \phi\}$.

State-transformation formulas. The definition to come for Presburger systems uses Presburger formulas to test properties of a data state that such a system may be in. It also allows transformations that data states undergo during system execution also to be specified using state-transition formulas, which are defined as follows.

A state transformation specifies how current values of variables will be related to new values after the transformation. To formalize state transformations as Presburger formulas, let $\mathcal{X}' = \{x' \mid x \in \mathcal{X}\}$ represent the set of "primed" versions of data variables. Then a Presburger formula A over the variable set $\mathcal{X} \cup \mathcal{X}'$ may be seen as the specification of a state transformation. For example, consider the formula $y' = x + 1$; this formula is satisfied if the "next" value of y is equal to the "current" value of x incremented by 1. We refer to formulas such as A as state-transition formulas and use $\mathbb{A}$ to represent the set of all such formulas.

Semantically, state-transformation formulas are interpreted with respect to pairs $\langle \rho, \rho' \rangle$ of data states, where ρ represents the "current" state and ρ' the "next" state. We write $\langle \rho, \rho' \rangle \models A$ when A is made true by taking the values for the variables of $\mathcal{X}$ from ρ and the variables of $\mathcal{X}'$ from ρ'.

Presburger systems. Presburger systems may be seen as symbolic state machines, with a finite sets of control locations and Presburger formulas and state transformations used to show how the data variables are modified as control locations are updated.

Definition 1. A *Presburger system* (PS) is a tuple $\langle S, R, S_I, \phi_I \rangle$, where S is a finite set of control locations; $R \subseteq S \times \Phi \times$

$\mathbb{A} \times \mathcal{A}\text{ct} \times S$ is a finite set of transitions; $S_I \subseteq S$ are the initial locations; and $\phi_I \in \Phi$ is the initial condition.

Intuitively, S_I contains the possible starting locations and ϕ_I the initial conditions on data variables. Based on the current control location and state of the data variables, transitions whose $\phi \in \Phi$ components are true may fire, with data variables being updated in a manner consistent with the transition's state-transformation formulas. These notions are made precise by interpreting PSs semantically using *concrete transition systems*.

Definition 2. A *concrete transition system* (CTS) is a tuple $\langle \Sigma, V \rightarrow_c, \Sigma_I \rangle$, where Σ is the set of states, $V : \Sigma \rightarrow \mathbb{N}^{\mathcal{X}}$ the valuation function, $\rightarrow_c \subseteq \Sigma \times \mathcal{A}\text{ct} \times \Sigma$ the transition relation, and $\Sigma_I \subseteq \Sigma$ the set of start states.

Given a PS $G = \langle S, R, S_I, \phi_I \rangle$, CTS $C_G = \langle \Sigma, V, \rightarrow_c, \Sigma_I \rangle$ is given as follows.

1. $\Sigma \subseteq S \times \mathbb{N}^{\mathcal{X}}$
2. $V(\langle s, \rho \rangle) = \rho$
3. $\langle s, \rho \rangle \xrightarrow{\alpha} \langle s', \rho' \rangle$, iff there is $\langle s, \phi, A, \alpha, s' \rangle \in R$, with $\rho \models \phi$ and $(\rho, \rho') \models A$
4. $\Sigma_I = \{\langle s_I, \rho \rangle \mid s_I \in S_I, \rho \models \phi_I\}$

As an example, we define the PS $G = \langle S, R, S_I, \phi_I \rangle$ for the system in Figure 1 as follows, where τ is a special internal action.

- $S = \{s_0, s_1\}$
- $R = \bigcup \begin{array}{l} \{s_0, x < 5; x' = x + 1; \tau; s_0\} \\ \{s_0, y < 8; y' = y + 1; \tau; s_0\} \\ \{s_0, y \leq 5; ; \tau; s_1\} \\ \{s_1, x > 2; x' = x - 1; \tau; s_1\} \\ \{s_1, y > 3; y' = y - 1; \tau; s_1\} \\ \{s_1, x \geq 2; ; \tau; s_0\} \end{array}$
- $S_I = \{s_0\}$
- $\phi_I = (x = 2 \wedge y = 3)$

2.2 The Presburger Modal Mu-Calculus

Temporal-logic query checking requires a "base logic" in which to write system requirements. In query-checking work, variants of the temporal logic CTL [8] are typically used for this purpose. In this work, we consider a different, lower-level, but more expressive logic that allows the definition of recursive formulas. We call this logic the Presburger Modal Mu-Calculus.

Our formulas are defined using *modal equation systems* (MESs). MESs consist of blocks of equations of the form $X = \psi$, where $X \in \mathbb{X}$ is a *formula variable* and ψ is a formula defined by the following grammar.

$$\psi \quad ::= \quad \phi_s \mid \psi \vee \psi \mid \psi \wedge \psi \mid \langle \alpha \rangle \psi \mid [\alpha] \psi \mid X$$

In the above, ϕ_s is a Presburger formula, and α is an action. Operators $\langle \alpha \rangle$ and $[\alpha]$ are called modal operators; these, together with $\vee$ and $\wedge$, are standard from the propositional modal mu-calculus [16].

The definition $fpv(\psi)$ of free formula variables in ψ is standard. We call a formula ψ *formula-closed* if $fpv(\psi) = \emptyset$. While negation is restricted in the logic to the Presburger subformulas, every formula-closed formula ψ has a formula $\mathsf{not}(\psi)$ that is semantically equivalent to ψ's negation.

MESs are intended to define a mutually recursive family of formulas, one for each equation. Since a given equation can have several solutions, blocks in MESs are equipped with an indication as to whether the "least" (most restrictive) "greatest" (most permissive) solution is intended.

Definition 3. An *equation block* has form $\langle p, \overline{E} \rangle$, where $p \in \{\mu, \nu\}$ is the *parity indicator* and $\overline{E} = \langle E_1, \ldots, E_n \rangle$ is a finite sequence of equations of form $X_i = \psi_i$, with the X_i distinct predicate variables and each ψ a formula.

