Abstraction based Automated Test Generation from Formal Tabular Requirements Specifications

Renzo Degiovanni[1], Pablo Ponzio[1], Nazareno Aguirre[1], and Marcelo Frias[2]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto and CONICET, Río Cuarto, Córdoba, Argentina.
E-mail: {rdegiovanni,pponzio,naguirre}@dc.exa.unrc.edu.ar
[2] Departamento de Ingeniería Informática, Instituto Tecnológico Buenos Aires and CONICET, Buenos Aires, Argentina. E-mail: mfrias@itba.edu.ar

Abstract. We propose an automated approach for generating tests from formal tabular requirements specifications, such as SCR specifications. The technique is based on counterexample guided abstraction refinement and the use of SMT solving. Moreover, in order to effectively perform automated test generation, we take advantage of particular characteristics of tabular requirements descriptions to aid the abstraction and abstraction refinement processes. The exploited characteristics are, most notably, the organisation of the requirements specification in *modes*, which is used to build an initial abstraction, and the execution model of tabular specifications, which is directed by changes observed in environment variables and is exploited for modularising the transition relation associated with tables, simplifying the calculation of abstractions. These characteristics enable us to effectively perform automated test generation achieving good levels of coverage for different criteria relevant to this context.
We compare our approach with a standard abstraction analysis, showing the benefits that exploiting the mentioned characteristics of tables provide. We also compare the approach with model checking based generation, using several model checking tools. Our experiments show that the presented approach is able to generate test cases from models whose complexity, with respect to the sizes of variables and data domains, cannot be coped with well by the model checkers we used.

1 Introduction

It is generally accepted that the quality of requirements specifications has a great impact in the whole development process, since crucial activities such as validation against user expectations, system verification against requirements, and the coherence of the requirements (and therefore also the system to be developed), depend on these specifications. Requirements specifications are mostly expressed in natural language, in textual form. Various approaches deal with this informal textual representation, and how its quality can be improved and assessed. However, *formal* requirements specifications may provide useful features, difficult to

achieve if informal notations are used. More precisely, formally specified requirements are better suited for analysis and its automation, which can be exploited for (semi-)automatically finding problems in the specification itself (inconsistencies, imprecisions, etc.), and contrasting the specification against the system (i.e., verification), or user expectations (i.e., validation).

Tabular notations, originally used to document requirements by D. Parnas and others [14], have proved to be a useful means for concisely and formally describing expressions characterising complex requirements. Indeed, tables have been successfully incorporated into various formalisms for requirements specification, most notably those reported in [17, 12]. Tables are used for formally describing various *relations* involved in requirements, such as the expected relationship between the observed and controlled environment variables once the system is put in place, assumptions about the environment due to conditions external to the system, or the relations that the system must maintain with the environment. Tables impose a *structure* on specifications, which provides useful features: it helps in organising large and complex formulas into well distinguished smaller formulas that are easier to follow, and impose simple constraints for guaranteeing characteristics such as certain forms of completeness (no missing cases) and consistency (no contradicting requirements).

In this paper, we are concerned with the automated analysis of formal tabular notations. More precisely, we propose the use of automated analysis techniques for generating *tests* based on requirements specifications. The approach we use for test generation is a variant of a lazy abstraction mechanism for automated analysis, and relies on scalability mechanisms that take advantage of the tabular structure of the specifications, i.e., exploiting characteristics inherent to the tabular specifications. This variant yields, in this context, performance enhancements, compared to standard lazy abstraction. It is automated, based on counterexample guided abstraction refinement and the use of SMT solving. Tests from tabular specifications correspond to *executions* of the requirements specification, i.e., sequences of events and states respecting the constraints prescribed by tables. These tests have an obvious role in the validation, i.e. contrasting with user expectations, and verification, i.e., contrasting against system behaviour. Moreover, they can also help in identifying problems in the specification (e.g., missing cases, contradictory statements, etc.). Especially for complex requirements, coming up with a test suite that "exercises" the specification under a good variety of cases (i.e., complying with particular *coverage* criteria) can be extremely difficult.

Especially for validation and verification, activities in which the engineer needs to compare the behaviours expected by users and the actual system behaviour against the formal specification of requirements, it is useful that the tests (i.e., the execution traces over the specification) maintain the level of detail of the original specification. Usually, due to analysis reasons, the engineer needs to manually simplify the specifications, so that analysis tools can deal with these; if test cases are generated from these simplified models, then they have to be "concretised" before contrasting them with the expected or actual

system behaviours, in order to disregard spurious tests resulting from the abstraction. This obviously complicates the validation and verification tasks. Our approach attempts to deal with specifications at their original level of abstraction, generating an abstraction level suitable for test generation, but producing (non spurious) test cases at the level of abstraction of the original specification. For some case studies we are able to generate test cases which several model checking tools are unable to cope with, in particular due to the sizes of variables and data domains of the models.

