Requirements Engineering manuscript No.
(will be inserted by the editor)

Toward Automation for Model Checking Requirements Specifications with Numeric Constraints*

Yunja Choi, Sanjai Rayadurgam, Mats P.E. Heimdahl

Department of Computer Science and Engineering, University of Minnesota
200 Union Street S.E., 4-192, Minneapolis, MN 55455, USA
e-mail: {yuchoi, rsanjai, heimdahl}@cs.umn.edu

The date of receipt and acceptance will be inserted by the editor

Abstract Model checking techniques have not been effective in important classes of software systems—systems characterized by large (or infinite) input domains with interrelated linear and non-linear constraints over the system variables. Various model abstraction techniques have been proposed to address this problem, but their effectiveness in practice is limited by two factors; first, the abstraction process is manual and requires a great deal of ingenuity, and, second, the abstraction may be coarse and introduce too many spurious behaviors to provide meaningful analysis results.

In this paper, we wish to propose *domain reduction abstraction* based on *data equivalence* and *trajectory reduction* as an alternative and complement to other abstraction techniques. Our technique applies the abstraction to the input domain (environment) instead of the model and is applicable to *constraint-free* and deterministic *constrained* data transition systems. Our technique is automatable with some minor restrictions. We provide formal proofs for the theoretical soundness of the technique, algorithms for automation, and an illustration of the approach with examples.

1 Introduction

Model checking (both explicit state and symbolic) has been successfully used to verify properties of various finite state systems, such as communications protocols [6,31], concurrent systems [8], hardware designs [9], and software requirement specifications [3,11,27]. Nevertheless, the usefulness in verifying properties of software systems has been limited since important classes of software systems involve large (or infinite) input domains (e.g., unbounded integer variables) as well as interrelated numeric constraints over the variables in the input domain—characteristics that severely limit the usefulness of model checking.

To address this problem, various techniques for *abstracting the model* to a simplified model that can be model checked is an active research area [8,16,19,20,22]. The major goal of the abstraction is to provide (1) a sound abstraction technique in a sense that the technique reduces the size of the system while preserving interesting properties, (2) an automated abstraction process that requires minimum user interaction to reduce human mistakes during the abstraction process, and (3) as tight an abstraction as possible to remove spurious counter examples.

To achieve these objectives, automation of the abstraction and precise abstractions with well understood properties are necessities. Here we propose *domain reduction abstraction* based on *data equivalence* and *trajectory reduction* as an alternative and complement to other abstraction techniques. While most current automated abstraction techniques are based on on-the-fly abstraction using iterative refinement or symbolic approximation [2,4,8,16,24], our technique is based on static analysis that can be applied on top of any existing model checking tools.

Important classes of software systems can be viewed as consisting of a finite *control component* and a (typically infinite or very large) *data component*; examples include many safety-critical control systems. In such systems, the variables in the data component represent the input quantities to the system. For example, in an avionics system, inputs may be altitude, range, bearing and various pilot selectable thresholds. The transitions in the system may be guarded by various linear and non-linear constraints on these variables. Furthermore, these systems interact with some environment and this environment is subject to various constraints that may need to be captured. In the avionics system example, the change of altitude is constrained by the aircraft's altitude rate as well as limited by aircraft characteristics. These types of constraints must often be considered in verification to produce meaningful results. Model checking systems with such constraints often requires exhaustive search over large data domains, which causes state-space explosion.

To address these problems, various techniques to abstract the model to a finite domain have been proposed, such as predicate abstraction [22] and symbolic approximation [2,

* This work has been partially supported by NASA grant NAG-1-224 and NASA contract NCC-01-001. An early version of this paper appeared as "Automatic Abstraction for Model Checking Software Systems with Interrelated Numeric Constraints", in the Proceedings of FSE/ESEC 2001.

7]. Other researchers have attempted to combine constraint solving with model checking [10]. These approaches have achieved some success and are discussed in Section 2.

In this work, we have investigated abstractions over the *input domain* of the systems rather than the system itself. For systems with no data constraints the abstraction is based on a data equivalence relation using the transition conditions on the input variables to partition the infinite input domain into a finite set of partitions from which one representative input is selected. The abstraction guarantees bi-simulation equivalence between the original and the abstract systems. In systems with data constraints, *trajectory reduction* maps a possibly infinite set of input variable trajectories through the state space to a single representative trajectory in a finite domain. This reduced domain can be computed by tracing minimal data trajectories over data equivalence classes defined by the numeric transition conditions, and taking into account the data constraints imposed on the input variables. Given certain constraints, trajectory reduction produces an abstract system that simulates the original systems.

In the next section we provide an overview of related work. After illustrating the motivation for our work with a simple example in Section 3 and introducing the underlying system model in Section 4, we formally describe the domain reduction technique in Section 5, and suggest an algorithm for automation in Section 6. A second application example (Section 7) and conclusions (Section 8) follow.

2 Related Work

State space explosion is a serious problem when verifying systems using model-checkers. This problem is especially pronounced in the case of software system specifications, which often have large or infinite data domains. To deal effectively with this problem, one should be able to derive meaningful conclusions about the reachable states of the system without actually exploring those states individually. There has been extensive work in addressing state space explosion problem in model-checkers [1,8,19–21,23,32,35,39], and many techniques such as, symbolic model-checking, partial order reduction, symmetry, induction, abstraction, and compositional reasoning are being successfully used. See [17] for a comprehensive treatment of the subject. Here, we discuss a few specific works that are closely related to ours.

Heitmeyer *et al.*[27] describe automatic abstraction techniques for specifications written in SCR. Given a property to verify, they describe an abstraction method to automatically *remove irrelevant data variables* and *remove detailed monitored variables*. The former is similar to slicing [25,26] and the cone of influence abstraction [17], where dependency information from the model is used to remove parts that do not have an effect on the property of interest. In our proposed method, irrelevant data variables would get replaced with a single representative value, effectively removing it. The latter is a data abstraction technique which is achieved in our method by choosing representative elements from the data equivalence classes of the variable.

Chan *et al.* [10] describe a technique for model-checking certain classes of systems in which data constraints across transitions can be separated as pre-state and post-state conditions connected by boolean operators. Such systems can be model-checked by representing the conditions as boolean variables and constraints as values of the variables before and after the transition. During model-checking, a constraint solver is used to eliminate infeasible combinations of conditions. This technique can handle systems with both linear and non-linear numeric conditions, but is limited to data-memoryless or data invariant systems which correspond to our notion of constraint-free systems described later. Our proposed method, when applied to such systems, would statically find representative values in the domain of the data variables for each feasible combination of the conditions before model-checking the system.

A promising approach to make model-checking feasible for software systems is predicate abstraction [22]. The states of a concrete system are mapped to states of an abstract system according to their evaluation over a finite set of predicates. Assignments to variables in the concrete system are replaced by assignments to the appropriate boolean variables in the abstract system. For example, consider a transition of the form $[a \longrightarrow b$ when $y > 50]$ with the data constraint $y' = y+1$. Now, $y > 50$ can be represented as an abstract boolean predicate p. $y' = y + 1$ would then be transformed to $p' = true$ if $(p = true)$ and $p' =$ one of $true$ or $false$ if $(p = false)$. We now have an abstract system using the boolean predicate p instead of the concrete system with the integer variable y. The abstract system is then model-checked to verify properties. Typically, the abstraction is conservative in the sense that if the property holds for the abstract system it also holds for the concrete system. However, the property may fail for the abstract system due to over-approximation. In that case, the abstract model is then refined so that the spurious error is removed and the verification cycle is repeated. Typically, the use of special-purpose tools such as decision procedures or theorem-provers will be required to guide in the search for predicates in the refinement process. Recent works [4,30,34,38], in the area of model-checking programs have developed approaches to automate predicate abstraction and address the refinement problem.

There are other approaches to model checking infinite state systems with data transition constraints in the domain of hybrid systems and concurrent systems [2,7]. These techniques are also based on on-the-fly symbolic approximation techniques and require special-purpose tools (e.g., [29]), in the model-checking process.

An alternative approach would be to retain the data-constraints as they are, but *bound* the domain of the data variables to make model-checking feasible. The work is performed upfront in reducing the domain of system variables using constraint solving techniques before model-checking the system, rather than using special-purpose tools for solving constraints on-the-fly during the verification process. Thus we apply abstraction techniques statically and use a conventional model-checking tool like SMV [33,14] to verify properties in the abstract system. One must be careful not to remove existing behavior in the process of bounding the domains, for then the resulting system may satisfy some properties that were not true in the original system (e.g., restricting y to $[0..10]$, in the earlier example would result in $(y < 50)$ becoming true of all states). Such under-approximations could be useful when the goal is to show the existence of an error. In such a case, one could presumably retain a subset of the behavior of the original system and yet successfully exhibit the presence of errors.

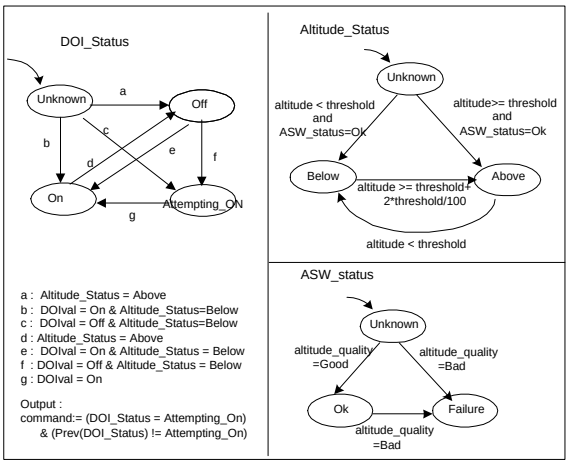

Fig. 1 The ASW system

In the context of verification, the abstraction must ensure that the data that is excluded by bounding the domain does not allow any unique behavior that is not captured by the data included within the bound. We pursue this alternative approach by suggesting such a method for bounding data domains.