In a block $\langle p, \overline{E} \rangle$ the parity indicator determines whether the "greatest" (ν) or "least" (μ) solution of the equations is intended. We write $\mathsf{lhs}(B) = \{X_1, \ldots, X_n\}$ for the set of left-hand-side variables in block B and $\mathsf{rhs}(B) = \{\psi_1, \ldots, \psi_n\}$ for the right-hand-side formulas. As an example, the block

$$X \stackrel{\mu}{=} (2 \leq x \leq 5) \wedge (3 \leq y \leq 8) \vee \langle \tau \rangle X \tag{1}$$

consists of a single equation for which we want to compute the the least fixpoint of the formula variable X. While the block

$$Y \stackrel{\nu}{=} X \wedge [\tau] Y \tag{2}$$

defines a single equation, where X is a free formula variable. Here we are interested in the greatest solution for Y.

Definition 4. A *modal equation system* is a finite sequence $\langle B_1, \ldots, B_n \rangle$ of equation blocks with the property that if $i \neq j$, then $\mathsf{lhs}(B_i) \cap \mathsf{lhs}(B_j) = \emptyset$.

The notions of lhs and rhs can be extended in the obvious manner to MESs. We call a MES E formula-closed if $\bigcup_{\psi \in \mathsf{rhs}(P)} fpv(\psi) \subseteq \mathsf{lhs}(P)$. For example, combine the equation block (1) and (2), we get a formula-closed MES.

$$\left\{ \begin{array}{lll} X & \stackrel{\mu}{=} & (2 \leq x \leq 5) \wedge (3 \leq y \leq 8) \vee \langle \tau \rangle X \\ Y & \stackrel{\nu}{=} & X \wedge [\tau] Y \end{array} \right.$$

The semantics of the modal formulas is given with respect to a CTS $C = \langle \Sigma, V, \rightarrow_c, \Sigma_I \rangle$, and takes the form of a relation $\sigma \models_{C,\theta} \psi$, which, given an environment $\theta : \mathbb{X} \mapsto 2^{\Sigma}$ mapping formula variables to sets of states, determines whether or not CTS state σ satisfies ψ. This relation may be given as follows (obvious cases omitted).

$\sigma \models_{C,\theta} \phi_s$ iff $\mathcal{V}(\sigma) \models \phi_s$
$\sigma \models_{C,\theta} X$ iff $\sigma \in \theta(X)$
$\sigma \models_{C,\theta} \langle \alpha \rangle \psi$ iff there is σ' s.t. $\sigma \xrightarrow{\alpha}_c \sigma'$ and $\sigma' \models_{C,\theta} \psi$
$\sigma \models_{C,\theta} [\alpha] \psi$ iff for all σ' s.t. $\sigma \xrightarrow{\alpha}_c \sigma'$, $\sigma' \models_{C,\theta} \psi$

We define $[\![\psi]\!]_{C,\theta} = \{\sigma \mid \sigma \models_{C,\theta} \psi\}$. We may now apply general fixpoint theory, as elaborated in [19], to define the semantics of MESs. The basis of the definition involves defining the semantics of a single block in an MES. According to Knaster-Tarski fixpoint theorem [20], every monotonic function over a complete lattice has a unique least fixpoint and a unique greatest fixpoint. As shown, for example, in [19], these fixpoints may also be interpreted as the "least permissive" and "most permissive" solutions to systems of equations defined over lattices. In the case of an MES block, the lattice in question is the set of environments

mapping left-hand side variables to subsets of Σ, with the lattice ordering being the pointwise extension of set containment to environments. The MES induces a monotonic function on this lattice that evaluates each right-hand side in the MES in the context of the "argument environment" and constructs an environment assigning these values to the appropriate left-hand side variables. These ideas can then be used to give semantics to lists of blocks; the details are complicated but not deep and can be found in [19, 22].

The greatest and least fixpoints of monotonic functions also have iterative characterizations that give rise to approximation-based approaches to computing them, when they can be computed. In the case of the least fixpoint of a monotonic function f, one starts with the least element in the lattice ($\bot$) and repeatedly applies f ($f(\bot), f(f(\bot))$, etc.) until the result does not change; this "fixed" result is the least fixpoint. Characterizing the maximum fixpoint is the same, except that the initial value is the maximum element of the lattice. In the case of MES blocks, the least value is the environment assigning $\emptyset$, the empty set of states, to each variable, while the greatest value is the environment assigning Σ, the complete set of states, to each variable.

We use $[\![M]\!]_{C,\theta}$ to represent the semantics of MES M with respect to CTS C and an environment θ that is used to interpret free predicate variables in M. If M is formula-closed then $[\![M]\!]_{C,\theta}(X) = [\![M]\!]_{C,\theta'}(X)$ for any $X \in \mathsf{lhs}(M)$ and θ, θ', and we write $[\![M]\!]_C$ for this value. It also follows that if M is formula-closed and ψ is such that $\mathit{fpv}(\psi) \subseteq \mathsf{lhs}(M)$, then $[\![\psi]\!]_{[\![M]\!]_{C,\theta}} = [\![\psi]\!]_{[\![M]\!]_{C,\theta'}}$ for any θ, θ'. When this holds we use $[\![\psi]\!]_{C,M}$ for this value, and we write $\sigma \models_{C,M} \psi$ if $\sigma \in [\![\psi]\!]_{C,M}$,

The definition of C_G implies an immediate interpretation of the mu-calculus with respect to PS G. In addition to the other notations defined for the mu-calculus, we also introduce the following. Let ψ be a mu-calculus formula, and s a control location in PS G, and let θ be a mapping of mu-calculus formula variables to sets of states in C_G paired with alternative system states. Then $[\![\psi]\!]_\theta(s) = \{\rho \mid \langle s, \rho\rangle \in [\![\psi]\!]_{C_G,\theta}\}$. That is, the "semantics" of a control location s vis à vis a formula is the set of system states that, when combined with s, make the formula "true". Similarly, if M is a formula-closed MES, and ψ is a mu-calculus formula with $\mathit{fpv}(\psi) \subseteq \mathsf{lhs}(M)$, we write $[\![\psi]\!]_{G,M}(s)$ for $\{\rho \mid \langle s, \rho\rangle \in [\![\psi]\!]_{C_G,M}\}$. In this case, we also say that an PS G satisfies a mu-calculus formula ψ with respect to equation system M (written $G \models_M \psi$) if for all $s_I \in S_I$, $\{\rho \mid \rho \models \phi_I\} \subseteq [\![\psi]\!]_{G,M}(s_I)$.