Our approach is based on the following few observations regarding formal tabular requirements specifications:

- In tabular requirements specifications, a special set of variables, known as *mode classes*, are employed in order to organise the states of the system into *modes*; mode changes are described via corresponding (mode) transition tables.
- The execution model of tabular specifications is directed by changes observed in environment variables, whose alterations are observed once at a time.
- The definition of variables involved in a tabular specification lead to a dependency relation which is inherently *acyclic*. Each symbol describes a different specification element, thus ruling out aliasing in specifications.
- Tabular requirements specifications often involve numeric variables, whose ranges are often larger than what automated tools (e.g., model checkers) are able to handle.

Contributions of this paper. The contribution of this paper is an approach for automatically generating *tests* based on formal tabular requirements specifications. The approach is automated, based on *abstraction* and counterexample guided abstraction refinement. The underlying technology supporting the approach is SMT solving, but the actual benefits come from the identification of the above elements, inherent to this kind of specifications, and mechanisms to exploit them in order to contribute to the abstraction. The technique exploits the above described characteristics of tables in the following way:

- *Modes* and mode transitions are employed as part of the *initial* abstraction of the specification. This provides important analysis benefits, as we will demonstrate later on, especially because both properties to be analysed and the structure of tables in the specification typically strongly depend on modes. We also observe that states within a mode tend to coincide in the level of "preciseness" for analysis, which we exploit to define our variant of *lazy abstraction*.
- The execution model of tabular specifications and the inherent acyclic dependence of syntactic elements in tabular requirements specifications are exploited for modularising the transition relation associated with tables according to monitored variables changes, and their dependencies. This allows us to speed up the calculation of abstractions, as well as reuse calculations.
- We define a technique for dealing with discrete numeric datatypes in abstracting tabular specifications, so that the degree of detail that is necessary

to incorporate as part of the abstraction refinement in relation to numerical variables is "localised" to parts of the abstract state space. As we show later on, this leads to better scalability when dealing with numerical variables in tabular specifications.

The above ways of exploiting tabular specifications significantly contributes to the abstraction process. We show the benefits of our approach by assessing it, in comparison with standard abstraction, for various case studies. We also use these case studies to compare the technique with several model checking tools, used for test case generation from requirements specifications. The technique is able to generate test cases from models whose complexity, with respect to the sizes of variables and data domains, cannot be handled well by the model checkers we used.

It is important to notice that our approach does not constitute a testing criterion. Our technique is a *test generation mechanism*, which, provided a test criterion, allows for the automated generation of test cases according to the test criterion. We refer the reader to [9] for a thorough description of testing strategies applied to tabular specifications.

Related work. Tabular notations, in particular SCR, have associated tool support providing different kinds of analysis, e.g., syntax checking, consistency analysis via theorem proving and model checking, and the verification of properties of requirements [2, 6, 13]. With respect to testing, the simulator in the SCR toolset [13] allows the developer to load specific scenarios, which are in principle provided by the engineer, and check whether certain associated assertions are violated or not in the particular executions described by the scenarios. Gargantini and Heitmeyer [11] used a model checker for automatically obtaining tests (table executions) transiting through particular *modes*. Recently, Fraser and Gargantini [10] made a thorough comparison between various model checkers (symbolic, bounded, explicit state, etc.) in order to automatically generate test cases from tables, and analysed the achieved coverage and scalability issues. Our evaluation of the technique is based on case studies analysed by Fraser and Gargantini, and our comparison with model checkers uses Fraser and Gargantini's employment of model checkers for test case generation from tables. Bultan and Heitmeyer [6] recently employed the ALV infinite state model checker to analyse tables; we do not report results using ALV for test case generation because the tool is discontinued, and in our preliminary experiments with it other model checkers exhibited better performance for this task.

To our knowledge, automated abstraction techniques have been applied to tabular requirements specifications infrequently, particularly for testing purposes. In [4] Bharadwaj and Heitmeyer applied abstraction to SCR specifications, for scalability purposes related to model checking. As opposed to our work, Bharadwaj and Heitmeyer's approach is based on the removal of irrelevant variables (slicing), and the transformation of "internal" variables into input ones (i.e., monitored), with the aim of removing monitored variables; given a property to be verified, their abstraction is *fixed*, it does not admit refinement.