3 Motivation and Approach

In our work we are translating state based requirement models to the input language of a model checker for verification purposes. For the types of systems in which we are interested (critical control applications) we inevitably encounter very large or infinite state spaces. Even the simplest system, such as the one described below, cannot be model checked without abstraction.

Our example is drawn from the avionics domain: a modified version of the Altitude Switch (ASW) [37]. The ASW is a hypothetical device that turns on another subsystem (the Device-Of-Interest – DOI), whenever the aircraft descends below a threshold altitude and turns the power back off again when the aircraft ascends above the threshold (plus a hysteresis factor). The hysteresis factor is defined in terms of the threshold – the ASW turns off the DOI when the aircraft ascends above $threshold + \frac{threshold}{50}$. Other systems in the cockpit may independently turn the DOI off and on. The input variable *DOIval* indicates the status of the DOI. There are no initial values or constraints on the change of data values in the system–we call this type of system as a *constraint-free system* (this notion will be discussed in detail in Section 4 and Section 5.1). Figure 1 shows the ASW state transition diagram.

In this setting we may want to check properties such as "the *On* command will not be issued if the *DOI* is currently *Off* and the altitude is above the threshold". Model checking properties such as this without abstraction is infeasible because of the large ranges of numeric variables in this system; *altitude: 0..40,000* and *threshold: 2,000..35,000*.

A simple predicate abstraction, may be applied to abstract out the large integer variables. The predicate abstraction replaces each numeric condition with a Boolean variable as follows:

1. $altitude \geq threshold \longrightarrow a$
2. $altitude < threshold \longrightarrow b$
3. $altitude \geq threshold + threshold * \frac{1}{50} \longrightarrow c$

In this case, the model checker would not recognize that $b \wedge c$ is always false, and thus introduce additional transitions in the abstract system by allowing $b \wedge c$ — a more precise abstraction is necessary.

More interesting problems arise when we want to check a property such as "the ASW shall never turn the DOI on while ascending". In the system described above, the altitude is unconstrained and may change to any value at any time (therefore, we have no concept of the aircraft climbing). In addition, the threshold may be changed at any time so that in one step the aircraft is above the threshold and in the next it is below even if the altitude has not changed. In actuality, the environment of the ASW is constrained—the threshold is always *fixed* before flight and constant thereafter, and the altitude can only change so much between two steps. This gives us a data constrained system where *threshold' = threshold*, and the altitude can be constrained to be in an ascent, for example, *altitude' = altitude + 10*. We call such a system a *constrained data transition system*—specifically, in this case we have a globally constrained data transition system (these types of systems will be discussed in detail in Section 4 and Section 5.2).

Several techniques are available for abstracting such a *constrained data transition system*, such as symbolic approximation [2,7] and iterative predicate abstraction with refinement [4,30]. As mentioned earlier, these are on-the-fly symbolic abstraction methods that either have non-termination problem or require user intervention in the abstraction process.

While incorporation of abstraction techniques into model checking tools is an active research area, a practical and reliable tool for model checking systems of our interest is not available at this point. We decided to investigate a static abstraction technique that can be applied independent of any specific model checker so that we can use existing reliable tools without building another special-purpose model checker.

3.1 Overview of Our Approach

To give the reader a general understanding of our approach we provide an informal outline of the abstraction techniques below. Formal treatment of the topic follows in the subsequent sections.

We tackle the problem of numeric variables with two complementary abstractions; first, when data-constraints are not present, a simple data abstraction technique based on a data equivalence relation will be used. If the system is data-constrained, one data trajectory, which is a series of data values satisfying all data-constraints, will be computed and used as a representative for all data trajectories with the same characteristics. Here, the characteristics of a data trajectory is determined by the ordered set of data-equivalence classes the data trajectory passes through.

Figure 2 shows an intuitive view of domain reduction abstraction. We partition the domain of numeric variables by the valuation of the numeric conditions which can be extracted from the transition conditions and verification properties. For example, the partition A represents the region that satisfies $altitude < threshold \wedge altitude < threshold + \frac{threshold}{50}$.

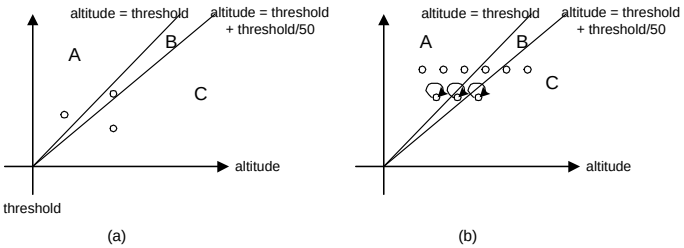

Fig. 2 General abstraction approach

We replace the set of possible values in a data-equivalence class with a randomly selected representative from the class (see (a) of Figure 2). This is a variation of the data abstraction technique suggested by Clarke et al. [15]; instead of mapping each partition class to a symbolic enumerated value, we simply select a representative from each class effectively removing the mapping process. The idea behind this technique is the same as that of partition testing [5].

The major benefits of this approach are (1) it provides an exact abstraction, i.e., sound and complete abstraction for a given property of the original system, and (2) the computation of the abstraction is done before model checking so that we do not need the refinement process required in dynamic abstraction techniques to remove spurious counter examples.

When data-constraints must be taken into account, such as the constraint $altitude' = altitude + 10$, random representatives cannot be selected since they are likely to violate the data constraints. Nevertheless, we can refine the abstraction by computing a minimal data trajectory that we use as a representative for all data trajectories passing through the same set of data-equivalence classes in the same order. For example, all data-trajectories passing through the equivalence classes A, B, C (in that order) can be simulated by one minimal trajectory (see (b) of figure 2). Simulation of the original system by the abstract system is ensured by introducing data stuttering so that the minimal trajectory can always be as long as any other trajectory. This approach provides us a conservative abstraction of the original system such that all the behaviors (transitions) of the original system are included in the abstract system. This abstraction enables us to verify a certain class of interesting properties, such as "the ASW shall never turn the DOI on while ascending" mentioned previously.

The next two sections provide a formal characterization of our systems of interest and provide all necessary definitions (Section 4), as well as the formal foundation for our abstraction technique (Section 5).

4 System Model and Definitions

To be able to formally define our abstraction techniques, we must first define our system model, and a collection of various terms and formalisms that are used in the remainder of the paper.

4.1 System Model

Our system model is a tuple $(N, N_0, v, D, D_0, \Delta, C, \Psi)$ where:

- N is a finite set of control nodes;
- N_0 is a subset of N containing initial control nodes;
- v is a finite vector $[v_1, \ldots, v_n]$ of data variables;
- D is the domain of v obtained as a cross-product of the domains of the components v_i as $D_1 \times \ldots \times D_n$;
- D_0 is a subset of D describing the initial values of data variables;
- C is a finite set of conditions on data variables of the form $c(v) \bowtie 0$ where $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$ and $c : D \longrightarrow \Re$;
- Ψ is a finite set of data constraints of the form $\psi(v, v') \bowtie 0$ where v' is a vector of data variables similar to v and $\psi : D \times D \longrightarrow \Re$; and,
- Δ is a relation between $N \times N$ and $b(C) \times b(C) \times b(\Psi)$, where $b(C)$ and $b(\Psi)$ are sets of finite boolean combinations of data-conditions C and data-constraints Ψ, respectively.

Data conditions in C define data regions of the state-space in terms of numeric inequalities while the data constraints in Ψ capture joint constraints involving data values in the pre-state and post-state. Informally, the relation Δ means that if $((n, m), (C_1, C_2, C_3)) \in \Delta$, the two control nodes and conditions are related in a way that there are some data values x_n for n, x_m for m such that x_n satisfies conditions in C_1, x_m satisfies conditions in C_2, and (x_n, x_m) satisfies the data constraints in C_3.

This system model can be used to represent a wide range of reactive systems. For our purposes, we require some restrictions on the type of data-constraints ψ. These restrictions are presented later in this section.

This system model can be considered to be a basic transition system $M = (S, S_0, R, L, AP)$, where:

- $S = N \times D$; every state has two components, a control node and a data node.
- $S_0 = N_0 \times D_0$; the initial states are given by the initial values for the control and data parts.
- $R((m, x), (n, y)) \iff \exists \alpha, \beta \in b(C), \exists \gamma \in b(\Psi) : \Delta((m, n), (\alpha, \beta, \gamma)) \wedge \alpha(x) \wedge \beta(y) \wedge \gamma(x, y)$; i.e., there is a transition between two states exactly when the system model has a transition between the corresponding control and data parts as defined by Δ.
- $AP = N \cup C$; i.e., propositions are control nodes and data conditions.
- $L(n, x) = \{n\} \cup \{\alpha \in C \mid \alpha(x)\}$; i.e., states are labeled by their control nodes and satisfied data conditions.

In this paper, when we refer to a basic transition system, it is assumed that there is an underlying system model, which is viewed as the basic transition system. For notational convenience we write $(s, t) \in R$ as $R(s, t)$ and call s and t the pre-state and post-state respectively.

We define some common notations below that we use in this paper.

1. $s|_N = n$, $s|_D = d$, when $s = (n, d) \in S$.
2. For $s \in S, R(s) = \{t \in S \mid R(s, t)\}$, for $A \subseteq S, R(A) = \bigcup_{s \in A} R(s)$, and $R(s)|_D = \{t|_D \mid t \in R(s)\}$.
3. $R^1(s) = R(s)$ and $R^{n+1}(s) = R(R^n(s))$ for $n \geq 1$.

$s|_N$, $s|_D$ denote respectively, the control and data components of state s. $R(s)$ denotes a set of post-states of s and $R(s)|_D$ denotes the projection $R(s)$ onto the data component. We would use D_i instead of D, if the projection to the domain of i^{th} data variable is required.

4.2 Basic Definitions

The following basic definitions, starting from a formal definition of a system path, will be used throughout this paper.