As a modal mu-calculus, MESs are expressive enough to encode many temporal logics, including CTL. For example, the invariant property $\mathsf{AG}\phi$ can be defined as $X \stackrel{\nu}{=} \phi \wedge [\tau]X$, the reachability formula $\mathsf{EF}\phi : X \stackrel{\mu}{=} \phi \vee \langle\tau\rangle X$, where τ is a special action that can label any transition (because there is no action label in CTL); the bounded liveness property $\mathsf{A}\phi\mathsf{W}\varphi : X \stackrel{\nu}{=} \varphi \vee (\phi \wedge [\tau]X)$; and $\mathsf{EX}\phi : \langle\tau\rangle\phi$ etc. We sometimes use these CTL operators as shorthands to specify query formulas whenever convenient.

2.3 Predicate Equation Systems

In this section we introduce predicate equation systems (PESs), which will be the vehicles we use for model checking and solving the query-checking problem. PESs are like MESs with each equation taking the form of $X = \psi$, where X is a *predicate variable* and ψ is a *predicate formula* defined by the following grammar,

$$\psi \quad ::= \quad \phi_s \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi[A] \mid X$$

In the preceding, ϕ_s is a Presburger formula, while A is a state-transformation formula. All operators are the usual ones, expect for $\psi[A]$. This operator may be thought of as a generalization of the substitution operation. To define it precisely, if ρ is a data state then define $\mathsf{post}(\rho, A) = \{\rho' \mid \langle\rho, \rho'\rangle \models A\}$ to be the "post-states" of ρ after A. Then $\rho \models \psi[A]$ holds exactly when every post-state $\rho' \in \mathsf{post}(\rho, A)$ satisfies ψ ($\rho' \models \psi$).

The definition *fpv* of free predicate variables is standard. We call a predicate ϕ *predicate-closed* if $\mathit{fpv}(\phi) = \emptyset$.

The semantics of predicates and PESs can be adapted from MESs in the obvious way. Now the lattice in question is $2^{\mathbb{N}^{\mathcal{X}}}$ (i.e. the lattice of sets of system states, ordered by set inclusion). A given predicate formula ϕ containing free predicate variable X, in the context of a predicate environment θ, may be seen as a monotonic function f_θ over this lattice as follows: $f_\theta(S) = [\![\phi]\!]_{\theta[X:=S]}$. Given a "starting" environment θ, the semantics, $[\![P]\!]_\theta$, of PES P is an environment θ' that, for any equation $X = \psi$ of P, satisfies: $\theta'(X) = [\![\psi]\!]_{\theta'[X:=\theta'(X)]}$ and is appropriately extremal. Note that if P is predicate-closed, then $[\![P]\!]_\theta(X) = [\![P]\!]_{\theta'}(X)$ for any $X \in \mathsf{lhs}(P)$ and θ, θ', and if ψ is additionally a predicate with $\mathit{fpv}(\psi) \subseteq \mathsf{lhs}(P)$, then $[\![\psi]\!]_{[\![P]\!]_\theta} = [\![\psi]\!]_{[\![P]\!]_{\theta'}}$ for any θ, θ'. In this case we write $[\![\psi]\!]_P$ for this common value, and if $\rho \in [\![\psi]\!]_P$ we represent this notationally as $\rho \models_P \psi$.

2.4 Model Checking and PESs

The Presburger system model-checking problem asks: given PS G, formula-closed MES M and $X \in \mathsf{lhs}(M)$, does $G \models_M X$? This section shows how to translate this question into an equivalent one involving PESs.

The translation from an PS G and an MES M to a PES is achieved by constructing a PES equation for each control location in G and equation in M. Formally, we define a function F that, given an PS G and formula-closed mu-calculus equation system M, yields a predicate-closed PES $F(G, M)$. F is applied on a block-by-block basis; that is, $F(G, \langle B_1, \ldots, B_n\rangle) = \langle F(G, B_1), \ldots, F(G, B_n)\rangle$. And $F(G, B) = F(G, \langle p, \overline{E}\rangle)$ in turn yields a predicate equation block of form $\langle p, \overline{E'}\rangle$, where for each equation $X = \psi$ in $\overline{E}$ and control location s in G, there is an equation of form $Y_{s,X} = F(s, \psi)$ in $\overline{E'}$. $F(s, \psi)$ is defined as follows.

$$\begin{array}{lcl}
F(s, \phi_s) & = & \phi_s \\
F(s, \psi_1 \vee \psi_2) & = & F(s, \psi_1) \vee F(s, \psi_2) \\
F(s, \psi_1 \wedge \psi_2) & = & F(s, \psi_1) \wedge F(s, \psi_2) \\
F(s, X) & = & Y_{s,X} \\
F(s, \langle\alpha\rangle\psi) & = & \bigvee\{\phi \wedge (F(s', \psi)[A]) \mid \\
 & & \langle s, \phi, A, \alpha, s'\rangle \in R\} \\
F(s, [\alpha]\psi) & = & \bigwedge\{\phi \rightarrow (F(s', \psi)[A]) \mid \\
 & & \langle s, \phi, A, \alpha, s'\rangle \in R\}
\end{array}$$

THEOREM 1. *Let $G = \langle S, R, S_I, \phi_I\rangle$ be an PS, and let M be a closed modal equation system. Then for any $s \in S$ and any $X \in \mathsf{lhs}(M)$, we have that $[\![X]\!]_{G,M}(s) = [\![Y_{s,X}]\!]_{F(G,M)}$.*

It follows that $G \models_M X$ iff the following statement is true,

$$\mathbb{N}^{\mathcal{X}} \quad = \quad [\![\phi_I \rightarrow \bigwedge_{s_I \in S_I} Y_{s_I,X}]\!]_{F(G,M)}$$