Many successful approaches to verification and test generation have been proposed. Some of these are based on SMT solving, abstraction, combinations and variants. Most works target *code analysis* rather than requirements specifications. In particular, lazy abstraction with abstraction refinement based on interpolation was used for automatically generating tests leading to the reachable locations of a program, with successful applications in device drivers and security critical programs [3]. Other related and successful approaches are reported in [16, 7]. Our approach is based on that presented in [3], which employs lazy abstraction [15] for test generation, but targets requirements specifications. Requirements specifications are not "control intensive", as the programming domains in which abstraction is successfully applied [8], thus constituting a novel interesting domain. The behavioural model corresponding to SCR specifications has a significant degree of nondeterminism, leading to high levels of "interleavings", which makes it difficult to apply techniques that work well in control intensive environments. Other automated tools such as Pex [20] and JPF [21] successfully implement automated white box test case generation for .NET and Java programs, respectively. These target code, and are based on symbolic execution instead of predicate abstraction with automated refinement.

2 Preliminaries

Tabular Requirements Specifications. Tables provide an unambiguous yet clear and concise way of writing formulas. This has been found to be particularly useful for describing complex requirements of software systems, most notably in the work of D. Parnas and collaborators [14], and in the SCR method [17, 12]. Our presentation is based on SCR because it is a very good representative of formal requirements languages using tables, which has extensive tool support and many real specifications to assess analysis techniques. In the SCR methodology, requirements are specified following Parnas' four-variable model [19]. In this context, tables are used, among other things, for describing **REQ**, the requirements as a relationship that the system should induce between *monitored* and *controlled* variables, environment variables that the system is able to observe and control, respectively. In order to describe this relationship, SCR uses *events*, *conditions*, *mode classes* and *terms*. Events occur when changes in the variables observed by the system take place, and conditions are logical expressions referring to these variables. Modes represent classes of states of the system (the values of particular variables called *mode classes*), and typically capture historical information of the system [4]; terms are functions on the variables of the specification. For describing events, SCR provides a simple notation. The notation `@T(c) WHEN d` describes the event in which expression `c` becomes true, when `d` is `true` in the current state, i.e., it represents the expression $\texttt{c'} \wedge \texttt{c} \wedge \texttt{d}$, where the primed expression refers to the next state. If `d` is true, the '`WHEN`' section is not written.

Let us consider an example, taken from [4]. Suppose that one needs to specify a safety injection system, whose task is to control the level of water pressure

Old mode	Event	New mode
TooLow	@T(mWaterPres >= Low)	Permitted
Permitted	@T(mWaterPres >= Permit)	High
Permitted	@T(mWaterPres < Low)	TooLow
High	@T(mWaterPres < Permit)	Permitted

(a) Mode class `mcPressure`

mode	Condition	
High, Permitted	True	False
TooLow	tOverridden	NOT tOverridden
cSafetyInjection	Off	On

(b) Controlled var. `cSafetyInjection`

mode	Events	
High	Never	@F(mcPressure=High)
TooLow, Permitted	@T(mBlock=On) WHEN mReset=Off	@T(mcPressure=High) OR @T(mReset=On)
tOverridden'	True	False

(c) Term `tOverridden`

Fig. 1. Tabular specification of the Safety Injection System

of a nuclear plant's cooling system. The system monitors the water pressure, and a couple of switches for blocking and resetting (represented by monitored variables `mWaterPres`, `mBlock` and `mReset`, respectively), and it controls a single boolean variable, indicating whether the safety injection system is on or off (controlled variable `cSafetyInjection`). Usually the behaviour of the system depends on a number of previous events or conditions (i.e., the history); these are characterised by the so called *modes* of the system (possible values of *mode classes*). In this case, a single mode class, `mcPressure`, whose corresponding modes are `TooLow`, `Permitted` and `High`, indicates whether the water pressure is considered to be too low (below a constant `Low`), in a permitted level or high (over a constant `Permit`), respectively. Basically, the system must start safety injection when the pressure becomes too low. The system can be "disengaged" via the switches, indicating that its actions are overridden. This is captured by a term `tOverridden`. Tables are used to define dependent symbols, i.e., mode classes, terms and controlled variables. Basically, tables are of two kinds: event tables, to define symbols whose changes are driven by the occurrences of events, and condition tables, which define symbols in terms of conditions over other symbols. Mode changes depend on events, and are described via special event tables, the *mode transition tables*. For the safety injection system, table in Figure 1(a) indicates how system modes change when certain events occur. For instance, when the system is in mode `TooLow` and the water pressure becomes higher than or equal to `Low`, the mode changes to `Permitted` (see first row of the mode transition table). Term `tOverridden` is defined via the event table in Figure 1(c). This table indicates exactly when the system actions are overridden (e.g., if the block switch is pressed while in any mode other than `High`, with the reset switch off). Finally, the controlled variable `cSafetyInjection` is defined via a condition table, shown in Figure 1(b). Tables have associated well formedness conditions. For instance, the disjunction of all conditions in a row of a condition table must be `True`, to ensure completeness (no missing cases), and the conjunction of any pair of different cells in the same row must be `False` (disjointness), to ensure no contradictions.