Definition 1 *A **path** in the structure M from a state s is an infinite sequence of states*[1] *$\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$. $\pi(s_0)$ denotes a path with initial state s_0.*

Definition 2 *Two data points $x, y \in D$ are **data equivalent**, written $x \equiv y$, if $\forall c \in C : c(x) = c(y)$.*

$D/_{\equiv}$ denotes the set of equivalence classes induced by $\equiv$ on D, e_i denotes the i^{th} data equivalence class, and $rep(D/_{\equiv})$ denotes a set of representatives from D, one per equivalence class e_i. Intuitively, x is $data\ equivalent$ to y if and only if x and y have the same valuations (i.e., same truth values) for all data conditions. A data equivalence class e_i is a set of data values that have same valuations for all data conditions.

Definition 3 *Two states $s, s' \in S$ are **state equivalent**, written $s \simeq s'$, if $L(s) = L(s')$, i.e., $s|_N = s'|_N \ \wedge \ s|_D \equiv s'|_D$.*

Two states are state equivalent when they have the same control node and their data nodes have same valuations for all data conditions. Note that two states are equivalent if and only if they have the same labels according to our system model. Data nodes from two equivalent states are data equivalent, but the reverse case is not necessarily true. $S/_{\simeq}$ denotes the set of equivalence classes induced by $\simeq$ on S and E_i denotes the i^{th} state equivalence class. A state equivalence class E_i is a set of states that are state equivalent to each other, i.e., a set of states with the same label.

In the case of the ASW example described in the previous section, we would have three data equivalence classes A, B, C as follows.

$$\begin{aligned}
A = \{&(altitude, threshold) \mid altitude \in [0, 40000], \\
&threshold \in [2000, 35000],\ altitude < threshold \\
&\wedge altitude < threshold + \frac{threshold}{50}\} \\
B = \{&(altitude, threshold) \mid altitude \in [0, 40000], \\
&threshold \in [2000, 35000],\ altitude >= threshold \\
&\wedge altitude < threshold + \frac{threshold}{50}\} \\
C = \{&(altitude, threshold) \mid altitude \in [0, 40000], \\
&threshold \in [2000, 35000],\ altitude >= threshold \\
&\wedge altitude >= threshold + \frac{threshold}{50}\}
\end{aligned}$$

On the other hand, the state equivalence classes take into account the valuation of control nodes as well as the valuation of the data conditions. The control variables of the ASW system are $Altitude\_Status$, ASW_status, $altitude\_quality$, $DOIval$, and DOI_Status and the state equivalence classes would be as follows.

$$\begin{aligned}
E_1 &= \{(n_1, altitude, threshold) \mid (altitude, threshold) \in e_1\} \\
E_2 &= \{(n_2, altitude, threshold) \mid (altitude, threshold) \in e_1\} \\
E_3 &= \{(n_3, altitude, threshold) \mid (altitude, threshold) \in e_2\} \\
E_4 &= \{(n_3, altitude, threshold) \mid (altitude, threshold) \in e_3\} \\
E_5 &= \{(n_4, altitude, threshold) \mid (altitude, threshold) \in e_2\} \\
E_6 &= \{(n_4, altitude, threshold) \mid (altitude, threshold) \in e_3\} \\
&\vdots = \vdots
\end{aligned}$$

where, the control nodes are given by,

$$\begin{aligned}
n_1 &= \{\textit{Below, Ok, Good, On, On}\} \\
n_2 &= \{\textit{Below, Ok, Good, Off, Attempting_On}\} \\
n_3 &= \{\textit{Above, Ok, Good, On, Off}\} \\
n_4 &= \{\textit{Above, Ok, Good, Off, Off}\} \\
&\vdots = \vdots
\end{aligned}$$

We classify transition systems into two categories: *constraint free data transition systems*, and *constrained data transition systems*.

Definition 4 *A transition system M is a **constraint-free data transition system** if $R(s, t)$ for some s, t implies $R(s', t')$ for all $s' \simeq s$, $t' \simeq t$.*

In a *constraint-free data transition system*, existence of a transition between two states (s and t) implies the existence of a transition between any pair of states (s' and t') from their corresponding state-equivalence classes. In other words, the systems does not impose any constraints on how data variables change value.

When a system is not a constraint-free data transition system, the transition relation constrains how the data variables may change values. In general, the data constraints that determine how the data variables may change value can be an arbitrary relation. In the work presented here we consider only relations that are functions over the data domain and that satisfy certain restrictions. These restrictions are formally defined in the following paragraphs.

Definition 5 *R is said to be a **constrained transition** for D_i if for each state-equivalence class E_j, there is a unique function $f_{E_j} : D_i \longrightarrow D_i$ such that, $t|_{D_i} = f_{E_j}(s|_{D_i})$, whenever $s \in E_j$ and $R(s, t)$ holds for states $s, t \in S$.*

A constrained data transition means that the transition relation imposes constraints on the specific data values of the pre-state and the post-state. The type of constraints we consider here are functions—i.e., the data in the post-state is an application of a function to the data in the pre-state. The specific function to be applied may be dependent on the label of the pre-state. Thus, whether or not a transition exists between two states is determined by the data transition function as well as the state labels.

When the transition functions are the same for all state equivalence classes, i.e., $f_{E_j} = f_{E_k}$ for all $E_j, E_k \in S/_{\simeq}$, we say that the data transition is a *global data transition*

[1] R is assumed to be a total relation.

for D_i. Otherwise, we call it a *local data transition* for D_i. Global data transitions are defined uniformly over D_i while local data transitions are defined depending on each state label. Typically, global data transitions are used for imposing environmental constraints, such as physical laws, on data variables. Local data transitions are typically used for data maintained by the system such as counters and flags.

A data variable v_i is said to be *globally constrained* if R is a global transition for D_i, *locally constrained* if R is a local transition for D_i, and *unconstrained* if there is no data transition constraint for D_i (i.e., the variable may change values randomly). Note that if v_i is unconstrained for every i, then the system is a constraint-free data transition system. If we have one or more of global or local transitions, we call it a constrained data transition system. In general, there may be constraints which do not fall into any of the three categories. Such systems are not considered here.

Definition 6 *A transition system M is a **constrained data transition system** if R is a local or a global data transition for some D_i.*

Note that there is an overlap between constraint-free and constrained data transition systems since systems with only one state per state equivalence class satisfy both categories.

Given a system model as defined above, the control node transition is deterministic on state labels and the data transition constraint. In other words, given a valid path $\pi = s_0 s_1 \dots$ (refer to Definition 1), any sequence of states $[s'_0, s'_1, \dots]$ that are state-equivalent to the corresponding states in π (i.e., $\forall i,\ s_i \simeq s'_i$) is also a valid path provided that the data nodes in the sequence satisfy the same data transition constraints as those in the path π. The following lemma can be inferred from the definition of our system model.

Lemma 1 *Transition relations between control nodes are deterministic on labels and data transition constraints; i.e., $R(s,t) \wedge s \simeq s' \wedge t \simeq t' \wedge t'|_D \in R(s')|_D$ implies $R(s',t')$.*

Proof The proof is obvious for constraint-free data transition systems. Suppose that R is constrained for D_i and unconstrained for D_j for all $j \neq i$. Then there exists a unique function f_i for the state equivalence class E where $s \in E$ and $t|_{D_i} = f_i(s|_{D_i})$. Since $t'|_D \in R(s')|_D$ and $s' \in E$ from $s' \simeq s$, we can infer that $t'|_{D_i} = f_i(s'|_{D_i})$. From the definition of R, $\exists \alpha, \beta \in b(C)$, $\exists \gamma \in b(\Psi)$ such that $\Delta((s|_N, t|_N, (\alpha,\beta,\gamma)) \wedge \alpha(s|_D) \wedge \beta(t|_D) \wedge \gamma(s|_D, t|_D)$. Note that $\gamma = f_i$ in a sense that only i_{th} data variable is constrained, i.e., $\gamma(x,y)$ is true if and only if $y|_{D_i} = f_i(x|_{D_i})$. The fact that $s \simeq s'$, $t \simeq t'$, and $t'|_{D_i} = f_i(s'|_{D_i})$ deduces $\Delta((s'|_N, t'|_N, (\alpha,\beta,\gamma)) \wedge \alpha(s'|_D) \wedge \beta(t'|_D) \wedge \gamma(s'|_D, t'|_D)$, and thus, $R(s',t')$. Cases with more than one constrained variables can be proved in a similar way. □

4.3 Bisimulation and Simulation Relations

The notion of bisimulation/simulation relation is used to justify abstraction techniques that replace a large system structure by a smaller one which satisfies the same properties. We use definitions of bisimulation and simulation relations borrowed from [17] for formal proofs presented in the next section.

Definition 7 *Let $M = (S, S_0, R, L, AP)$ and $M' = (S', S'_0, R', L', AP)$ be two structures with the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a **bisimulation relation** between M and M' if and only if for all s and s', if $B(s,s')$ then following conditions hold:*

1. *$L(s) = L'(s')$.*
2. *For every state s_1 such that $R(s,s_1)$ there is s'_1 such that $R'(s',s'_1)$ and $B(s_1,s'_1)$.*
3. *For every state s'_1 such that $R'(s',s'_1)$ there is s_1 such that $R(s,s_1)$ and $B(s_1,s'_1)$.*

Definition 8 *The structures M and M' are **bisimulation equivalent** if there exists a bisimulation relation B such that for every initial state $s_0 \in S_0$ in M there is an initial state $s'_0 \in S'_0$ in M' such that $B(s_0,s'_0)$. In addition, for every initial state $s'_0 \in S'_0$ in M' there is an initial state $s_0 \in S_0$ in M such that $B(s_0,s'_0)$.*

Bisimulation equivalence guarantees that M and M' satisfy the same set of CTL^*[2] formulas constructed using the atomic propositions in AP. However, bisimulation is a strong requirement and for many systems it may not yield a significant reduction in the state-space. In order to achieve a greater reduction, the notion of a *simulation relation* is introduced.