2.5 Local Model Checking as Proof Search

We now introduce a "goal-directed" proof system in Fig 2. The proof system establishes when a set of predicate-closed formulas $\Phi \stackrel{\text{def}}{=} \{\phi_1, \ldots, \phi_n\}$ implies a formula ψ potentially containing predicate variables from a PES P. The proof rules operates on *sequents* of the form: $\Phi \vdash_P \psi$, which we shall interpret as the formula $\bigwedge \Phi \rightarrow \psi$. The rules follow the following syntactic conventions: ϕ, ϕ_i, φ are predicate closed, while ψ, ψ_i need not be; and Φ, ϕ is short-hand for $\Phi \cup \{\phi\}$. Conclusions are also written above subgoals, which are separated by a ";".

$$\vee_1 \frac{\Phi \vdash_P \psi_1 \vee \psi_2}{\Phi \vdash_P \psi_1} \qquad \vee_2 \frac{\Phi \vdash_P \psi_1 \vee \psi_2}{\Phi \vdash_P \psi_2} \qquad \vee_3 \frac{\Phi \vdash_P \phi \vee \psi}{\Phi, \mathsf{not}(\phi) \vdash_P \psi}$$

$$\vee_4 \frac{\Phi \vdash_P \psi \vee \phi}{\Phi, \mathsf{not}(\phi) \vdash_P \psi} \qquad \wedge \frac{\Phi \vdash_P \psi_1 \wedge \psi_2}{\Phi \vdash_P \psi_1 \quad ; \quad \Phi \vdash_P \psi_2}$$

$$\vee \frac{\Phi \vdash_P \psi_1 \vee \psi_2}{\Phi, \phi \vdash_P \psi_1 \quad ; \quad \Phi, \neg\phi \vdash_P \psi_2} \qquad [] \frac{\Phi \vdash_P \psi[A]}{\mathsf{post}(\Phi, A) \vdash_P \psi}$$

$$C \frac{\Phi \vdash_P X}{\Phi \vdash_P \psi} \; (X = \psi \; \in P \;)$$

Figure 2: A Gentzen-like Proof System for PESs.

Rules $\vee_1 - \wedge$ are familiar from the standard predicate calculus; not function "drives" negations inside; Intuitively, rule $\vee$ is used for splitting conditions, The remaining rules are for the substitution operator and predicate variables. $\mathsf{post}(\Phi, A) \stackrel{\text{def}}{=} \{\rho' \mid \rho \models \Phi \text{ and } (\rho, \rho') \in A\}$ defines the strongest postcondition of Presburger formulas Φ wrt. the state transformation A. These postconditions may also be represented as Presburger formulas.

The rules also share an implicit side condition: they may only be applied to *non-leaf* sequents. These are defined as follows.

Definition 5. Let σ be a sequent of form $\Phi \vdash_P \psi$. σ is a *(successful) leaf* if one of the following conditions holds. (a). ψ is a Presburger formula (successful if $\bigwedge \Phi \rightarrow \psi$ is a tautology). (b). $\psi \in \Phi$ (always successful). (c). $\psi = X[A]$ (A may be empty) with parity p, and there is another sequent σ' of form $\Phi' \vdash_P X[A']$ on the path from the root node of the proof to σ with the property that no $\sigma'' \vdash_P X''[A'']$ such that X'' has parity different than p and X'' is defined in an earlier block in the PES than X, and $\mathsf{post}(\Phi, A)$ logically implies $\mathsf{post}(\Phi', A')$. (In this case, σ' is called the *companion* of σ) Such a leaf is successful if the parity of X is ν.

The definition of (successful) leaf is based on the one given in [14], which also gives a success criterion for leaves involving μ-formulas. This criterion is not needed in this paper, so we omit further mention of it.

A proof built using these rules is *valid* if and only if it is finite, every path ends in a leaf, and every leaf is successful. The following is true.

THEOREM 2. *The proof rules in Figure 2 are sound: if $\Phi \vdash_P \psi$ has a valid proof then $[\![\Phi \vdash_P \psi]\!]_P = \mathbb{N}^{\mathcal{X}}$, where P is the PES containing the definitions of the predicate variables.*

In general, the proof rules will not be complete, although Presburger arithmetic is decidable. This is because proofs of certain sequents may require infinite-depth trees (i.e. fixpoint computation does not converge in finite time [10]). To overcome this, conservative approximation techniques and acceleration heuristics (e.g. [6]) may be needed to achieve convergence of an approximate fixpoint computation. On the other hand, if all the data variables are bounded (i.e. take values from a range), then the proof system is complete. In this case, the set of Presburger formulas are finite. The argument of the completeness is similar to the propositional model checking [9].

3. SOLVING QUERIES ON TABLEAUX

In this section, we describe our query checking approach for the Presburger modal mu-calculus. Queries in our setting will consist of formulas in the mu-calculus augmented with *placeholders* of form $?_X$, where X is a formula variable used only for identification purposes. Placeholders may also have negation applied to them in queries: so $\neg?_X$ may also appear within a query. A query may also have multiple placeholders $?_X, ?_Y$, etc., distinguished by the formula variables labeling the placeholders. For technical convenience, throughout of the paper we assume that placeholders are different from formula variables, and thus that queries are "formula closed". We call an occurrence of the placeholder $?_X$ is *positive* in a query ψ if it appears under no negation in the query ψ, and *negative* if it appears negated. A placeholder $?_X$ is *pure* in a query ψ if all of its occurrences have the same polarity (positive or negative), and *mixed* otherwise.

A *query problem* consists of a query formula ψ and a PS G; a *solution* to such a problem is an assignment of formulas to placeholders in ψ such that G satisfies the resulting mu-calculus ψ' obtained by replacing the placeholders in ψ by their formulas. In general, a query problem can have many solutions; we are particularly interested in *strongest / weakest* solutions, when they exist.

Sometimes we are interested in a subset of data variables in a solution to a placeholder. This is done with a *projection* operator ": {}" after the placeholder. For example, suppose $\mathcal{X} = \{x, y, z\}$, we only care about variable x and y, then $?_X : \{x, y\}$ projects the solution from $\mathbb{N}^{\{x,y,z\}}$ to $\mathbb{N}^{\{x,y\}}$.

