Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 03-023

Model Checking Software Requirement Specifications Using Domain Reduction Abstraction

Yunja Choi and Mats P. Heimdahl

May 12, 2003

Model Checking Software Requirement Specifications Using Domain Reduction Abstraction

Yunja Choi and Mats Heimdahl

Computer Science and Engineering, University of Minnesota,
{yuchoi, heimdahl}@cs.umn.edu

Abstract. Model checking techniques have not been effective in important classes of software systems characterized by large (or infinite) input domains with interrelated linear and non-linear constraints over the input variables. In a previous paper we proposed *domain reduction abstraction* based on data equivalence and trajectory reduction as an alternative and complement to other abstraction techniques. Domain reduction abstraction applies the abstraction to the input domain (environment) instead of the model.
In this paper, we describe a prototype integration of the technique in the NuSMV symbolic model checker and illustrate its applicability in the context of model checking software requirements specifications. Results from the verification of three industrial size avionics systems demonstrates the feasibility of the approach.

Keyword: Domain reduction abstraction, model checking software specifications, numeric constraints

1 Introduction

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 are prevalent in safety critical embedded systems such as aircraft control, train control, and medical device systems. In such systems, the transitions between control variables are guarded by various linear and non-linear data conditions, and the transitions between data values may be subject to various constraints which can be non-deterministic. For example, a temperature control system can have the guarding condition $temp < 10$ for a control transition and $temp' = temp + [-\alpha, \beta]$ as a constraint for the temperature change.

In previous papers [10, 11] we have investigated abstractions over the *input domain* of the systems rather than the system itself—a technique we call *domain reduction abstraction*. Domain reduction abstraction statically analyzes a model, extracts numeric conditions and constraints, reduces the data domain by selecting representative data values that subsume possible system behaviors, and leaves the control part of the system unchanged. For systems where there are no data constraints, the abstracted system bisimulates the original system, if there are data constraints, the abstracted system simulates the original system.

In domain reduction abstraction the effort of computing an accurately abstracted system is expended before verification and the abstraction cost is unrelated to the number of properties to be verified. We have observed that the verification of a substantial software specification will involve hundreds of properties that will have to be re-verified every time the software specification changes—something that happens quite frequently. In the face of scores of properties and frequent *regression verification*, we anticipate that domain abstraction will compare well with other proposed techniques that require the abstraction process to be performed for each property [4, 7, 12, 16, 19, 21]. Nevertheless, it may be the case that

the model is still too large to check after domain reduction abstraction. In that case, other techniques such as counter-example guided abstraction refinement [4, 12, 19] can be used in concert with domain reduction abstraction.

In this report, we describe a prototype integration of domain reduction abstraction into the symbolic model checker NuSMV [22] in connection with a linear/integer programming tool *lp_solve* [2]. Though domain reduction abstraction applies to both linear and non-linear data conditions and constraints in theory, our current prototype implementation is limited to linear data conditions and constraints because of some automation issues as discussed in Section 3. Extensions to the non-linear cases require a reliable constraint solving and/or numeric computation capability. Our implementation was developed as a technology demonstration and there are many rather obvious performance improvements we will investigate in the near future.

Using our prototype implementation, we demonstrate the usability and efficiency of domain reduction abstraction in the context of model checking software requirements specifications. The case examples demonstrate the following; (1) the abstraction is fully automated, (2) the abstracted domain can be reused unless numeric conditions are changed, and thus, it is suitable for *regression verification*, (3) and counter examples are straight-forward to interpret.

In the next section we briefly recalls domain reduction abstraction at an intuitive level. Section 3 presents our automation framework in some detail. We present our case studies in Section 4 followed by a discussion (Section 5).

2 Domain Reduction Abstraction

Domain reduction abstraction is motivated from the observation that only a subset of data values from a data domain has a distinct effect on the system behavior. For example, a system with a numeric condition $\{x < y\}$ over the domain $x = 0..100$, $y = 0..100$ would behave same as one with a reduced domain $x = \{1, 2\}$, $y = \{1, 2\}$; the reduced domain contains a combination of x, y values that makes the condition true or false. The approach is based on selecting representative values that will exercise all possible truth values of the numeric conditions in the system, resulting in huge performance gain when model checking.