Definition 9 *Given two structures M and M' with $AP' \subseteq AP$, a relation $H \subseteq S \times S'$ is a **simulation relation** between M and M' if and only if for all s and s', if $H(s,s')$ then the following conditions hold.*

1. *$L(s) \cap AP' = L'(s')$.*
2. *For every state s_1 such that $R(s,s_1)$, there is a state s'_1 such that $R'(s',s'_1)$ and $H(s_1,s'_1)$.*

Definition 10 *M' **simulates** M if there exists a simulation relation H such that for every initial state s_0 in M there is an initial state s'_0 in M' such that $H(s_0,s'_0)$.*

When M' simulates a structure M, every behavior of M is also a behavior of M'. However, the abstracted structure M' may have behaviors that are not possible in the original structure M. It is shown that if M' simulates M then every $ACTL^*$[3] formula satisfied by M' is also satisfied by M [17].

5 Abstraction Theory

For clarity of presentation, we present the formal foundation for our abstraction technique in this section. The application of the technique on an example can be found in Section 6. Section 6 also contains a detailed discussion of how the abstraction presented in this section can be automated.

5.1 Constraint-Free Systems

When the system does not have data constraints, it is bisimular to a structure that has one representative from each equivalence class. This is formally established here.

[2] CTL^* (Computational Tree Logic) is a temporal logic widely used to specify temporal properties in model checking.

[3] The restriction of CTL^* to universal path quantifiers.

For a given constraint-free data transition system $M = (S, S_0, R, L, AP)$, let $D' = rep(D/_{\equiv})$ and $M' = (S', S'_0, R', L', AP)$ where $S' = N \times D'$, $R' = R \cap (S' \times S')$, $S'_0 = S_0 \cap S'$[4] and $L' = S' \lhd$ [5] L.

Theorem 1 *The state equivalence relation $\simeq$ is a bisimulation relation between M and M'.*

Proof Suppose $s \simeq s'$ where $s \in S, s' \in S'$.

1. $L(s) = L(s') = L'(s')$ since s and s' have the same control label and the same valuation for all data conditions.
2. Suppose $R(s,t)$ holds for some $t \in S$. Let e_i be the data-equivalence class of t, i.e., $t|_D \in e_i$. By the definition of D', there is some $r_i \in D'$ such that $r_i \in e_i$. Let $t' = (t|_N, r_i)$. $t \simeq t'$ is obvious. Since M is a constraint-free data transition system, by definition, $R(s,t) \wedge s \simeq s' \wedge t \simeq t'$ implies $R(s',t')$. Therefore $R'(s',t')$ holds since $R' = R \cap (S' \times S')$ and $s', t' \in S'$.
3. Suppose $R'(s',t')$ holds for some $t' \in S'$. By the definition of R', $R(s',t')$ holds in M where $t' \in S' \subseteq S$. Let $t = t'$. From the fact that M is a constraint-free data transition system, $R(s',t') \wedge s' \simeq s \wedge t' \simeq t$ implies $R(s,t)$. □

Theorem 2 *The structures M and M' are bisimulation equivalent.*

Proof Given an initial state $s_0 \in S_0$, let $e_i \in D/_{\equiv}$ be the equivalence class to which s_0 belongs. By definition, D' contains a representative r_i of the class e_i. Let $s'_0 = (s_0|_N, r_i)$. Then, $s'_0 \in S'$ and $s_0 \simeq s'_0$. The converse is trivial. □

As a result, we can reduce the number of states of the abstracted system by choosing one representative from each data equivalence class without affecting the system behavior. The resulting system M' is an exact abstraction of the concrete system M. In other words, the resulting abstract system M' is a sound and complete abstraction of the original system M; any property satisfied by M is also satisfied by M' and vice versa since both systems satisfy the same set of formula.

The idea behind this abstraction technique is similar to that of partition testing. When the input data space can be partitioned in a way that the system behavior is determined completely by identifying the partition to which a given input data belongs irrespective of the actual data values, one could completely test the system by selecting one representative from each input partition [5].

5.2 Constrained Systems

When a system has constrained data variables, domain reduction based on state equivalence can still produce a conservative abstraction[6], but it is not sufficient to preserve path properties in the original system. A more refined abstraction is required to preserve interesting path properties while at the same time reducing the size of the data domain.

To get such a refined data abstraction for constrained data systems, we define a reduced domain D' such that it includes every possible data path in terms of data equivalence. The approach can be conceptually viewed as follows: For each path in M, find the sequence of data equivalence classes as given by the states in the path in order. Find a *minimal path* (refer to definition 14) in M that passes through the same sequence of data equivalence classes. The reduced domain D' contains all the data values in the *minimal path*. As we will show later, with some restrictions and by introducing data stuttering, the abstract system obtained by such a domain abstraction simulates the original system.

For simplicity, the constrained data transition systems that we consider here are assumed to satisfy the following assumption. [7]

Assumption 1 *Constrained and unconstrained data are exclusive in a system, i.e., a data condition cannot refer to both unconstrained and constrained data variables.*

The assumption is to separate out non-determinism in the transition system. Based on the assumption, we can separate numeric conditions with constrained data variables from numeric conditions with unconstrained data variables. For data equivalence classes partitioned by numeric conditions with unconstrained data variables, applying the abstraction technique described in the previous section suffices. Thus in this section, without loss of generality, we assume that all data variables are constrained.

5.2.1 Definitions As mentioned above, our abstraction technique is based on the selection of representative data trajectories from sets of equivalent data trajectories—here, equivalence is defined in terms of the sequence of data equivalence classes the trajectory passes. The notion of *trajectory reduction* is the key concept used to reduce the state space. To define trajectory reduction we first need to introduce some auxiliary concepts.

Definition 11 *A **trace** $T[\pi(s_0)]$ of a path $\pi(s_0)$ is a sequence of data equivalence classes $e_0 e_1 e_2 \ldots$ with $e_i \neq e_{i+1}$ for all i such that $s_0|_D \in e_0$ and for all $i, j \geq 0$, $s_i|_D \in e_j$ implies $s_{i+1}|_D \in e_j$ or $s_{i+1}|_D \in e_{j+1}$.*

In other words, a trace $T[\pi(s_0)]$ is a permutation of a subset of data equivalence classes of D that are reachable from s_0 in order. We also define a trace as a *maximal trace* when it is not a subsequence of any other traces. A trace defines the characteristics of a path and *trajectory reduction* will be defined on the set of paths with the same trace.

Definition 12 *A **node of change** in a path $\pi(s_0)$ is either s_0 or a state s_i such that $s_i|_D \in e_j$ and $s_{i-1}|_D \notin e_j$ for some data equivalence class e_j.*

Nodes of change are defined with respect to a given path. Nodes of change are states along the path at which there is a change in data equivalence class. The notion of *node of change* is needed to compare the lengths of the segments of two paths that lie in the same data equivalence class.

[4] It is assumed that $D_0 = D$ in constraint-free systems.

[5] $\lhd$ is the notation for domain restriction in Z. $S \lhd R$ of a relation R to a set S relates x to y iff R relates x to y and x is a member of S [36].

[6] We say that abstraction is conservative if the abstract system retains every transition of the original system.

[7] This assumption can be removed with a minor change to Theorem 3 to make $R(s'_{i-1})|_D$ deterministic. Details are out of the scope of this paper.

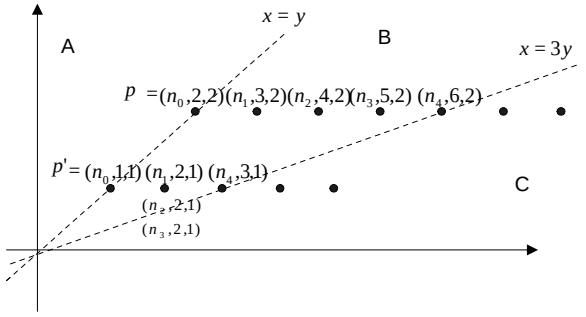

Fig. 3 Same trace with different segment length sequence

π	$(n_0,2,2),(n_1,3,2),(n_2,4,2),(n_3,5,2),(n_4,6,2),\dots$
π'	$(n_0,1,1),(n_1,2,1),(n_2,2,1),(n_3,2,1),(n_4,3,1),\dots$

Table 1 Path simulation using data stuttering

Definition 13 *The **segment length sequence** $N[\pi(s_0)] = [n_1, n_2, \dots]$ of a path $\pi(s_0)$ is an ordered sequence of natural numbers where $n_i = q - p$, s_p and s_q are nodes of change such that $s_p|_D \in e_{i-1}$, $s_q|_D \in e_i$ for two adjacent data equivalence classes e_{i-1}, e_i in $T[\pi(s_0)]$.*

Thus, $N[\pi(s_0)] = [n_1, n_2, \dots]$ is a sequence of natural numbers that represents the number of steps needed to reach from one equivalence class to another along the path $\pi(s_0)$.

For example, the two paths, π and π', from Table 1 (which are visually illustrated in Figure 3) have the same trace sequence $T[\pi] = T[\pi'] = [A, B, C]$, where $A = \{(x,y) \mid x \le y\}$, $B = \{(x,y) \mid x > y,\ x < 3y\}$, and $C = \{(x,y) \mid x > y,\ x \ge 3y\}$. However, the segment length sequence $N[\pi] = [1,3]$ while $N[\pi'] = [1,1]$. The set of nodes of change in π is $\{(n_0,2,2),(n_1,3,2),(n_4,6,2)\}$ whereas the set of nodes of change in π' is $\{(n_0,1,1),(n_1,2,1),(n_4,3,1)\}$.

We say $N[\pi(s_0)] \preceq N[\pi'(s'_0)]$ if $\forall i,\ n_i \le m_i$ where $n_i \in N[\pi(s_0)]$ and $m_i \in N[\pi'(s'_0)]$, $N[\pi(s_0)] \prec N[\pi'(s'_0)]$ when $N[\pi(s_0)] \preceq N[\pi'(s'_0)]$ and $n_i < m_i$ for some i. It is easy to see that $N[\pi'] \preceq N[\pi]$ in Figure 3. We also say that π' is a *trajectory reduction* of π if $T[\pi] = T[\pi']$ and $N[\pi'] \preceq N[\pi]$.