3.1 A Simple Example

We use a simple example to show how our query checking works. Considering the query formula $\mathsf{AG}?_X$, that is

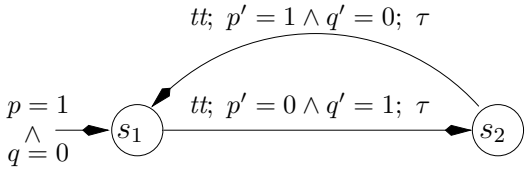

Figure 3: A Simple Transition Graph

$Y \stackrel{\nu}{=} (?_X \wedge [\tau]Y)$, against the transition system in Figure 3, computing the "solution" for $?_X$ amounts to answering the question "What's the strongest invariant in the system?". Note that data variables p and q are unbounded. We show that this problem can be solved with our Gentzen-like proof system. Taking the placeholder $?_X$ as a Presburger formula, we translate the query formula and the system model into the following PES.

$$\begin{cases} Y_1 \stackrel{\nu}{=} & ?_X \wedge Y_2[p' = 0 \wedge q' = 1] \\ Y_2 \stackrel{\nu}{=} & ?_X \wedge Y_1[p' = 1 \wedge q' = 0] \end{cases}$$

Then the query checking problem is reduced to finding the strongest formula that can replace $?_X$ such that $\Phi \rightarrow Y_1$ is a tautology, where $\Phi \stackrel{\mathsf{def}}{=} (p = 1 \wedge q = 0)$ is the initial condition of the transition system, and Y_1 is generated from the initial control location s_1 and the formula variable Y. Applying the proof rules, we get the following tableau.

<table>
<tr><td colspan="3">$\Phi \vdash Y_1$</td></tr>
<tr><td colspan="3">$\Phi \vdash ?_X \wedge Y_2[p' = 0 \wedge q' = 1]$</td></tr>
<tr><td>$\Phi \vdash ?_X$</td><td colspan="2">$\Phi \vdash Y_2[p' = 0 \wedge q' = 1]$</td></tr>
<tr><td></td><td colspan="2">$\Phi \vdash ?_X[p' = 0 \wedge q' = 1] \wedge Y_1[p' = 1 \wedge q' = 0]$</td></tr>
<tr><td></td><td>$\Phi \vdash ?_X[p' = 0 \wedge q' = 1]$</td><td>$\Phi \vdash Y_1[p' = 1 \wedge q' = 0]$</td></tr>
<tr><td></td><td>$p = 0 \wedge q = 1 \vdash ?_X$</td><td>A ν-leaf is reached</td></tr>
</table>

For this tableau (proof structure) to be successful, all its leaves must be successful. Now:

- The ν-leaf $\Phi \vdash Y_1[p' = 1 \wedge q' = 0]$ is successful.
- For sequent $\Phi \vdash ?_X$ to be a successful leaf, we need $p = 1 \wedge q = 0 \rightarrow ?_X$ to be a tautology. The strongest formula to replace $?_X$ such that this sequent is valid would be Φ itself, that is $(p = 1 \wedge q = 0)$.
- Similarly, for leaf $p = 0 \wedge q = 1 \vdash ?_X$ to be successful, the strongest formula is $(p = 0 \wedge q = 1)$.

Therefore, to make the proof successful, we have the strongest

$$?_X \stackrel{\mathsf{def}}{=} (p = 1 \wedge q = 0) \vee (p = 0 \wedge q = 1)$$

3.2 Existential Query Checking

The above example suggests an efficient symbolic approach for solving query problems: determine the solutions to placeholders at the leaves of a *potentially successful tableau* (PST) that arise when applying our model-checking procedure to a query. By PST, we mean a tableau whose leaves may contain occurrences of placeholders and which could be identified as successful by appropriate assignments to these placeholders. For queries, it may be shown that leaves containing placeholders in a PST may have one of two forms: $\phi \vdash ?_X$ or $\phi \vdash \neg ?_X$, where ϕ is predicate closed and *placeholder closed*, (i.e. without any occurrence of any placeholder). We call such leaf sequents *potentially successful leaves*. Note that the occurrence of $?_X$ in $\phi \vdash ?_X$ is positive, and the occurrence of $?_X$ in $\phi \vdash \neg ?_X$ is negative.

Let G be a PS and ψ a query formula with (possibly multiple) placeholders$?_X(, \ldots, ?_Y)$. To solve ψ against G, i.e. give a solution to all the placeholders we compute the PES (augmented with queries) from the model-checking problem for G and ψ, treating placeholders like Presburger formula in the translation; we call the resulting extended PES a *query predicate*, since it contains placeholders. We then search for a potentially successful tableau T by applying proof rules to the query predicate.

Existential query checking provides solutions to placeholders according to a single (potentially) successful tableau. Once we have identified a potentially successful tableau T, we compute solutions for the placeholders as follows.

1. The solution to the *positive* occurrences of placeholder $?_X$ with respect to T, written as $[\![?_X^+]\!]_T$, is given by

$$[\![?_X^+]\!]_T \stackrel{\mathsf{def}}{=} \bigvee \{ \ \phi \ \mid \ \phi \vdash ?_X \text{ is a sequent in } T \}$$

2. The solution to the *negative* occurrences of placeholder $?_X$ with respect to T, written as $[\![?_X^-]\!]_T$, is given by

$$[\![?_X^-]\!]_T \stackrel{\mathsf{def}}{=} \bigwedge \{ \ \mathsf{not}(\phi) \ \mid \ \phi \vdash \neg ?_X \text{ is a sequent in } T \}$$

We now have the following.

THEOREM 3. *Let T be a potentially successful tableau for a PES constructed from query formula ψ and PS G, and let $?_X$ be a placeholder. Then replacing all occurrences of $?_X$ in T by any formula ϕ such that $[\![?_X^+]\!]_T$ implies ϕ and ϕ implies $[\![?_X^-]\!]_T$ results in a tableau that can be extended into a successful tableau.*

In other words, this theorem asserts that $[\![?_X^+]\!]_T$ and $[\![?_X^-]\!]_T$ "bound" the solutions to the query problem that can be inferred from the PST T.

When $?_X$ is pure and positive (i.e. $?_X$ has no negative occurrences in ϕ), then it may be shown that $[\![?_X^-]\!]_T = \bigwedge \emptyset = \mathit{true}$; and similarly for the case when $?_X$ is pure and negative we have $[\![?_X^+]\!]_T = \bigvee \emptyset = \mathit{false}$. These observations lead to the following corollary of the above theorem.