A finite run is a sequence $\sigma = s_0, s_1, \ldots, s_k$ of states such that s_0 satisfies some specified initial condition, and every state s_{i+1} is obtained from its predecessor s_i by modifying a single monitored variable mV, and propagating the modifications of all variables depending on mV, according to what is prescribed by the corresponding tables. Variables not depending on mV maintain their corresponding values in s_i. The change in a monitored variable that triggers the move from a state to another can be interpreted as an *input event*, while the resulting changes in controlled variables can be interpreted as the *output*. Thus, a run is a sequence of input events and corresponding outputs. For example, consider the state $s = \langle \texttt{High}, 150, \texttt{on}, \texttt{off}, \mathit{True}, \texttt{off} \rangle$, corresponding to the values of `mcPressure`, `mWaterPres`, `mBlock`, `mReset`, `tOverridden` and `cSafetyInjection`, respectively; from state s, the input event `mWaterPres' = mWaterPres-1` takes the system to the state $\langle \texttt{Permitted}, 149, \texttt{on}, \texttt{off}, \mathit{False}, \texttt{off} \rangle$, assuming that $\texttt{Permit} = 150$.

Abstraction. A *labelled transition system* (LTS) $\mathcal{S} = (S, \Sigma, \rightarrow)$ consists of a set S of states, a finite set Σ of labels, and a labelled transition relation $\rightarrow \subseteq S \times \Sigma \times S$. A *region structure* $\mathcal{R} = (R, \bot, \sqcup, \sqcap, \mathit{pre}, \mathit{post}, [.])$ for an LTS $\mathcal{S}$ is a structure consisting of a set R of regions, where each region r represents a set $[r]$[3] of states of S; $\bot$ represents the empty set of states, $r \sqcup r'$ and $r \sqcap r'$ are the union and intersection operators of $[r]$ and $[r']$, respectively; $\mathit{pre}(r, l)$ and $\mathit{post}(r, l)$ are the weakest precondition and strongest postcondition operators with respect to labels (e.g., $\mathit{pre}(r, l)$ returns the largest region such that, from all its states and traversing arcs labelled by l one arrives at states in r), respectively. We denote by $r \subseteq r'$ the fact that $[r] \subseteq [r']$, and by $r \equiv r'$ that $[r] = [r']$. An abstraction structure $\mathcal{A} = (\mathcal{R}, \mathit{pre}^{\mathcal{A}}, \mathit{post}^{\mathcal{A}}, \trianglelefteq)$ for an LTS $\mathcal{S}$ consists of a region structure $\mathcal{R}$ for $\mathcal{S}$, abstract pre and post operators $\mathit{pre}^{\mathcal{A}}$ and $\mathit{post}^{\mathcal{A}}$, and a precision preorder $\trianglelefteq$, such that $\mathit{pre}(r, l) \subseteq \mathit{pre}^{\mathcal{A}}(r, l)$, $\mathit{post}(r, l) \subseteq \mathit{post}^{\mathcal{A}}(r, l)$, and $\mathit{pre}^{\mathcal{A}}$ and $\mathit{post}^{\mathcal{A}}$ are monotonic with respect to $(\trianglelefteq \cap \subseteq)$. A region r can be thought of as representing an abstract state, in some abstract state space, whose concretisation is $[r]$, with the abstract operators $\mathit{pre}^{\mathcal{A}}$ and $\mathit{post}^{\mathcal{A}}$ enabling abstract state propagations. The precision preorder $\trianglelefteq$ indicates how close the abstract pre and post operators are to the concrete ones, for given regions.

A particular way of abstracting a state space is via *predicate abstraction*. In this context, regions are characterised by sets of state properties called *support predicates*, and the concretisation is simply the set of states satisfying the corresponding predicates. More precisely, let $\mathcal{S} = (S, \Sigma, \rightarrow)$ be an LTS, and P a set of predicates over S (i.e., for every $p \in P$, $[p] \subseteq S$). An abstraction structure $A_P(\mathcal{S}) = (\mathcal{R}_P(\mathcal{S}), \mathit{pre}^{A_P}, \mathit{post}^{A_P}, \trianglelefteq_P)$ can be defined, as follows:

- $\mathcal{R}_P(\mathcal{S}) = (R, \bot, \sqcup, \sqcap, \mathit{pre}, \mathit{post}, [.])$, where the regions in R are pairs (φ, Γ), with $\Gamma \subseteq P$ being a finite set of (local) *support predicates*, and φ is a boolean formula over the predicates of Γ. The remaining elements of $\mathcal{R}_P(\mathcal{S})$ are defined as follows: $\bot = (\mathit{false}, \emptyset)$; $(\varphi, \Gamma) \sqcup (\varphi', \Gamma') = (\varphi \vee \varphi', \Gamma \cup \Gamma')$; $(\varphi, \Gamma) \sqcap (\varphi', \Gamma') = (\varphi \wedge \varphi', \Gamma \cup \Gamma')$; $\mathit{pre}((\varphi, \Gamma), l) = (\varphi_l^{pre}, \Gamma_{ls})$ where φ_l^{pre} is the

[3] $[.] : R \rightarrow 2^S$ is the function that maps each region to the set of states it represents.

weakest precondition of φ with respect to l and Γ_{ls} is the least superset of Γ which contains all predicates in φ_l^{pre}. Operator *post* is defined in a similar way. The concretisation $[(\varphi, \Gamma)]$ of (φ, Γ) is defined as $[\varphi]$ (the set of states satisfying φ).

- The abstract operator $post^{A_P}$ is defined as follows: let (φ, Γ) be a region with $\varphi = \varphi_1 \vee \cdots \vee \varphi_k$ in DNF (with support predicates as atomic formulas), and l be a label. $post^{A_P}((\varphi, \Gamma), l)$ is the disjunction $\psi_1 \vee \psi_2 \vee \cdots \vee \psi_k$, where each ψ_i is a conjunction of all literals γ appearing positively or negatively in Γ, and such that $\varphi_i \Rightarrow pre(\gamma, l)$. The operator pre^{A_P} is defined in a similar way.
- $\trianglelefteq$ is defined in the following way: $(\varphi, \Gamma) \trianglelefteq (\varphi', \Gamma')$ iff $\Gamma \supseteq \Gamma'$.

This characterisation corresponds to *lazy predicate abstraction*, as introduced in [15]. The process is called *lazy* because the abstraction predicates are *local* to the regions, enabling the refinement of the portions of the abstract computation tree that require it. The lazy predicate abstraction algorithm is a symbolic forward search algorithm with the capability of refining the abstract regions as needed.

Given an abstraction structure $\mathcal{A}$, an initial region r_0 and an error region ε, the procedure tries to verify that ε is not reachable in the abstract model by constructing an abstract reachability tree. Each node can be in one of two states: *unmarked* (not yet treated) or *marked* (already treated). The algorithm starts by creating an unmarked initial node with region r_0. It then iterates, by taking an unmarked node n, and doing one of the following steps, depending on the characteristics of n: *(i)* if the region of n intersects ε, then the abstract error region is reachable; *(ii)* if n's region has already been covered, i.e., it is included in the join of marked nodes, then n is marked too; *(iii)* if n's region does not intersect ε and has not been covered, then n is marked, the abstract successors of n's region, with respect to $post^A$ and all labels, are calculated, and new unmarked nodes are created to hold these regions.

The algorithm terminates, in principle, when the error region is reached, or no unmarked nodes remain. When the error region is reached, i.e., an abstract trace leading to an error is found, it remains to check whether the counterexample is spurious or not. This is done by checking the feasibility of the abstract trace. If it is feasible, a real counterexample has been found; if not, then a new suitable predicate can be added to one of the regions to refine the abstraction, removing the spurious counterexample. The abstract computation subtree with the refined region as a root has to be recalculated, and the process can continue. If the formalism to express the predicates is chosen appropriately, both the feasibility of the abstract trace and the calculation of a suitable predicate to refine the abstraction can be done automatically.

A way of computing suitable predicates to add to the regions in the above abstraction setting is via the use of *interpolation* [18]. Let us suppose that $\sigma = \varphi_1, \varphi_2, \ldots, \varphi_k$ is a sequence of formulas such that their conjunction is inconsistent (e.g., these formulas might correspond to an abstract counterexample trace). An *interpolant* for σ is a sequence of formulas $\psi_0, \psi_1, \ldots, \psi_k$ such that: *(a)* $\psi_0 = True$ and $\psi_k = False$, *(b)* $\forall 1 \leq i \leq k \cdot \psi_{i-1} \wedge \varphi_i \Rightarrow \psi_i$, and *(c)* $\forall 1 \leq i \leq k \cdot vars(\psi_i) \subseteq$

$vars(\varphi_1, \ldots, \varphi_i) \cap vars(\varphi_{i+1}, \ldots, \varphi_k)$. Given an abstract spurious trace σ, the interpolants provide suitable predicates to add to each of the regions in the trace so that σ is ruled out as an abstract behaviour.