Definition 14 *A path $\pi(s_0)$ is a **minimal path** if and only if $N[\pi'(s'_0)] \not\prec N[\pi(s_0)]$ for any path $\pi'(s'_0)$ with $T[\pi(s_0)] = T[\pi'(s'_0)]$ and $s_0 \simeq s'_0$.*

We call s_0 a minimal state when $\pi(s_0)$ is a minimal path. Intuitively, a minimal path is a path with minimal segment lengths among all paths with the same trace.

The $\preceq$ relation defines a partial order on the set of paths, and thus, there can be incomparable paths using the $\preceq$ relation. The above definition ensures that those incomparable minimal paths are all considered and will be included in our reduced domain. The definition also ensures the existence of a minimal path per trace as long as the data constraints are deterministic. Note that for some special cases, such as when the system has a unique maximal trace with linear data conditions, the $\preceq$ relation defines a total order on the set of paths.

Our reduced domain for data variables is composed of data nodes on the minimal paths.

Definition 15 *D' is a subset of D including $\{R^n(s_0)|_D \mid 0 \le n \le \Sigma n_j,\ n_j \in N[\pi(s_0)],\ s_0$: minimal state, $\pi(s_0)$: minimal path $\}$.*

D' is the reduced domain for abstraction including one trace representative per minimal path up to the last node of change. It includes every data node on a minimal path obtained by repeatedly applying the data transition relation from each minimal state until it reaches the last node of change. Using this reduced domain, we can guarantee that any path from the original system can be simulated by a path in the abstract system—this claim is formally proved in the next section.

5.2.2 Abstraction We first introduce a data identity transition R^D_{id} defined as $((n,x),(n',x)) \in R^D_{id}$ if $((n,x),(n',x')) \in R$ and $x \equiv x'$ for any control nodes n, n' and data nodes x, x'. Intuitively, the data identity transition allows the system to stay in the same data node while the control node changes according to the transition relation R. This allows for stuttering of data nodes.

For a given constrained data transition system $M = (S, S_0, R, L, AP)$, let $M' = (S', S'_0, R', L', AP')$ be an abstracted transition system of M where $AP' = AP$, $S' = N \times D'$, $S'_0 = N_0 \times D'_0$, $L' = S' \lhd L$, and $R' = (R \cup R^D_{id}) \cap (S' \times S')$. Here, D' is the reduced bound defined in Definition 15 and $D'_0 = \{d'_0 \in D' \mid d'_0 \equiv d_0$ for some $d_0 \in D_0\}$.

Theorem 3 *For any path $\pi = s_0 s_1 \dots s_n \dots$ in M, there exists an* equivalent path $\pi' = s'_0 s'_1 \dots s'_n \dots$ *in M' such that $s_i \simeq s'_i$ for all i.*

Proof By the definition of D', there exists a minimal path $\tilde{\pi}(t_0)$ in M' such that $T[\pi] = T[\tilde{\pi}]$ and $N[\tilde{\pi}] \preceq N[\pi]$ where $t_0 \simeq s_0$. Define $\pi' = s'_0 s'_1 \dots s'_n \dots$ as follows:
(1) $\forall i,\ s'_i\,|_N = s_i\,|_N$.
(2)

$$s'_i\,|_D = \begin{cases} t_0\,|_D & \text{if } i = 0, \\ R(s'_{i-1})|_D & \text{if } R(s'_{i-1})|_D \equiv s_i\,|_D \\ & \quad \text{and } R(s'_{i-1})|_D \in D', \text{ and} \\ s'_{i-1}\,|_D & \text{otherwise.} \end{cases}$$

$s_i \simeq s'_i$ is guaranteed by the fact that $T[\pi] = T[\tilde{\pi}]$ and $N[\tilde{\pi}] \preceq N[\pi]$. $s'_i \in S'$ for all i since any node of change along the path $\tilde{\pi}$ is included in D'. π' is a path in M' since $R'(s'_i, s'_{i+1})$ holds either by $R(s'_i, s'_{i+1})$ or by the data identity transition. □

By Theorem 3, for any path in M there is an equivalent path in M' with equal length with data stuttering.

To illustrate, suppose a system has two data variables x, y with two numeric conditions $x > y$, $x < 3y$, and data transition constraints $x' = x + 1$, $y' = y$. With these constraints, the path π (Table 1) through the state space can be simulated by π' since data stuttering is allowed in the abstract system. The path π' in the abstract system is actually a legal path in M' by Lemma 1.

Definition 16 *$s' \in S'$ is on the **same trajectory** as $s \in S$ if and only if*

1. *$s \simeq s'$ and*
2. Existence of trajectory reduction:
   Given path $\pi(s) = s s_1 s_2 \dots$ in M, $\exists \pi'(s') = s' s'_1 s'_2 \dots$ in M' such that
   (a) trace equivalence: *$T[\pi(s)] = T[\pi'(s')]$ and*

(b) path equivalence of each trace segment:
$s_i \mid_D \equiv s'_i \mid_D$ *implies* $s_i \simeq s'_i$ *for all* i*, and*
(c) trajectory reduction: $N[\pi'(s')] \preceq N[\pi(s)]$.

Intuitively, a state $s' \in M'$ is on the same trajectory as $s \in M$ if a path $\pi'(s')$ has the same trace as a path $\pi(s)$ and it reduces the number of steps needed to reach each data equivalence class along the path $\pi(s)$. Note that if $\pi'(s'_0)$ in M' is an equivalent path of $\pi(s_0)$ then s'_0 is on the same trajectory as s_0 when M is a system with deterministic data transition constraints.

Now, we define a relation between M and M' to prove that M' simulates M.

Definition 17 *$H \subseteq S \times S'$ is a relation on $S \times S'$ such that $H(s, s')$ if and only if s' is on the same trajectory as s.*

Theorem 4 *H is a simulation relation between M and M'.*

Proof

1. Given $H(s, s')$, $L(s) = L(s') = L'(s')$ from $s \simeq s'$ and by the definition of the labelling function. $L(s) \cap AP' = L'(s')$ is obvious since $AP = AP'$.
2. Suppose $R(s, t)$ for some $t \in S$. By the determinism of control node transition and data transition constraints, the path $\pi(st)$ is uniquely defined and there is a path $\pi'(s'k)$ in M' such that $T[\pi(st)] = T[\pi'(s'k)]$ and $N[\pi'(s'k)] \preceq N[\pi(st)]$ by $H(s, s')$.
   (a) Case 1 : $k \simeq t$ when $k \mid_D \equiv t \mid_D$
   Let $t' = k$ and $\pi'(t') = \pi'(s'k) - \{s'\}$. $t' \simeq t$ is clear and $R'(s', t')$ is a legal transition in M' by $R'(s', k)$. $H(t, t')$ directly follows from $H(s, s')$.
   (b) Case 2 : $t \simeq s \simeq s'$ when $k \mid_D \not\equiv t \mid_D$.
   Let $t' = s'$ and $\pi'(t') = \pi'(s'k)$. $t' \simeq t$ and $R'(s', t')$ holds by identity transition. $\pi'(t')$ satisfies the trace equivalence and path equivalence condition. To see that the trajectory reduction holds, let $N[\pi'(s'k)] = \{n_0, n_1, n_2, \dots\}$ and $N[\pi(st)] = \{m_0, m_1, m_2, \dots\}$. We can infer that $n_0 < m_0$ and $\forall i, n_i \leq m_i$ from $N[\pi'(s'k)] \preceq N[\pi(st)] \wedge t \simeq s \wedge k \not\simeq t$. Note that $N[\pi(t)] = \{m_0 - 1, m_1, m_2, \dots\}$, i.e., $m'_0 = m_0 - 1$ and $\forall i,\ m'_i = m_i$ for $m'_i \in N[\pi(t)]$. $N[\pi'(s'k)] = N[\pi'(t')] \preceq N[\pi(t)]$ follows.

Therefore, t' is on the same trajectory as t and $H(t, t')$ holds. □

Theorem 5 *M' simulates M.*

Proof Directly follows from Theorem 3. □

We conclude that domain abstraction based on trajectory reduction is conservative, and thus, any $ACTL^*$ formula satisfied by M' is also satisfied by M.

6 Automation

We illustrate the theory in the previous section with the ASW example. At the same time, we introduce the algorithms automating the abstraction as well as a brief discussion on the computational complexity. The algorithms presented here are a possible way of automation. Efficiency and optimization are further research issues.

```
get_equi_class(C)
    if C = {c_i}  /* only one condition */
      T = {c_i}, F = {¬c_i}
      return T ∪ F
    else
        C = C − {c_i}  /*select any condition */
        Permute = get_equi_class(C)
        T = F = ∅
        while (Permute ≠ ∅)
            Let p ∈ Permute,
                Permute = Permute − {p}
            T = T ∪ {c_i ∧ p}
            F = F ∪ {¬c_i ∧ p}
        return T ∪ F.
end get_equi_class

get_rep(P)
    Set_Rep = ∅
    for each e_i ∈ P
            if feasible(e_i)
              select x ∈ e_i
              Set_Rep = Set_Rep ∪ {x}
end get_rep
```

Fig. 4 Identifying data equivalence classes and representatives

6.1 Constraint-free Systems

As we showed in the previous section, one representative data value from each data equivalence class suffices to verify properties for constraint-free data transition systems. For automatic selection of data representatives, the data equivalence classes in the system must first be identified, and then we select a random value from each identified equivalence class.