To give the reader a general understanding of our approach, we provide an informal outline of the abstraction technique in this section. Formal definitions and proofs of soundness of the approach can be found in [10, 11].

2.1 Informal Description

We tackle the problem of numeric variables with two complementary abstractions; first, when the values of the data variables only depend on inputs from the system environment (we call it *constraint-free data transition systems*), a simple data abstraction technique based on a data equivalence relation is used. When the change of the data values is constrained by some data transition rules (we call it *constrained data transition systems*), one data trajectory (a series of data values satisfying all data-constraints) will be computed and used as a representative for all data trajectories with the same characteristic. Here, the characteristic of a data trajectory is determined by the ordered set of data-equivalence classes the data trajectory passes through.

Figure 1 shows a view of domain reduction abstraction. We partition the domain of numeric variables by the valuation of the numeric conditions appearing in the transition conditions and verification properties. For example, the partition A represents the region

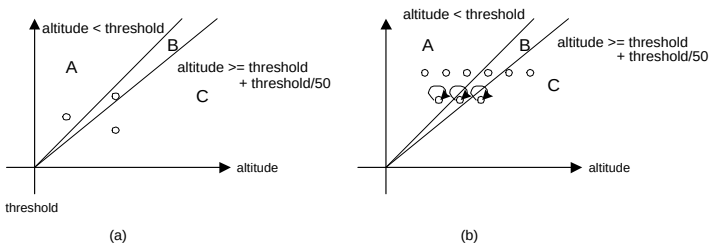

Fig. 1. General abstraction approach

that satisfies $altitude < threshold \wedge altitude < threshold + \frac{threshold}{50}$. We replace the set of possible values in a data-equivalence class with a randomly selected representative from the class (see (a) in Figure 1). This is a variation of the data abstraction technique suggested by Clarke et al. [13]; 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 closely related to that of partition testing [5].

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) in Figure 1). 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.

The major benefits of this approach are (1) it provides a sound abstraction for data-constrained systems and a sound and complete abstraction for constraint-free systems (in terms of the temporal logic CTL^*), (2) the computation of the abstraction is done before model checking so that the number of properties to be model checked is unrelated to the abstraction cost, unlike other existing counter-example guided iterative refinement abstraction techniques [4, 12, 19], (3) we can reuse the abstracted data domain as long as there is no change in data conditions and constraints, and (4) since it is orthogonal to other abstraction techniques, we can apply any other existing abstraction techniques in concert with domain abstraction when necessary.

2.2 Abstraction Theory

An extensive treatment of the theory behind *domain reduction abstraction* can be found in [10, 11]. In this section, we briefly restate the basic definitions and theorems to provide the foundation of the technique. We use the same system model introduced in [8] as a basis to classify our systems of interest.

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

- N is a finite set of control nodes and $N_0 \subseteq N$ is a set of initial control nodes.
- v is a finite vector of data variables over D where D is the Cartesian product of the domains of the data variables.
- Δ is a mapping from N^2 to $2^{D \times D}$.

– C is a finite set of conditions on v of the form $c := \alpha(v) \bowtie 0$ where $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$ and $\alpha : D \longrightarrow \Re$.

The system model defines a basic transition system $M = (S, S_0, R, AP, L)$ [8] where $S = N \times D$ is a set of states, $S_0 = N_0 \times D$ is a set of initial states, $AP = N \cup C$ is a set of atomic propositions, $L(n, v) = \{n\} \cup \{c \in C \mid c(v)\}$ labels each state with atomic propositions in AP, and R is a transition relation defined on $S \times S$ so that $R((m, x), (n, y))$ iff $(x, y) \in \Delta((m, n))$.

Definition 1 ***Data and state equivalence***

1. *$x, y \in D$ are **data equivalent**, written $x \equiv y$, iff $\forall c \in C : c(x) = c(y)$.*
2. *Two states $s, s' \in S$ are **state equivalent**, written $s \simeq s'$, iff $L(s) = L(s')$, i.e., $s|_N = s'|_N \ \wedge \ s|_D \equiv s'|_D$.*

We denote $D/_{\equiv}$ ($S/_{\simeq}$) for the set of equivalence classes induced by $\equiv$ ($\simeq$) on D (S) and $e_i(E_i)$ for the i^{th} data (state) equivalence class. 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. For a state $s = (n, d) \in S$, we use $s \mid_N = n$, $s \mid_D = d$ to represent the control node and the data node respectively. We would use D_i instead of D, if the projection to the i^{th} data variable is required.