COROLLARY 1. *Let T be a potentially successful tableau for a PES constructed from query formula ψ and PS G.*

1. *If $?_X$ is a placeholder in ψ, then $[\![?_X^+]\!]_T$ is the strongest formula ϕ such that T may be extended into a successful tableau when each positive occurrence of $?_X$ is replaced by ϕ.*
2. *If $?_X$ is a placeholder in ψ, then $[\![?_X^-]\!]_T$ is the weakest formula ϕ such that T may be extended into a successful tableau when each negative occurrence of $?_X$ is replaced by ϕ.*

For instance, the positive solution to the placeholder $?_X$ in section 3.1 is $[\![?_X^+]\!] \stackrel{\mathsf{def}}{=} (p = 1 \wedge q = 0) \vee (p = 0 \wedge q = 1)$, while the negative solution is $[\![?_X^-]\!] \stackrel{\mathsf{def}}{=} \mathit{true}$. Therefore, any formula that is implied by $(p = 1 \wedge q = 0) \vee (p = 0 \wedge q = 1)$ and implies *true*, e.g. $p = 0 \vee p = 1$, $q \leq 5$, or $p \geq 0$ etc., can be used to replace $?_X$ in the query formula $\nu Y = (?_X \wedge [\tau]Y)$, makes the model-checking problem successful.

All the applications of the temporal-logic query checking established for propositional systems [11], such as *reachability analysis*, discovering invariants, *guard discovery*, *guided simulation* and *test case generatation* etc. can be formulated immediately as existential query-checking for Presburger systems. The full paper will consider this point more fully.

3.3 Universal Query Checking

In general, a query problem may give rise to many PSTs, each yielding a bound on the solution for each placeholder. *Universal query checking* provides multiple solutions by the means of all (potentially) successful tableaux.

Previous works [11] shows that solving a propositional tempo-ral-logic query with a single placeholder takes 2^{2^n} times slower than checking an equivalent model-checking

property, where n is the number of atomic propositions in the system. Application of these algorithms to industrial systems turns out to be impractical, since a moderate system might contain dozens of boolean variables, not mention of even a single unbounded integer variable in the system.

In our setting, universal query checking has to find all potentially successful tableaux, while existential query checking only needs to locate one (potentially) successful tableau; Consequently, existential query checking has the same time complexity as local model checking, while universal query checking might take a longer time(since tableaux must be enumerated). Therefore, the existential query checking leads to a significant advantage in practice.

Note that for queries involving in an invariant or bounded liveness property, which basically need to compute a single greatest fixpoint, the universal query checking might have the same time complexity as the existential one, because there are only a very small number of potentially successful tableaux in most cases. For example, the query $\mathsf{AG}?_X$ as in section 3.1 only leads to one PST.

4. EXPERIMENTAL RESULTS

To evaluate the performance of our query-checking technique, we have built a prototype called CWB-QC (Concurrency Workbench–Query Checking). The algorithm uses a depth-first search technique with caching. The proof rules in Figure 2 are used to generate sequents needed to be proved next in order for the goal sequent to be true. The cache contains sequents that have either been proved or disproved, or are currently assumed to be true. When a sequent is generated, the cache is first checked to see if it is implied by something in the cache; if this is the case, then no more searching is necessary for this sequent. If the sequent is not in the cache, it is added into the cache, and rules are then recursively applied to it. The precise details of cache management are similar to those for on-the-fly propositional model checkers [1, 4], so we omit further discussion here.

The implication-checking procedure we employ is the MONA tool [15], which provides an automata-based decision procedure for WS1S (weak monadic second order logic with one successor) and takes BDDs to represent the internal transitions. All the experimental data was collected on an Intel Pentium III 700MHZ CPU and 512MB memory laptop, running Linux 2.4.

4.1 A Simple Thermostat

As an example, we analyze the requirements of a simple thermostat; this example is adapted from the previous specification in [2]. The target system is responsible for keeping the room's temperature in a moderate range between *low* and *high* if *Switch* is on. The SCR specification [12] of the system is given in Figure 4.

An SCR requirements specification models the system in an event-driven fashion. The input interface of the system is given as a set of monitored variables and the output interface as a set of controlled variables. For example, the thermostat reads the sensor of the room temperature and the status of the switch; and it controlls variables for the power switches of the heat and air conditioner. The state space is partitioned into sets of states called *modes*. The system changes its state due to conditioned events. For example, the event @T(SwitchIsOn) represents condition "switch is turned On from Off at next state" and @T(TooCold) describes the condition "*temp* < *low* becomes true at next state", while condition SwitchIsOn says "switch is On" at the current state.

Constants:	low, high : Integer;
Monitored Variables:	temp : Integer;
	Switch : {On,Off}
Mode Class:	Thermostat : {Off,Inactive,Heat,AC}
Initial Conditions:	Switch=Off;
	Thermostat=Off;
	low < high;

$$\text{TooCold} \stackrel{\text{def}}{=} \text{temp} < \text{low}$$
$$\text{TempOK} \stackrel{\text{def}}{=} \text{temp} \geq \text{low} \;\&\; \text{temp} \leq \text{high}$$
$$\text{TooHot} \stackrel{\text{def}}{=} \text{temp} > \text{high}$$

Old Mode	SwitchIsOn	Event	New Mode
Off	@T	TooCold	Heat
	@T	@T(TooCold)	
	@T	TempOK	Inactive
	@T	@T(TempOK)	
	@T	TooHot	AC
	@T	@T(TooHot)	
Inactive	@F	–	Off
	t	@T(TooCold)	Heat
	t	@T(TooHot)	AC
Heat	@F	–	Off
	t	@T(TempOK)	Inactive
AC	@F	–	Off
	t	@T(TempOK)	Inactive

Figure 4: SCR specification of a simple Thermostat

Note that in the specification given in Figure 4, values of the constants *low*, *high* are unspecified. These constants can take any integer value as long as they satisfy the ordering *low* ≤ *high*. Our representation of the system also leaves these constants as unspecified.

Modeling the thermostat as a PS. We model the Thermostat as a symbolic transition system $T \stackrel{\text{def}}{=} \langle S, R, S_I, \phi_I \rangle$, where $S = S_I = \{s\}$, $\phi_I \stackrel{\text{def}}{=}$ (Switch = Off & Thermostat = Off), and the set of transition R is defined as what follows.