Figure 4 outlines the algorithm for automatic computation of the reduced domain; get_equi_class computes all possible equivalence classes using the set of numeric conditions extracted from the specification and the property in question. For example, if $C = \{c_1, c_2\}$ where $c_1 \equiv f(x) > 0$ and $c_2 \equiv g(x) \leq 0$, then $get\_equi\_class(C)$ produces $T = \{(f(x) > 0) \wedge (g(x) \leq 0)\ , (f(x) > 0) \wedge \neg(g(x) \leq 0)\}$ and $F = \{\neg(f(x) > 0) \wedge (g(x) \leq 0), \neg(f(x) > 0) \wedge \neg(g(x) \leq 0)\}$. The union of T and F will be returned as the set of possible equivalence classes.

Next, get_rep checks the feasibility of each equivalence class from the set $P = get\_equi\_class(C)$ and returns a set of random representatives from each of the feasible equivalence classes. The integer valued representative from each equivalence class is effectively computed by finding real-valued representatives and obtaining the closest integer in the class—a problem of polynomial complexity. In general, if we have n conditions, we have 2^n possible equivalence classes. In actuality, the number of satisfiable equivalence classes is much smaller. There are ways of reducing the number of equations we need to solve to find the equivalence classes, but this problem is beyond the scope of this paper and will not be discussed further here.

Using the ASW system from Section 3, we show how the algorithm works for checking properties such as "the command will not be issued if the DOI is currently Off and altitude is above threshold". This property ($safety_1$) can be

equiv. class	min(a)	max(a)	min(t)	representative (a, t)
e_1	0	35000	2000 @ $a = 0$	(0, 2000)
e_2	2000	35000	2000 @ $a = 2000$	(2000, 2000)
e_3	2040	40000	2000 @ $a = 2040$	(2040,2000)
e_4	2001	35700	2000 @ $a = 2001$	(2001,2000)

Table 2 Selecting representative from each equivalence class

specified in CTL^* as:

$AG(\ (DOIval = Off) \wedge (altitude > threshold)$
$\Rightarrow AX \neg command\ )$

To compute the equivalence classes needed for abstraction, the numeric conditions are extracted from the transition relation and the property in question. In this example, *altitude* $<$ *threshold*, *altitude* $\geq$ *threshold*, and *altitude* $\geq$ *threshold* $+ \frac{threshold}{50}$ are the data conditions extracted from the transition constraints and *altitude* $>$ *threshold* is extracted from the property $safety_1$. Note that this condition is not part of the system description (i.e., initial state and transition conditions) but is rather an environment condition specified in the property for verification. These conditions define the equivalence classes. In the ASW model, there are only four satisfiable equivalence classes: (below, a represents *altitude*, and t represents *threshold*).

$$e_1 \ : \ a < t \ \wedge \ a \leq t \ \wedge \ a < t + \frac{t}{50}$$
$$e_2 \ : \ a \geq t \ \wedge \ a \leq t \ \wedge \ a < t + \frac{t}{50}$$
$$e_3 \ : \ a \geq t \ \wedge \ a > t \ \wedge \ a \geq t + \frac{t}{50}$$
$$e_4 \ : \ a \geq t \ \wedge \ a > t \ \wedge \ a < t + \frac{t}{50}$$

Using a constraint solver, the maximum and the minimum value in the class are identified and one of them is used for selecting the representative (Table 2). The constraint solver $CLP(q, r)$ [18] is used for this purpose.[8] The last column shows the *(altitude, threshold)* pair representing each equivalence class—this reduces the input domain to *altitude:* $\{0, 2000, 2001, 2040\}$ and *threshold:* $\{2000\}$. With this abstraction, the property $safety_1$ is verified to be true using the model checker NuSMV [14] in seconds.

6.2 Constrained Systems

For constrained data transition systems, constructing a reduced data domain is more involved; it is necessary to identify data values along a minimal path for each unique trace. There is no guarantee that the abstraction will produce a reduced domain for general cases and we may end up with the original domain D in the worst case. Nevertheless, in practice, this method results in domain reduction for many systems.

The automation for path reduction presented in this section relies on some additional constraints on the transition system. The automation for more general cases is a research issue we are actively pursuing.

[8] Alternatively, an integer programming tool can be used.

```
min_trace(e_i, SAT, step, REACH)
    while REACH ≠ ∅
        pick e_j ∈ REACH
        REACH := REACH − {e_j}
        k := 1;, distance := ∞
        while(1)
            Target := f^{k+step}(e_j)
            if SAT ∧ Target is satisfiable
              SAT := SAT ∧ Target
              step := step + k
              min_trace(e_i, SAT, step, REACH)
            else
                d := distance_test(SAT, Target)
                if d < distance
                  distance := d ; k^{++};
                else break;
    add (SAT, step) for the data node of a minimal state.
```

Fig. 5 Trace and minimal data path generation algorithm

Assumptions for Automation:

1. Data transition constraints linearly change the values of data variables, i.e., they are in the form of $x' = ax + b$ where $a \in \{0, 1\}$ and b is an integer constant.
2. Data transition constraints are either independent of control nodes, i.e., the change of the data value is global, or consists of a linear constraint, constant assignments, and/or stuttering, i.e., we require *partial monotonicity* of all data constraints that are not global.
3. Each numeric function $c(v)$ appearing in the set of numeric conditions C (refer to Section 4.1) is continuous and grows within a polynomial bound.

The second assumption makes it possible to separate the data transitions from the control transitions. Here, local data transition constraints are partially allowed for constant assignment such as $x' = 0$ in a specific control node. This allows, for example, resetting a timer variable. Assumptions 1 and 3 guarantees that the trace of each path is finite since (1) the numeric conditions are not cyclic, (2) the change in data values follow a linear trajectory, and (3) the number of equivalence classes is finite.

By these assumptions, we know that the number of reachable data equivalence classes is finite. Therefore, for any given equivalence class that may have the initial data nodes, we can compute a sequence of reachable equivalence classes with minimal segment lengths based on the data transition constraints with the aid of a constraint solver. Figure 5 outlines a prototype minimal data state generation algorithm.

The min_trace algorithm accepts an equivalence class e_i, a satisfiable list SAT which is initially e_i, a natural number $step$ which is initially 0, and a set of data equivalence classes $REACH$ which is initially the set of all data equivalence classes except e_i. $REACH$ represents all data equivalence classes whose reachability from e_i we must investigate.

When numeric functions appearing in the numeric conditions have higher degree than linear, the initial set $REACH$ would be different from that of the linear case. In the linear case, each class can only appear once on a trace. If we

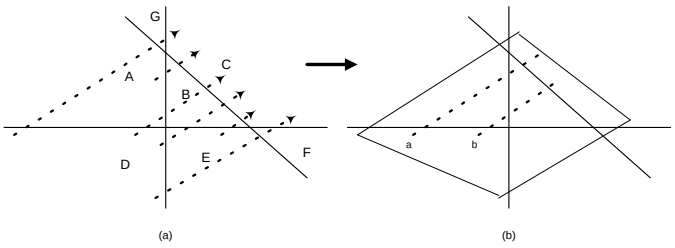

Fig. 6 Domain Reduction from minimal data trajectories

have higher degree polynomials in the data conditions, we need to include several copies of each equivalence class in $REACH$ since a class may appear several times on a trace. The maximum number of times that each class can appear is determined by the degree of the data conditions. The algorithm is described here for linear cases; other cases can be treated in a similar way except for the choice of the initial set of $REACH$.

The algorithm recursively computes the satisfiability of $SAT \wedge (f^m(e_j))$ by exercising all permutations of the set of equivalence classes until there are no more equivalence classes to be added to the satisfiable trace SAT. When there are no more satisfiable equivalence classes to be added to SAT, we then pick a point x from the region of SAT. x generates a sequence that passes the reachable data equivalence classes in minimal number of steps by applying $step$ times, the data transition function f to x.

Our reduced domain is the smallest convex hull including the maximum and minimum values in such sequences. The algorithm identifies minimal data trajectories that minimize the segment lengths of a trace and the convex-hull includes all the minimal data trajectories. Figure 6 (a) shows a view of minimal data trajectories generated using the algorithm; A to G represent data equivalence classes partitioned by the numeric conditions and the dashed arrow represents each minimal data trajectory identified by the algorithm. The resulting reduced domain is shown in (b). Note that the minimal data trajectory represented as a in (b) is not originally identified by the algorithm, and yet included in the convex-hull.

Figure 7 describes the minimal data path generation process in the min_trace algorithm. In order to find a minimal path from e_i to e_j, the linear transformation using the data transition constraints $x' = f(x)$, where $f(x)$ is linear, is applied to the equivalence class e_j repeatedly, until it reaches e_i. The minimal data path is generated from a point x in the intersection region $e_i \wedge f^n(e_j)$.

For the case when a class e_j is not reachable from e_i, the Euclidean distance between e_i and $f^n(e_j)$, defined as $\mid x - f^n(y) \mid$, where x and y are pre-selected random values from e_i and e_j respectively, is used as a stopping condition in each n_{th} step of the transformation. The algorithm stops computing if the distance grows.

6.2.1 The ASW Revisited. Recall the constrained altitude switch example described in Section 3, with data transition constraints $altitude' = altitude + 10$ and $threshold' = threshold$. The property in question ($safety_2$) is "the ASW shall never turn the DOI on while ascending" which can be expressed in CTL^* as

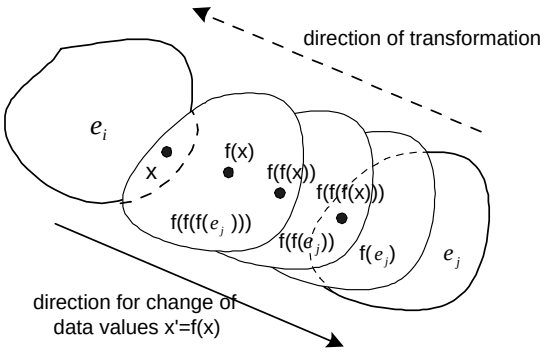

Fig. 7 Transformation operation on an equivalence class

$$AG(\ (DOIval = OFF \wedge \neg(DOI\_Status = On)) \Rightarrow AX\neg(DOI\_Status = On)\ ),$$

assuming the system models an ascending aircraft (i.e., $altitude' = altitude + 10$).