Definition 2 *R is said to be a **constrained data transition** for D_i if for each state-equivalence class E_j, there is a finite set of data transition functions $F^i_{E_j} = \{f_i \mid f_i : D_i \longrightarrow D_i\}$ such that*

1. *$\{f_i(x) \mid f_i \in F^i_{E_j}\} \neq D_i$ for some $x \in D_i$, and* [1]
2. *$R(s, t)$ for $s \in E_j$ iff $(s|_D, t|_D) \in \Delta((s|_N, t|_N))$ and $t|_{D_i} = f_i(s|_{D_i})$ for some $f_i \in F^i_{E_j}$.*

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 is a finite set of functions—i.e., the data in the post-state is an application of a function to the data in the pre-state. When $F^i_{E_j}$ has more than one function element, the data transition is taken by non-deterministic choice among the several data transition functions. In this way, we allow finite non-determinism in the system. Nevertheless, for the simplicity of discussion, we assume that $F^i_{E_j}$ has a unique transition function f_i in this paper.

Theorem 1. *For a system with no constrained data transitions (constraint-free data transition systems) $M = (S, S_0, R, L, AP)$, let $D' = rep(D/_{\equiv})$* [2] *and $M' = (S', S'_0, R', L', AP)$ where $S' = N \times D'$, $R' = R \cap (S' \times S')$, $S'_0 = S_0 \cap S'$ and $L' = S' \lhd$* [3] *L. Then state equivalence relation $\simeq$ is a bisimulation relation between M and M'.*

Proof. See [11].

For systems with constrained data transitions, we can select a representative data trajectory that satisfies constrained data transitions by identifying an initial data value (*minimal data node*) of a *minimal data trajectory* –a data value v can be a representative of other data

[1] This condition is to make sure that the application of transition functions has different effect from random value assignment.

[2] A set of representative values from each class in $D/_{\equiv}$.

[3] $\lhd$ 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 [23].

value v' if for any data trajectory from v' there is a minimal data trajectory from v that moves through the same data equivalence classes in fewer steps.

In order to avoid repeating all formal definitions necessary to explain our approach, we simply provide informal descriptions of the three key definitions, *minimal data trajectory*, a *minimal data node*, and the *last node of change*.

In Figure 2, black dots form a *minimal data trajectory* that passes through a sequence of data equivalence classes $[A, B, C]$, where the equivalence classes are partitioned by two data conditions $\{x > y, x <= 3y\}$. The change of data values on the data trajectory is constrained by a constraint function $x' = x + 2,\ y' = y + 1$. The data trajectory is a minimal in a sense that the number of steps to pass through the class B is less than any other possible data trajectories starting from the equivalence class A. A *minimal data node* in the minimal data trajectory for the trace $[A, B, C]$ is a data node that belongs to A and whose next data value belongs to B. The *last node of change* is the first data node that belongs to C, as depicted in Figure 2.

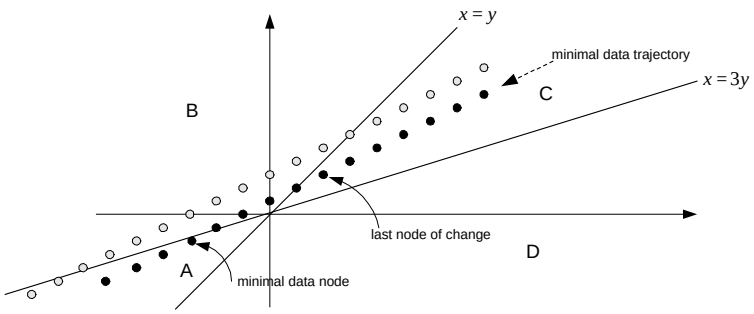

Fig. 2. Minimal data trajectory

Our reduced domain D' includes all the data values on a minimal data trajectory from the minimal data node to the last node of change. We then 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 for the stuttering of data nodes within the same data equivalence class.