⟨ s, Thermostat = Off & Switch = Off & temp ≤ low, Switch′ = On & Thermostat′= Heat, τ, s ⟩ ;

⟨ s, Thermostat = Off & Switch = Off & temp ≥ low, Switch′ = On & temp < low & Thermostat′ = Heat, τ, s ⟩ ;

. . .

⟨ s, Thermostat = AC & Switch = On & (temp > high | temp < low), low < temp′ < high & Thermostat′ = Inactive, τ, s ⟩ ;

Query formulas. Invariants summarize relationships between data variables in the model and are often useful to add confidence to system designers. For example, we can use the following queries to find interesting invariants in each mode.

(a). AG (Thermostat=Off $\rightarrow ?_{X_1}$:{Switch})

(b). AG (Thermostat=Inactive $\rightarrow ?_{X_2}$:{Switch,temp,low,high})

(c). AG (Thermostat=Heat $\rightarrow ?_{X_3}$:{Switch,temp,low})

(d). AG (Thermostat=AC $\rightarrow ?_{X_4}$:{Switch,temp,high})

To check the status of the thermostat under different conditions, one can use the queries

$$\mathsf{AG}\ (\text{Thermostat=Heat}) \rightarrow \mathsf{EX}?_{X_5} : \{\text{Thermostat}\}$$
$$\mathsf{AG}\ (\text{Switch=On \& temp<low}) \rightarrow ?_{X_6} : \{\text{Thermostat}\}$$

The query $\mathsf{AG}(?_{X_7} : \{\text{Switch}\} \rightarrow \text{Thermostat=Heat})$ can return the status of the switch when the Thermostat is heating, and $\mathsf{AG}?_{X_8}$:{Thermostat} can return all reachable modes in the system.

Performance results. One can check each of the above query formulas separately, i.e. run CWB-QC multiple times, or query the conjunction of all the above formulas, i.e. run CWB-QC only once. We first take the former way and perform the existential query checking with CWB-QC. Each run takes about 0.01 seconds and a maximal memory of 3 megabytes. For the latter way, CWB-QC takes a total of 0.02 seconds and a maximal memory of 4 megabytes. Here are the output formulas to query predicates:

$[\![?^+_{X_1}]\!]_T$: Switch = Off
$[\![?^+_{X_2}]\!]_T$: Switch = On & low $\leq$ temp $\leq$ high
$[\![?^+_{X_3}]\!]_T$: Switch = On & temp $<$ low
$[\![?^+_{X_4}]\!]_T$: Switch = On & temp $>$ high
$[\![?^+_{X_5}]\!]_T$: Thermostat=Inactive | Off
$[\![?^+_{X_6}]\!]_T$: Thermostat=Heat
$[\![?^+_{X_7}]\!]_T$: Switch=On
$[\![?^+_{X_8}]\!]_T$: Thermostat=Off | Inactive | Heat | AC

Note that the system we check is unbounded since *low*, *high* remain unspecified. One cannot query such a system with a finite-state query checking algorithm like the one [11], without using some abstraction techniques.

4.2 Performance Comparisons

Due to the absence of publicly available implementations of the query-checking tools, we compare the performance of our query checker with our model checker, and with the Action Language Verifier (ALV) [3] (version 0.3) for Presburger systems.

Comparison with model checking. The performance data are collected in Table 1. Besides the thermostat, we also check the cruise control system [2] for several invariant properties. Note that the performance of our model checker and existential query checker are virtually identical.

Comparison with ALV. ALV is a symbolic model checker for Presburger systems that which uses the Composite Symbolic Library (CSL) [21] as its symbolic manipulation engine. CSL combines different symbolic representations, which include the automata representation from the MONA package adopted by CWB-QC. To be fair for the performance comparison, we run ALV with the option "-F -I B" for forward and "-A -I B" for backward analysis respectively, and only automata representations are used.

Besides the above thermostat (all invariants are checked together), we have also verified the mutual exclusion properties for both bakery and ticket protocols, and an invariant property for the sleeping barber problem. The specifications for these systems are the same as [3]. The query formula we check for these examples is $\mathsf{AG}?_X$ and with projection over some data variables. Table 2 contains the performance data.

These figures in Table 1 and Table 2 show that the query checking runs as fast as the model checking of CWB-QC, and the latter is more efficient than ALV. This fact together with the running time from the previous subsection indicates that our proof-based symbolic query checking technique can provide very efficient service to the design of Presburger systems in practice.

Table 1: Performance comparison with model checking. The letters in the names of the systems refer to the indexes of properties; s : CPU time in seconds; k : maximum kilobytes of memory used by the verifier.

Example	CWB-QC query check	CWB-QC model check
thermostat-a	0.01s/3236k	0.01s/2812k
thermostat-b	0.01s/3316k	0.01s/2812k
thermostat-c	0.01s/3348k	0.01s/2808k
thermostat-d	0.01s/3304k	0.01s/2812k
cruise-a	0.02s/2932k	0.02s/2700k
cruise-b	0.02s/2528k	0.01s/2528k
cruise-c	0.01s/4088k	0.01s/3240k
cruise-d	0.02s/3940k	0.01s/3264k
cruise-e	0.01s/3508k	0.01s/3248k
cruise-f	0.02s/3084k	0.02s/2836k

4.3 Fast Error-Detection of CWB-QC

The (potentially) successful tableau constructed by CWB-QC provides a straight *witness* to show why the solution satisfies the query. On the other hand, a *counter-example* is reported when a formula is violated by the model. As an on-the-fly model checker, CWB-QC can detect errors very fast. We show this by checking buggy formulas for the above case studies.

The buggy formula we checked for the Thermostat is that

$$\mathsf{AG}(\text{Thermostat=AC} \rightarrow \text{temp} < \text{high}).$$

The property we verified for both ticket and bakery protocol is that whether it is allowed for a second process in the *try* mode while one is already in the *critical section*; The barber algorithm is checked with the negation of an invariant constraint. The performance data is reported in Table 3. CWB-QC generally outperforms ALV on these case studies. We conjecture that the superior performance in this case is due to the forward proof-based analysis of our technique.