3 Abstracting Requirements Specifications

We now define an abstraction for tabular requirements specifications. A main hypothesis in our construction is that modes structure the tabular specifications in a way that is relevant to many properties of interest on tables, particularly those having to do with test case generation. Thus, exploiting this structure for abstraction may provide significant benefits. Moreover, we base the construction on *lazy* abstraction motivated by the fact that, when the system is in different modes it generally has different capabilities, i.e., it responds to different input events and in different ways; so, quite possibly different kinds and levels of precision might be necessary when the system is in different modes. Since we want to make the process automated, we choose linear integer arithmetic (LIA), one of the formalisms behind the SMT solver MathSAT [5] which we use as a decision process for calculating and refining abstractions, as our abstract domain. This language, which due to space restrictions we do not describe here, is expressive enough for our needs. For the interpolants, we use difference logic, a sublanguage of LIA for which MathSAT is able to calculate interpolants automatically. This is essential for making the abstraction refinement process completely automated.

Let us start describing the construction. First, an SCR requirements specification *Spec* corresponds to an LTS $\mathcal{S}_{Spec} = (S, E, \rightarrow_T)$, where S is the set of all possible values for variables in the specification, E is the set of events on monitored variables (i.e., input events), and $\rightarrow_T$ contains a tuple (s_i, e, s_j) iff s_j is obtained from s_i as a consequence of input event e and the propagation of changes to all variables according to what is prescribed by the tables in *Spec* [4]. The set R of regions for our abstract domain is $M_{Spec} \times LIA \times \wp(LIA)$, with M_{Spec} being the set of modes in *Spec*, and LIA and $\wp(LIA)$ the domains of formulas and sets of formulas in LIA, respectively. The concretisation of a region is defined as follows: $[(m, \varphi, \Gamma)] = \{s \in S | s \models \varphi$ and m is the mode of $s\}$. The formula φ must be a conjunction of literals based on predicates from Γ as atomic formulas. As it can be noticed in the definition of R, the current mode of the system is an essential part of a region, meaning that the abstraction is precise with respect to "mode location". If the specification has more than one mode class, M_{Spec} would correspond to the cartesian product of mode classes. The remaining elements of our region structure $\mathcal{R}_{Spec}$ are defined in a standard way, from $\mathcal{S}_{Spec}$.

As we explained previously, in a lazy setting the support predicates are *local to regions*. In our case, and for the reasons explained at the beginning of this section, the support predicates will be made *local to a mode*. That is, all regions that share the mode will also share the support predicates. Given a mode m, we will denote by $SP(m)$ the current set of support predicates for mode m.

In order to complete the definition of our abstract domain $\mathcal{A}_{Spec}$, we have to define the abstract operators $pre^{\mathcal{A}_{Spec}}$ and $post^{\mathcal{A}_{Spec}}$, and the precision preorder $\trianglelefteq_{Spec}$. Since the approach is based on a *forward* construction, we concentrate on the definition of $post^{\mathcal{A}_{Spec}}$ ($pre^{\mathcal{A}_{Spec}}$ is defined similarly). Our abstract operator $post^{\mathcal{A}_{Spec}}$ is defined as follows: for every region (m, φ, Γ) and input event l, $post^{\mathcal{A}_{Spec}}((m, \varphi, \Gamma), l)$ is the set of all tuples (m', ψ, Γ'), such that:

- m' is reachable from m via l,
- $\Gamma' = SP(m')$, and
- ψ is the conjunction $\gamma_1 \wedge \cdots \wedge \gamma_k$, where each γ_i is a literal appearing positively or negatively in Γ', and $\varphi \Rightarrow pre(\gamma_i, l)$.

Finally, the precision preorder $\trianglelefteq_{Spec}$ is defined in the following way: for every pair of regions (m, φ, Γ) and (m', φ', Γ'), $(m, \varphi, \Gamma) \trianglelefteq_{Spec} (m', \varphi', \Gamma')$ iff $\Gamma' \supseteq \Gamma$.

The proofs of $pre((m, \varphi, \Gamma), l) \subseteq pre^{\mathcal{A}_{Spec}}((m, \varphi, \Gamma), l)$, $post((m, \varphi, \Gamma), l) \subseteq post^{\mathcal{A}_{Spec}}((m, \varphi, \Gamma), l)$, and the monotonicity of $pre^{\mathcal{A}_{Spec}}$ and $post^{\mathcal{A}_{Spec}}$ with respect to $(\trianglelefteq_{Spec} \cap \subseteq)$, are relatively straightforward.