Theorem 2. *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 $AP' = AP$, $S' = N \times D'$, $S'_0 = N_0 \times D'$, $L' = S' \lhd L$, and $R' = (R \cup R^D_{id}) \cap (S' \times S')$. Then, M' simulates M.*

Proof. See [11] for a proof for systems with deterministic data transition constraints. See [10] for a general proof.

3 Model Checking Framework

Box (a) in Figure 3 shows an overview of our existing verification and validation framework for software specifications written in the formal specification language RSML^{-e} [24]. In this framework, we validate the behavioral aspect of system specifications using the execution environment in the NIMBUS Toolset [18] and then translate the system model into the NuSMV [22] input language using the built-in translator for verification purposes [9].

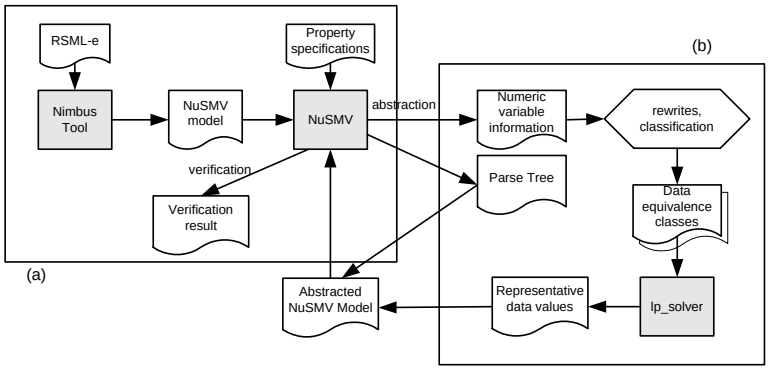

Fig. 3. Verification Framework

RSML^{-e} is a hierarchical, synchronous data-flow specification language that has similar language constructs to those of NuSMV, and thus, the translation is straightforward and we have been successfully using this framework for verifying interesting properties for industry specifications. Nevertheless, the straightforward translation of the data variables over large data domains often caused state-space explosion problem when model checking, giving us a compelling reason to incorporate an automated abstraction technique into the framework. To address this problem, we have implemented domain reduction abstraction as an extension to NuSMV. To give our work broad applicability, we chose to implement the abstraction as an extension to NuSMV as opposed to incorporating it to the NIMBUS Tool.

Box (b) in Figure 3 illustrates how the abstraction extensions interacts with our existing verification framework. We extract all information about numeric variables from the flattened system model in NuSMV after taking property specifications into account. We modified the NuSMV source code to add the domain abstraction functionality. By using the command-line option $-abs$, NuSMV initiates the abstraction process and generates an abstracted model (Figure 3).

Though the abstraction theory applies to both linear and non-linear cases, our current implementation is limited to systems with linear data conditions and constraints mainly due to the automation issues related to the selection of representative values; (1) the feasibility checking of a set of non-linear data conditions is an undecidable problem in general, and thus, requires a more reliable way of data selection method than using a constraint solver, and (2) our current termination condition for the minimal data trajectory computation is determined based on the Euclidean distance as described at the end of this section, which is not valid for general non-linear data constraints. Extensions to non-linear cases are to be investigated further.

Our implementation of the domain reduction abstraction consists of several sub-components: rewriting, classification, and selection.

Rewriting: Rewriting is to simplify complex numeric conditions into a form that can be comprehended by *lp_solve*. The current rewriting simplifies numeric conditions by (1) replacing aliases with actual expressions, (2) applying associativity and multiplication rules to convert expressions in a form of linear equations, and (3) eliminating division operations by multiplying left and right sides of the numeric condition with the (least) common multiples of denominators.

before rewriting :

$t\_dist := p\_dist + ((t\_alt - p\_alt\_upper) * 19/6076)$;
$a\_dist \geq t\_dist$;

after rewriting :
$a\_dist * 6076 \geq p\_dist * 6076 + t\_alt * 19 - p\_alt\_upper * 19$;

Our current rewriting process is limited to syntactic rewriting. We are looking for other rewriting options such as semantic rewriting which converts a set of numeric conditions into a simplified and semantically equivalent one. For example, the *Simplify()* function in Mathematica [1] can be used for semantic rewriting. Since there can be many semantically equivalent numeric conditions that cannot be eliminated by syntactic rewriting, semantic rewriting would help us reduce both the total number of conditions as well as the number of interrelated variables.