5. CONCLUSION AND FUTURE WORK

We have proposed a framework for solving temporal-logic query checking for Presburger systems based on a tableau-based symbolic model-checking technique. Existential query checking returns one solution to the query predicate by locating one potentially successful tableau. While universal query checking returns multiple solutions from all such potentially successful tableaux. Our query checking works with multiple placeholders and placeholders with both positive and negative occurrences. Our experience with CWB-QC demonstrates that temporal-logic query checking for the class of (bounded) integer systems is practically feasible. Comparison results show that our query-checking technique is very efficient.

Table 2: Performance comparison with ALV-0.3. The numbers in the names of the systems refer to the numbers of processes in the models. s : CPU time in seconds; k : maximum kilobytes of memory used by the verifier; N/A : "UNABLE TO VERIFY" reported by the model checker (in this case we still report the time and memory consumption); O/M: computation does not terminate within one hour.

Example	CWB-QC query check	CWB-QC model check	ALV-0.3 forward	ALV-0.3 backward
thermostat	0.02s/3804k	0.02s/2468k	0.11s/16288k	0.10/16192k
ticket-2	0.02s/3076k	0.01s/2468k	0.08s/15288k	N/A(0.14s/15520k)
ticket-3	0.16s/3864k	0.11s/3368k	0.30s/15896k	N/A(0.64s/16364k)
ticket-4	1.22s/5224k	1.04s/4760k	2.98s/19300k	N/A(5.26s/26492k)
ticket-5	14.21s/12056k	14.00s/11468k	20.83s/33552k	N/A(30.24s/61584k)
bakery-2	0.01s/2832k	0.01s/2656k	0.11s/15576k	0.04s/15228k
bakery-3	0.51s/5496k	0.49s/3476k	14.40s/28368k	N/A(0.79s/17604k)
barber-10	0.11s/4972k	0.10s/4688k	0.15s/16636k	0.16s/16952k
barber-12	0.12s/5500k	0.12s/4972k	0.17s/16924k	0.18s/17136k
barber-14	0.15s/5896k	0.15s/5376k	0.19s/17156k	0.19s/17560k
barber-16	0.18s/6496k	0.17s/5376k	0.21s/17360k	0.21s/17748k

Table 3: Performance comparison with ALV-0.3 for buggy properties. The numbers in the names of the systems refer to the numbers of processes in the models. s : CPU time in seconds; k : maximum kilobytes of memory used by the verifier; O/M: computation does not terminate within one hour.

Example	CWB-QC model check	ALV-0.3 forward	ALV-0.3 backward
thermostat	0.00s/2784k	0.03s/15516k	0.02/16220k
ticket-2	0.01s/2848k	0.13s/15340k	0.06s/15360k
ticket-3	0.02s/2968k	2.64s/17644k	0.19s/16036k
ticket-4	0.03s/3432k	48.90s/45732k	1.45s/20796k
ticket-5	0.06s/4880k	820.84s/316988k	9.47s/37956k
bakery-2	0.00s/2468k	0.16s/15624k	0.07s/15116k
bakery-3	0.01s/2740k	13.79s/28504k	0.51s/17464k
barber-10	0.01s/2472k	O/M	0.17s/17024k
barber-12	0.01s/2704k	O/M	0.18s/17396k
barber-14	0.01s/3184k	O/M	0.21s/17410k
barber-16	0.01s/3272k	O/M	0.24s/17576k

The efficient query checking and model checking together with the fast-error-detection capability make CWB-QC extremely interesting for the understanding of system designs. We are planning to apply CWB-QC to some industrial case studies from several automotive and aerospace companies in the future.

6. ACKNOWLEDGMENTS

We would like to thank Tevfik Bultan and Constantinos Bartzis, for their great help related to the tool ALV.

7. REFERENCES

[1] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1), 1994.

[2] J. M. Atlee and M. A. Buckley. A logic-model semantics for scr software requirements. In *ISSTA'96*, pages 280–292, 1996.

[3] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *International Journal of Foundations of Computer Science (IJFCS)*, 14(4):605–624, Aug. 2003.

[4] G. S. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for ctl*. In *Proceedings of the 10th Annual Symposium on Logic in Computer Science (LICS '95)*, pages 388–397, San Diego, July 1995. IEEE Computer Society Press.

[5] G. Bruns and P. Godefroid. Temporal logic query-checking. In *Proc. of 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 409–417, Boston, MA, USA, June 2001. IEEE Computer Society.

[6] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In O. Grumberg, editor, *Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

[7] W. Chan. Temporal-logic queries. In *Proceedings of the12th Conference on Computer Aided Verification (CAV'00), LNCS 1855*, pages 450–463. Springer-Verlag, July 2000.

[8] E. M. Clarke and E. A. Emerson. Design and synthesis

of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs, Yorktown Heights*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[9] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[10] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98), Vol. 1427 of Lecture Notes in Computer Science*, pages 268–279. Springer-Verlag, 1998.

[11] A. Gurfinkel, B. Devereux, and M. Chechik. Model exploration with temporal logic query checking. In *Proc of SIGSOFT Conference on Foundations of Software Engineering (FSE'02)*, pages 139–148, Nov. 2002.

[12] C. Heitmeyer, R. Jeffords, and B. Lawbaw. Automated consistency checking of requirements specification. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[13] S. Hornus and P. Schnoebelen. On solving temporal logic queries. In *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France*, volume 2422 of *Lecture Notes in Computer Science*. Springer, 2002.

[14] J.Bradfield and C.Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96:157–174, 1992.

[15] N. Klarlund and A. Moller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, Jan. 2001.

[16] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, Dec. 1983.

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

[18] M. Samer and H. Veith. Validity of ctl queries revisited. In *CSL'03, LNCS 2803*, pages 470–483, 2003.

[19] L. Tan and R. Cleaveland. Evidence-based model checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.

[20] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Math.*, 1955.

[21] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *TACAS 2001, LNCS 2031*, pages 52–66, Apr. 2001.

[22] D. Zhang and R. Cleaveland. Fast generic model-checking for data-based systems. In F. Wang, editor, *Proceedings of the 25th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, volume 3731 of *Lecture Notes in Computer Science*, Taiwan, Oct. 2005. Springer-Verlag.