Let us provide a very simple example illustrating the above definition of $post^{\mathcal{A}_{Spec}}$. Consider the specification of the safety injection system given previously. Suppose that, for each of the three modes, the following are the current support predicates:

- `TooLow`: `tOverridden`, `mWaterPres < Low`
- `Permitted`: `tOverridden`, `mWaterPres < Low`, `mBlock = on`
- `High`: `tOverridden`, `mWaterPres >= Low`, `mBlock = on`, `mReset = on`

Consider the abstract region $r = \langle$`Permitted`, *True*, *False*, *False*$\rangle$. Its successor with respect to $post^{\mathcal{A}_{Spec}}$ and the event `mWaterPres' = mWaterPres-1` is the disjunction of r itself, and $\langle$`TooLow`, *True*, *True*$\rangle$. Similarly, the successor of r with respect to the event `mWaterPres' = mWaterPres+1` is the disjunction of r, $\langle$`High`, *False*, *True*, *False*, *False*$\rangle$ and $\langle$`High`, *False*, *True*, *False*, *True*$\rangle$.

This algorithm works essentially in the same way as the lazy abstraction algorithm explained previously, but using our abstract strongest postcondition operator. More precisely, given a property of interest P whose reachability needs to be analysed, the process starts with the initial abstract state, with only P as a support predicate; it then starts calculating the abstract reachability tree using $post^{\mathcal{A}_{Spec}}$, trying to reach an abstract state satisfying P and refining the regions via interpolants resulting from spurious counterexamples. Whenever a new predicate is added to the support predicates of a region r, the same predicate is incorporated to all regions of nodes whose modes coincide with that of r; similarly, when a new region is incorporated, this region "inherits" the support predicates of its corresponding mode, as the definition of $post^{\mathcal{A}_{Spec}}$ indicates.

When no unmarked nodes remain and P has not been reached, then P is unreachable. If a concrete trace reaching a state satisfying P is constructed, a concrete run witnessing the reachability of P is obtained. The process can however "diverge" when the number of support predicates for some region becomes too large to be dealt with.

As opposed to the original lazy abstraction approach, our abstraction is precise with respect to mode location, and that support predicates are local to

modes instead of regions. Experimenting with tabular specifications, we found out that as support predicates are discovered, these tend to be shared (i.e., would be "rediscovered" by the original algorithm) by abstract states with the same mode. Adding these predicates to all regions with the same mode enabled us to improve the construction of the abstraction. This fact is related to the "shape" of the LTS corresponding to a tabular specification, in which every state admits changes in any of the monitored variables (i.e., the LTS is not "control intensive", as opposed to some applications of abstraction for automated analysis [8]). Another important difference of our approach has to do with the mechanism for checking whether a state is covered or not. While the standard lazy abstraction algorithm keeps the value of the already computed symbolic abstract space (the join of all the already computed regions) and a decision procedure is employed to check if the current node is included there, our approach looks at all the other nodes within the same mode, to see if there is another one weaker than the current. This last check can be done without the use of a decision procedure, since all the nodes with the same mode share the same predicates.

Modularising the Transition Relation. Given a (concrete) state of the system and an event, the transition relation leads to a process for computing the next state: when a monitored variable changes, the tables are looked up to update other variables whose values depend on the change of the monitored variable. The modularity of the transition relation defined by the tabular structure, together with the variable dependencies and the absence of aliasing, can be straightforwardly used to "localise" changes, and save time in the calculation of the next state. We can do something similar for the case of computing the abstract successors of an abstract state. However, taking into account the structure of our abstraction states, we modularise the transition relation in a different way. Basically, we take the global state transition relation T, as defined by the tables, and for each monitored variable mV, we produce transition (sub)relations $T_{mV}^{m_1}, \ldots, T_{mV}^{m_k}$, where $m_1, \ldots, m_k$ are the modes of the specification. Each $T_{mV}^{m_i}$ corresponds to the transition subrelation associated with the behaviour of the tables when the monitored variable that changes is mV, and the current mode is m_i. This modularisation is straightforward to obtain from the tables, and is similar to a kind of "cone of influence" approach. The acyclicity of the dependency between symbols of the specification and the absence of aliasing enables us to perform this modularisation of the transition relation easily. As we show in the following section, this modularisation provides an important benefit with respect to the calculation of abstract successors, since besides the simplifications in the calculations of concrete weakest preconditions (notice that these are necessary for computing abstract successors), it enables us to identify predicates whose current boolean value will not be altered, since none of its associated syntactic elements depends on the modified variable that changed. The next section provides an evaluation of the benefits of this modularisation.