Classification and Generation of Data Equivalence Classes: For a given finite set of conditions $C = \{c \mid c := \alpha(v) \bowtie 0 \ , \ \alpha : D \rightarrow \Re\}$, a brute-force feasibility checking for possible data equivalence classes requires 2^n computations, where n is the number of elements in C. We can, however, classify the elements in C so that different classes do not share the same variables by using the relation

$$c_1 \sim c_2 \longleftrightarrow Var(\alpha(v)) \cap Var(\beta(v)) \neq \emptyset \ ,$$

where $Var(f)$ represents the set of variables with non-zero coefficient in function f.

$\sim$ defines an equivalence relation over C and partitions C into subclasses, say $C_1, C_2, \ldots, C_k$, and thus, the computation cost reduces to $2^{m_1} + 2^{m_2} + \ldots + 2^{m_k}$, where $m_i = \mid C_i \mid$ and $\Sigma_{i=1}^{k} = n$.

In our implementation, we first classify numeric conditions into subgroups using the equivalence relation. From each subgroup, we generate possible data equivalence classes by systematically alternating the boolean operators of the numeric conditions in the subgroup. *lp_solve* is then used to check the feasibility of the equivalence class during the selection of representative data values (see below).

The computational cost can be reduced further by identifying subsets of infeasible combinations of data conditions and by removing all data equivalence classes that subsumes the infeasible combinations. Currently, in our prototype tool, we simply check all possible combinations.

Selection of Representative Values: *lp_solve* serves two purposes; (1) checking the feasibility of each equivalence class, and (2) selecting a representative set of data values for each feasible equivalence class. We use the minimization capability of *lp_solve* to select representative values. If *lp_solve* cannot find a set of values that satisfies a given data equivalence class, we conclude that the equivalence class is infeasible.

When a subgroup of of numeric conditions does not contain constrained data variables, we simply select a representative set of values for each data equivalence class by minimizing a variable in the equivalence class. We initially imposed constraints to *lp_solve* to find integer solutions for all variables. Since integer programming is NP-hard — checking satisfiability is a NP-complete problem and finding optimized solutions is a NP-hard problem — whereas finding real-valued solutions is a polynomial time problem, we currently take an approach that finds real-valued solutions using *lp_solve* first and then checks for nearby integer solutions close to the real-valued solution. We do this by narrowing the domain of all variables to a small region $[x - \epsilon, x + \epsilon]$ (in our experiments, $\epsilon = 10$) around the real valued

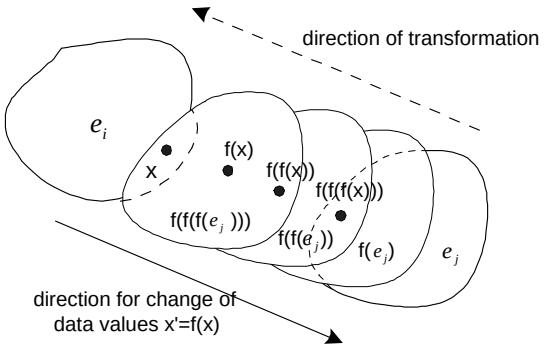

Fig. 4. Transformation operation on an equivalence class

solution x and then use *lp_solve* to look for an integer solution in this reduced domain. If we fail to find an integer solution around the real-valued solution, we finally let *lp_solve* search for an integer solution by imposing integer constraints for all variables over the whole domain. Even though this approach may perform redundant computations for some cases, our experiment shows a performance gain[4] in general.

When a subgroup of numeric conditions contains constrained data variables, the selection of representative data values is performed by minimal data trajectory computation as described next.

Minimal data trajectory computation: Automatic computation of minimal data trajectory is a non-trivial problem in general, especially when the data transition constraint function is complex. In this section, we briefly describe the automation approach for the minimal data trajectory computation under assumptions that both data conditions and constraints are linear;

1. Each data transition constraint function $f \in F_{E_i}$ for a variable v_k is in the form of $f(v_k) = av_k + b$, where $a \in \{0, 1\}$ and b is an integer constant.
2. Each numeric function $\alpha(v)$ appearing in the set of data conditions is linear.