Note that the constrained altitude switch example satisfies the assumptions for automation. Since the data constraint is global, it is independent of control nodes— the second assumption is satisfied; the data transition constraint function is linear and so the first assumption is satisfied; and, the numeric conditions are also linear, and so the third assumption is satisfied.

Again, the data conditions are extracted from the numeric conditions as follows.

$$e_1 \ : \ a < t \ \wedge \ a < t + \frac{t}{50}$$
$$e_2 \ : \ a \geq t \ \wedge \ a < t + \frac{t}{50}$$
$$e_3 \ : \ a \geq t \ \wedge \ a \geq t + \frac{t}{50}$$

Using the algorithm, we compute a data node for a minimal state and corresponding trace based on the set of equivalence classes $\{e_1, e_2, e_3\}$ as follows.

Starting from an equivalence class e_i, we check the satisfiability of $e_i \wedge f(e_j)$ where $f(e_j)$ is a transformation of each numeric condition in e_j by applying the data transition constraint. For example, the data condition $a < t$ that helps define e_1 would be transformed to $a + 10 < t$ since the data constraints force $t' = t$ and $a' = a + 10$. If $e_i \wedge f(e_j)$ is satisfiable, it means that e_j is reachable from e_i in one step. Otherwise, there are two cases; e_j is not reachable from e_i or is reachable from e_i in more than one step. We measure progress by computing the Euclidean distance between the two regions e_i and $f^n(e_j)$ from $n = 1$ — if the distance increases in an iteration, given our restrictions, we know the equivalence class is not reachable and we can terminate this computation. If the distance is shrinking, we keep applying the transformation to e_j until $e_i \wedge f^n(e_j)$ is satisfied. When $e_i \wedge f^n(e_j)$ is satisfiable, then n is the minimum number of steps required to move from e_i to e_j. The constraint solver $CLP(q, r)$ [18] is used for the satisfiability check in the algorithm.

Table 3 shows a sample trace generation starting from the equivalence class e_1. The domain constraints $0 \leq a \leq 40000$

trace	step	k	$SAT \wedge Target$	satisfy?	d	continue?	(a,t)
$e_1 \rightarrow e_3$	0	1	$a < t$ & $a < t + t/50$ & $a + 10 \geq t$ & $a + 10 \geq t + t/50$	no	30	yes	
		2	$a < t$ & $a < t + t/50$ & $a + 20 \geq t$ & $a + 20 \geq t + t/50$	no	20	yes	
		3	$a < t$ & $a < t + t/50$ & $a + 30 \geq t$ & $a + 30 \geq t + t/50$	no	10	yes	
		4	$a < t$ & $a < t + t/50$ & $a + 40 \geq t$ & $a + 40 \geq t + t/50$	no	0	yes	
		5	$a < t$ & $a < t + t/50$ & $a + 50 \geq t$ & $a + 50 \geq t + t/50$	yes		finish	$a = 1990$, $t = 2000$
$e_1 \rightarrow e_3 \rightarrow e_2$	5	1	$a < t$ & $a < t + t/50$ & $a + 50 \geq t$ & $a + 50 \geq t + t/50$ & $a + 60 \geq t$ & $a + 60 < t + t/50$	no	10	yes	
		2	$a < t$ & $a < t + t/50$ & $a + 50 \geq t$ & $a + 50 \geq t + t/50$ & $a + 70 \geq t$ & $a + 70 < t + t/50$	no	20	no	

Table 3 Sample trace computation for Altitude Switch

and $2000 \leq t \leq 35000$ are assumed in the table. The table shows that e_3 is reachable from e_1 in five steps and there are no other reachable classes further. This produces the trace $e_1 \rightarrow e_3$. Once a trace $[e_1, e_3]$ is computed, the algorithm attempts to determine if the trace $[e_1, e_3, e_2]$ is possible. Here, the value of $step$ continuously increases from previous value of k to retain the information of the minimal number of steps from e_1 to e_2.

Two traces $\{e_1 \rightarrow e_3\}$, $\{e_1 \rightarrow e_2 \rightarrow e_3\}$ are generated by the application of the algorithm to e_1, e_2, and e_3 in order. Both traces required 5 steps. We identified the initial points of the traces to be $(a, t) = (1990, 2000)$ in the region that satisfies $\{e_1 \rightarrow e_3\}$ and $(a, t) = (1991, 2000)$ in the region that satisfies $\{e_1 \rightarrow e_2 \rightarrow e_3\}$. With these initial points, we can generate the required domain for *altitude* (a) and *threshold* (t) by iterating the data constraints five times (one for each trajectory—in this case they were both five steps long). This yields the following domains:

$altitude$: { 1990, 1991, 2000, 2001, 2010, 2011, 2020, 2021, 2030, 2031, 2040, 2041 }

$threshold$: { 2000 }

As a result, we have a reduced domain $altitude = [1990, 2041]$ and $threshold = [2000, 2000]$. Using the reduced domain, the property $safety_2$ is verified in seconds with NuSMV [14].

Note that the minimal data trajectory itself—without using the convex hull including the data values on the trajectory—can be used as a reduced domain in this example since the system has a unique maximal trace $[e_1, e_2, e_3]$, and thus, all paths are totally ordered by the $\preceq$ relation.

6.2.2 Complexity. We briefly discuss here the complexity involved in the computation of the reduced domain. In general, the number of minimal trace computations required would be $n!$ when there are n data equivalence classes. However, in practice this computation still seems feasible due to several factors. First, the number of equivalence classes, which is dependent on the number of conditions in C, is small in practice. Second, the actual number of computations may be far less than $n!$. Since at each step, we determine the possible candidates for the next data equivalence class in the trace, when there are no such candidates we can stop exploring that trace any further. Finally, one may group interrelated numeric conditions into sets, partition the input domain for each such set, and compute the minimal data path for each partition. Such an approach would reduce the number of data equivalence classes that must be considered together in computing the minimal trace.

7 Example: Train Control System

Figure 8 shows a hypothetical railroad gate control system motivated by a model from the hybrid systems domain [28].

A train runs at speed 50 meters per second on a circular track with a gate. When the train approaches the gate, the controller starts to lower the gate, which makes the gate start the lowering process, and raise the gate back after the train past the gate. Since there is a physical limitation on the speed of lowering/raising the gate, minimum time requirements for lowering/raising the gate to full extent must be imposed in the model. The timer t and the timeout condition $t > timeout$ are used to model the time requirement. The timer increases by 1 when the gate is in the lowering/raising state, resets to zero when the gate enters the Open or Closed states, and remains the same otherwise.

In Figure 8, the distance of the train from the gate is represented by the variable x. The value of x is set between $1,900$ and $4,900$ at the distance of 100 meters past gate since the length of circular track is between $2,000$ and $5,000$. The initial distance value x is between $1,000$ and $5,000$.

The safety requirement of this system is to ensure that the value of $timeout$ is large enough to open/close the gate to full extent, and at the same time, ensure that the gate is safe for pedestrians, i.e., it is fully closed whenever the train is within 10 meters of the gate. The safety property ($safety_3$) can be expressed in CTL^* as
$AG((-10 \leq x \leq 10) \Rightarrow (gate = Closed))$.

This model shows a typical example with a small number of control nodes and data variables over large domain which makes model checking infeasible without abstraction. Note that the automatic domain reduction algorithm is applicable

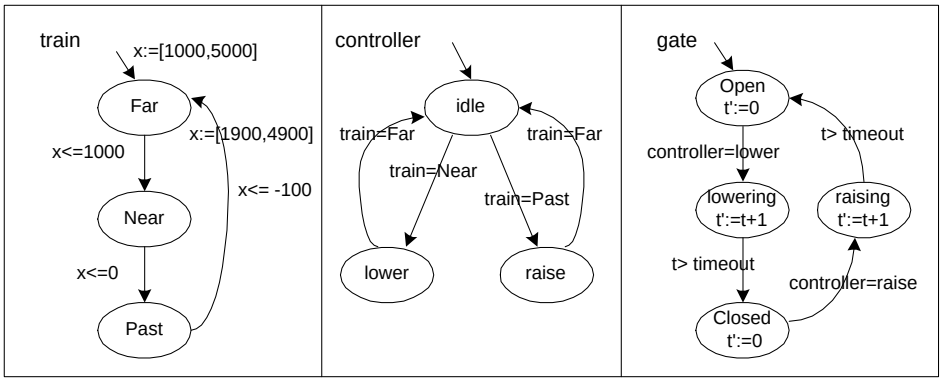

Fig. 8 Train Control System

to this system since (1) the numeric conditions are linear, (2) the data transition constraints are also linear, and (3) the data transition constraints consist of a linear constraint, constant assignments, and stuttering even though they are control node dependent.

We decompose the numeric conditions into two groups. The decomposition is applicable since the variables t and x are not interrelated, i.e. they do not share data variables.

1. $\{\ 1900 \leq x \leq 4900, x \leq 1000,$
   $x \leq 0, x \leq -100, -10 \leq x \leq 10\ \}$.
2. $\{\ t > timeout, t = 0\ \}$.

The first group is identified from numeric conditions on the variable x from the model and the property in question. The second group is from the numeric condition on the variable t and the assignment expression. The first group generates eight data equivalence classes, while the second one generates three as follows. Here, the domain of t is assumed to be $[0, \infty]$ while the domain of x is not specified.

$$\left\{\begin{array}{l} e_{x1} : x \leq -100 \\ e_{x2} : -100 < x < -10 \\ e_{x3} : -10 \leq x \leq 0 \\ e_{x4} : 0 < x \leq 10 \\ e_{x5} : 10 < x \leq 1,000 \\ e_{x6} : 1,000 < x < 1,900 \\ e_{x7} : 1,900 \leq x \leq 4,900 \\ e_{x8} : x > 4,900 \end{array}\right\} \left\{\begin{array}{l} e_{t1} : 0 < t \leq time\_out \\ e_{t2} : t > time\_out \\ e_{t3} : t = 0 \end{array}\right\}$$

Also, the data transition constraints can be extracted as: $\left\{\begin{array}{l} x' = x - 50 \\ t' = t + 1 \end{array}\right\}$

The data stuttering $t' = t$ is ignored since it does not play a role in the minimal data path generation algorithm.

Note that the initial values for the variables can be used to increase the efficiency of the abstraction process; the number of computations for the minimal data path generation can be reduced by checking the satisfiability of the initial value conditions $x = [1000, 5000]$, $t = 0$ with each equivalence class. In this case, e_{x5}, e_{x6}, e_{x7}, and e_{x8} are the only possible initial data equivalence classes of a trace for the variable x and e_{t3} is the only possible initial data equivalence class of a trace for the variable t.

We first set the $timeout$ to 20 seconds and checked the property $safety_3$. The algorithm generates a minimal data path:
$x = \{-149, -99, -49, 1, 51, \ldots, 4751, 4801, 4851, 4901\}$, $t = \{0, 1, 2, \ldots, 20, 21\}$.
Table 4 shows a sample trace computation. The random value from each satisfiable region SAT is chosen by trying previously selected data values first in order to reduce the number of distinct values. For example, the minimal data value for x for trace $e_{x5} \rightarrow e_{x4} \rightarrow e_{x2} \rightarrow e_{x1}$ can be any value in $(50, 60]$. The value 51 is selected among them since it was already chosen for the previous trace. Using this reduced domain, the model checker NuSMV generates a counter example in seconds.

Next, the $timeout$ is set to 10 seconds, and this time the minimal data path for t becomes $t = \{0, 1, \ldots, 10, 11\}$ whereas that of x remains same. The property $safety3$ is verified in seconds. Therefore, $timeout = 10$ guarantees the satisfaction of the $safety_3$.

Note that our reduced domain for x would be $x = [-149, 4901]$ if we used a continuous bound from the convex-hull including the minimal data trajectory, which may be still too large to model check. Nevertheless, we use the collection of values on minimal data paths itself since the data conditions and transition constraints are all one dimensional—trivial case of systems with a unique maximal trace— and thus, the minimal data path represents any other data path with the same segment lengths.

We can combine abstraction by scaling numeric variables with our domain reduction especially for the case when the use of a continuous bound is necessary, i.e.,when the system has several maximal traces. For example, we can scale down the domain and numeric conditions of the variable x by a factor of 10 producing modified equivalence classes $[e_{x_1}, e_{x_2}, e_{x_3}, e_{x_4}, e_{x_5}, e_{x_6}, e_{x_7}, e_{x_8}] = [x \leq -10, -10 < x \leq -1, -1 < x \leq 0, 0 < x \leq 1, 1 < x \leq 100, 100 < x < 190, 190 \leq x \leq 490, x > 490]$. The same algorithm produces a reduced bound $x = [-14, 491]$ for the data transition constraint $x' = x - 5$. Checking the safety property $AG((-1 \leq x \leq 1) \rightarrow (gate = Closed))$ produces the same result as the one without scaling. Effective use of our domain reduction technique along with other existing techniques is a topic for further research.

8 Discussion

We presented a new static abstraction technique, *domain reduction abstraction*, for model checking software specifications with inter-related numeric variables over large domains.

trace	step	k	$SAT \wedge Target$	satisfy?	d	continue?	x
$e_{x5} \rightarrow e_{x4}$	0	1	$10 < x \leq 1,000$ & $0 < x - 50 \leq 10$	yes		finish	51
$e_{x5} \rightarrow e_{x4}$ $\rightarrow e_{x3}$	1	1	$10 < x \leq 1,000$ & $0 < x - 50 \leq 10$ & $-10 \leq x - 100 \leq 0$	no	30	yes	
		2	$10 < x \leq 1,000$ & $0 < x - 50 \leq 10$ & $-10 \leq x - 150 \leq 0$	no	80	no	
$e_{x5} \rightarrow e_{x4}$ $\rightarrow e_{x2}$	1	1	$10 < x \leq 1,000$ & $0 < x - 50 \leq 10$ & $-100 < x - 100 < -10$	yes		finish	
$e_{x5} \rightarrow e_{x4}$ $\rightarrow e_{x2} \rightarrow e_{x1}$	2	1	$10 < x \leq 1,000$ & $0 < x - 50 \leq 10$ & $-100 < x - 100 < -10 \& x - 150 \leq -100$	no	0	yes	
	4	2	$10 < x \leq 1,000$ & $0 < x - 50 \leq 10$ & $-100 < x - 100 < -10 \& x - 200 \leq -100$	yes			51

Table 4 Sample trace computation for the train controller system

We aimed for a fully automated abstraction technique that can be applied independent of specific model checking tools. While other existing abstraction techniques require some degree of user guidance and/or aid of decision support tools during the model checking process, our technique can be fully automated and used as a pre-process before model checking.

Domain reduction abstraction can be viewed as a static version of predicate abstraction; in predicate abstraction, a boolean variable replaces a range of values of a data variable, whereas we use a set of actual values to represent the range of a data variable. In our approach, computations for the abstraction is separated from the model checking process, and thus, does not cause non-termination problem inherent in symbolic iterative abstraction techniques. Especially, for constraint-free systems, our approach guarantees a sound and complete abstraction while predicate abstraction may require several iterations of counter example analysis and refinements of the predicates.

Our approach moves the performance overhead caused by the abstraction process up-front instead of using decision procedures on-the-fly while model checking. In our case studies, the technique has been effective and we believe that in conjunction with other reduction techniques, such as slicing, we will be able to extend the reach of model checking into the domain of software specification for a class of systems of particular interest to us, namely safety-critical control systems.

Nevertheless, we cannot guarantee that the abstracted domain is small enough to model check, especially for the constrained systems. The effectiveness and practicality of our approach need to be evaluated by (1) benchmark application of the technique on large systems and (2) performance comparison with current on-the-fly abstraction techniques.

Moreover, the limitation to deterministic data constraints is quite severe. Most realistic applications require at least limited non-determinism in the data constraints. For example, we would like to model the altitude in the ASW example to change at most 300 *up or down* each step. We have extended the theoretical basis for this case [12] and completed the proof, and are working on practical algorithms for automation along with approaches to relax the assumptions we have imposed for the automation.

References

1. Rajeev Alur, Luca de Alfaro, Thomas A. Henzinger, and F.Y.C. Mang. Automating Modular Verification. In *Proceedings of 10th International Conference on Concurrency Theory (CONCUR '99)*, pages 82–97, 1999.
2. Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.
3. Joanne M. Atlee and John D. Gannon. State-based model checking of event-driven system requirements. In *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems. Software Engineering Notes. Volume 16 Number 5*, 1991.
4. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriam K. Rajamani. Automatic predicate abstraction of c programs. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 36(5):203–213, May 2001.
5. Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.
6. B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. *Formal Methods and System Design*, 14(3):237–255, May 1999.
7. Tevfik Bultan, Richard Gerber, and Christopher League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *ISSTA*, 1998.
8. Tevfik Bultan, Richard A. Gerber, and William 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, July 1999.
9. Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.
10. W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Computer Aided Verification*, pages 316–327. Springer Verlag, 1997.
11. W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
12. Yunja Choi, Mats P.E. Heimdahl, and Sanjai Rayadurgam. Domain reduction abstraction. Technical Report 02-013. University of Minnesota, April 2002.
13. Yunja Choi, Sanjai Rayadurgam, and Mats Heimdahl. Automatic abstraction for model checking software systems with interrelated numeric constraints. In *Proceedings of the 9th ACM*

SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE-9), pages 164–174, September 2001.
14. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Checker, 2004. Available at http://nusmv.irst.itc.it/.
15. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transaction on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
16. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, July 2000.
17. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
18. Constraint logic programming over rational or real. Available at http://www.ai.univie.ac.at/clpqr/.
19. M. Colon and T. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304. Springer, 1998.
20. E. Emerson and K. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Thirteenth Annual IEEE Symposium on Logics in Computer Science*, pages 70–80, 1998.
21. Parice Godefroid, Doron Peled, and Mark Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering)*, 22(7):496–507, July 1996.
22. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the Computer Aided Verification(CAV 1997)*, pages 72–83, 1997.
23. O. Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
24. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.
25. John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, December 2000.
26. Mats P.E. Heimdahl, Jeffrey M. Thompson, and Michael W. Whalen. On the effectiveness of slicing hierarchical state machines: A case study. In *Proceedings of the Twenty-fourth EUROMICRO Conference*, volume 1, pages 435–444, 1998.
27. Constance Heitmeyer, James Kirby Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.
28. Thomas A. Henzinger. The theory of hybrid automata. In *Theoretical Computer Science 138:3-34, 1995.*, 1995.
29. Thomas A. Henzinger and Pei-Hsin Ho. HyTech: The Cornell Hybrid Technology Tool. In *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 265–294. Springer-Verlag, 1995.
30. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Rupak Majumdar. Lazy abstraction. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, pages 58–70, January 2002.
31. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software Engineering*, pages 279–295, May 1997.
32. Z. Manna, M. Colon, B. Finkbeiner, H. Sipma, and T. Uribe. Abstraction and modular verification of infinite-state reactive systems. In *Requirements Targeting Software and Systems Engineering (RTSE), LNCS*. Springer Verlag, 1998.
33. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
34. Kedar S. Namjoshi and Robert P. Kurshan. Syntatic program transformations for automatic abstraction. In *12th International Conference, CAV2000*, pages 435–449, July 2000.
35. Prasad Sistla and Patrice Godefroid. Symmetry and Reduced Symmetry in Model Checking. In *Proceedings of 13th Conference on Computer Aided Verification*, July 2001.
36. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.
37. Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.
38. Willem Visser, SeungJoon Park, and John Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proceedings of the Third ACM Workshop on Formal Methods in Software Practice*, pages 3–12, August 2000.
39. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Computer Aided Verification*, pages 88–97. Springer, 1998.