These assumptions ensure each trace, a possible sequence of data equivalence classes that a data trajectory passes through, is finite. The finiteness of a trace is guaranteed since (1) the number of equivalence classes is finite, (2) the data values do not change signs periodically, and (3) the data conditions $\alpha(v)$ are not periodic. A detailed discussion with an algorithm can be found in [11].

Figure 4 describes the minimal data trajectory generation process. In order to find a minimal data trajectory from an equivalence class e_i to a class e_j, the linear transformation using the data transition constraints $x' = f(x)$ is applied to the equivalence class e_j repeatedly until it reaches e_i. The minimal data trajectory is generated from a minimal data node x in the intersection region $e_i \wedge f^n(e_j)$. We perform the computation for every possible permutation of data equivalence classes.

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

[4] One example shows the reduction of computation time from 7 minutes to a few seconds.

	1	2	3	4	5	6	7	8	9	10
system	spec size	smv size	# BDD & ADD nodes	# numeric conditions	# constrained	time for abs	# lp_solver	# BDD & ADD nodes	verification time	memory
ASW	> 590	384	692,179	9 (19)	0	63 s	1024	18,974	1.11 s	5 M
FGS	3832	2953	1,013,026	25 (30)	9	11 s	55	183,331	5,561 s	46.8 M
FMS	3366	1839	.	92 (26)	4	< 1.3 hour	131,174	129,591	5.38 s	8.8 M

Fig. 5. Performance Data

and e_j respectively, is used as a stopping condition in each n_{th} step of the transformation. The computation terminates if the distance grows.

4 Application Results

We have applied the prototype abstraction tool to three industry problems in the domain of aircraft control systems: a version of a Flight Guidance Systems (FGS) and a version of a Flight Management System (FMS) from Rockwell/Collins Advanced Technology Center. A hypothetical (although realistic) Altitude Switch System (ASW) we have used in deviation analysis [17] is also used as an example. All tests were performed on 800 Mhz Linux machine with 512 M memory. All verification data is a result of using default dynamic variable reordering and the cone of influence reduction in NuSMV.

Figure 5 shows the performance data on the three avionics systems. Columns 1 through 5 describe the characteristics of the system models; columns 1 and 2 show the size of the specification and the size of the NuSMV model in lines of codes, column 3 shows the number of BDD and ADD nodes after parsing the model and encoding variables in NuSMV, in column 4, $x(y)$ represents the number of numeric variables, x, and the number of numeric conditions, y, in the model, and column 5 shows the number of numeric variables with constrained data transitions.

Before applying the abstraction technique, NuSMV failed to check a property even for the smallest system, the ASW. NuSMV required 171 M for the BDD encoding of variables and was not able to finish model checking a property in an hour. For the FMS model, NuSMV fails to encode variables to BDDs returning an error message even on a larger machine with 1.5 G memory.

Columns 6 to 10 in Figure 5 show the result of the domain reduction abstraction: the time used for abstraction (column 6), the number of calls to *lp_solve* (column 7), the number of BDD and ADD nodes after abstraction (column 8), the time for generating a counter example using NuSMV (column 9), and the memory usage for the counter example generation (column 10). The abstraction time for the ASW is 63 second. After the abstraction, NuSMV was able to check the same property we attempted before within 2 seconds. For the FGS, the time for abstraction was 11 seconds and the time for counter example generation was about 1.5 hour. The high verification cost for the FGS is mainly due to its high complexity even without data variables; it is a relatively large system where variables are heavily inter-connected. The FMS, which has the most complex numeric conditions among the three examples, required almost 1 hour and 15 minutes for the abstraction and about 6 seconds for generating a counter example.

Note that the abstraction cost for the FGS is the lowest even though it has a larger number of numeric variables and/or numeric conditions than the other systems. This is mainly because the abstraction performance depends on the number of inter-related numeric

variables and conditions, not on the overall number of numeric variables and conditions; the most complex subgroup of the FGS after classification has 4 inter-related conditions over 3 numeric variables whereas that of the ASW has 10 inter-related numeric conditions over 7 numeric variables and that of the FMS has 17 inter-related numeric conditions over 12 numeric variables.

The abstraction of the FMS is the most costly mainly due to the high complexity of the numeric conditions – 17 inter-related conditions over 12 variables. Our current brute-force feasibility checking is one major source of the performance degradation which can be avoided with a better implementation; after inserting monitoring code in the current implementation, we observed that 125,520 out of 131,174 calls to *lp_solve* for the FMS were with unsatisfiable sets of data conditions. This number of calls can be reduced significantly if we identify infeasible sets of data conditions up-front. For example, if we identify 2 out of 17 data conditions in a set are inconsistent, we can avoid 2^{15} calls to *lp_solve*. We are looking to using an efficient decision procedure to identify the obviously infeasible sets of data conditions.

5 Discussion

We have presented application results of domain reduction abstraction on some industrial size problems. The results show dramatic reduction of the size of the models and this reduction made the use of model checking feasible. The abstraction process is automated and requires no user intervention. Counter examples generated from the abstracted model are straight-forward to understand, and thus, there is no need for interpretation. In addition, since domain abstraction is orthogonal to other abstraction techniques, we can apply other existing techniques [3, 6–8, 13–15] in concert with domain reduction abstraction if necessary.

There are other automated abstraction techniques which can be effective for abstracting numeric variables and conditions, such as syntactic transformation [21] and predicate abstraction [4, 19, 15]. Nevertheless, these approaches have no guarantee of termination in general and may not be cost-effective in our verification domain; a domain where hundreds of properties need to be checked over a system, since the abstraction process is required for each property to be verified. On the contrary, domain reduction abstraction is one-time abstraction that can be reused unless some numeric conditions are changed. Even when some numeric conditions are changed, we need to apply the abstraction only to those subgroups affected by the change. When large number of properties need to be verified[5], our approach could be more cost effective.

Though we believe our technique is promising, there are many remaining areas that need improvement. First, we need to investigate a better way of eliminating infeasible data equivalence classes as we pointed out in the previous section. Second, although the cost of minimal data trajectory computation was negligible in our case examples since most of the constrained data variables were one dimensional timers, we expect that minimal data trajectory computation can be quite expensive, especially when it involves a large number of inter-related numeric conditions. We would like to collect more data on industry problems to assess the cost-effectiveness. Lastly, the automation is implemented for a limited type of data conditions and constraints. We plan to expand the capability to non-linear conditions and limited non-deterministic constraints.

[5] For example, researchers at Rockwell-Collins Inc. have identified 293 properties for the Flight Guidance System and verified them all in batch mode using NIMBUS and NuSMV.

References

1. Mathematica. http://www.wolfram.com/products/mathematica/index.html.
2. Mixed integer/linear programming tool lp_solve version 3.1. ftp://ftp.ics.ele.tue.nl/pub/lp_solve/.
3. 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.
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. Ramesh Bharadwaj and Constance Heitmeyer. Model checking complete requirements specifications using abstraction. In *First ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.
7. Tevfik Bultan, Richard A. Gerber, and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Computer Aided Verification*. Springer Varlag, 1997.
8. William Chan, Richard Anderson, Paul Beame, and David Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Proc. of CAV'97, LNCS 1254*, pages 316–327. Springer, June 1997.
9. Yunja Choi and Mats Heimdahl. Model checking RSML^{-e}requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, October 2002.
10. Yunja Choi, Mats P.E. Heimdahl, and Sanjai Rayadurgam. Domain reduction abstraction. Technical Report 02-013. University of Minnesota, April 2002.
11. 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.
12. 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.
13. 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.
14. 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.
15. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the Computer Aided Verification(CAV 1997)*, 1997.
16. 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.
17. Mats P.E. Heimdahl, Yunja Choi, and Mike Whalen. Deviation analysis via model checking. In *International Conference on Automatied Software Engineering*, September 2002.
18. Mats P.E. Heimdahl, Jeffrey M. Thompson, and Michael W. Whalen. Executing state-based specifications in a heterogeneous environment. Technical Report TR 98-029, University of Minnesota, Department of Computer Science, Minneapolis, MN, 1998.
19. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Rupak Majumdar. Lazy abstraction. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, January 2002.
20. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer Aided Verification*, July 2002.
21. Kedar S. Namjoshi and Robert P. Kurshan. Syntatic program transformations for automatic abstraction. In *12th International Conference, CAV2000*, pages 435–449, July 2000.
22. NuSMV: A New Symbolic Model Checking. Available at http://http://nusmv.irst.itc.it/.
23. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.
24. Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.