Treating Numerical Domains. A characteristic that our abstraction process is sensitive to is the use of large numerical domains in the models. These numeri-

cal domains are rather common in SCR requirements specifications, so we need to propose a mechanism to deal with these. Basically, the problem has to do with our lazy abstraction having support predicates local to modes. This means that whenever a location s needs to incorporate a support predicate (because of abstraction refinement), that predicate is added to all locations with the same mode as s. Suppose that the behaviours of the specification require the system to be within a mode m along long "chains" of successive numerical values for some variable before producing a change to a mode m'. What would typically happen is that the abstraction refinement process will need to introduce support predicates for distinguishing all these successive values of the variable (in our case, these support predicates will be introduced by the interpolation process), in order to remove spurious counterexamples taking the system from m to m'. All these support predicates will be associated with a single mode (m), making our abstraction process impractical. In order to deal with this problem, we use a heuristics that consists of introducing *intervals* over these numerical variables. Basically, we learn from short executions of the abstraction process which numerical variables potentially have the issue we just described. We then take these variables, partition their domains in a number of intervals, and make support predicates to be local to mode, and corresponding domain interval. The actual degree used to partition intervals is calculated from the size of the numerical domain being partitioned, and the maximum number of support predicates for a location; notice that locations corresponded to modes, whereas now they will correspond to a mode, and intervals for the numerical variables that require partitioning. Consider, for instance, the safety injection system described previously. Suppose also that variable `mWaterPres`'s range is [0..5000], and it changes in steps of 1..10 (decrementing or incrementing a value in this range). The refinement based on interpolation will then introduce support predicates $\texttt{mWaterPress} \leq 0$, $\texttt{mWaterPress} \leq 10$, $\texttt{mWaterPress} \leq 20$, and so on. So, if we do not want to introduce more than 20 support predicates (per location) associated with `mWaterPres`, then we would introduce intervals $0 \leq \texttt{mWaterPres} \leq 199$, $200 \leq \texttt{mWaterPres} \leq 399$, $400 \leq \texttt{mWaterPres} \leq 599$, and so on, to define the new, finer locations. That is, these intervals are now part of the locations, meaning that abstraction refinement will be local to a mode and interval of `mWaterPres`.

Generating Test Cases from Tables using Abstraction. Test case generation via predicate abstraction is performed in a similar way as the generation using model checking, as reported, e.g., in [3]. First, one needs to build all test predicates corresponding to the coverage criterion of interest. Each of these test predicates characterises a particular equivalence class of test cases, in the corresponding test criterion; these are used as "trap properties", one at a time, for the abstraction algorithm to produce concrete traces reaching the predicates.

For each test predicate P, we run the algorithm and obtain three possible outputs:

- the abstract state space is covered without reaching P, meaning that the corresponding test case equivalence class is infeasible, or

– a concrete run reaching P is produced (i.e., a test case), or
– the process does not converge, and is stopped due to timeout, exhausted memory or any other resource related problem (e.g., excessive introduction of support predicates). In this case the process is inconclusive, and the corresponding test predicate is marked as an *error*, as these are called in [10].

We perform the test case generation using two stages. First, we use a so called *cartesian abstraction*, in which different but related abstract states are combined into a single representation; this is equivalent to having three possible values for support predicates: `true`, `false` or `*`, the latter meaning "don't care". In this way the number of states that need to be handled for the test case generation, but the interpolation based refinement might fail (since we are not dealing with actual abstract runs, but sets of abstract runs, when using these cartesian abstract states). Whenever the interpolation process produces, for a given mode, a support predicate that has already been introduced previously, we stop the test generation process for the current test predicate, and move to a second stage, in which the abstraction is "precise" (i.e., non cartesian), for this test predicate (the remaining test predicates will be treated first with cartesian abstraction, and then precise abstraction if necessary). This latter process, the precise abstraction, has no issues regarding abstraction refinement, but its scalability diminishes.

Reusing Calls to the Decision Procedure. Given a particular test criterion, the corresponding test predicates are in general related in some way to each other. For instance, in *table coverage* predicates correspond to cells of tables; two different predicates originating in the same table might, for instance, share a row, meaning that they coincide in the mode, or share a column, meaning that the resulting value is the same, or if in different columns, in the same table, the satisfaction of one of these implies the unsatisfiability of the other, and vice versa. So, the predicates discovered while covering a particular test predicate P might also be relevant for the covering of other test predicates of the same test criterion. For this reason, we "cache" the calls to the decision procedure while computing abstract successors in the covering of a test predicate, so that these calculations can be reused in the covering of other test predicates within the same test criterion. This resulted to be fruitful, as we show in the next section.

4 Experimental Results

The contents of our experimental results section are two fold. First, we provide an evaluation of our approach in comparison with standard predicate abstraction, which enables us to assess the benefits of our variant that exploits characteristics of tables. Second, we compare our approach with automated test case generation based on model checking. The experiments are based on some of the case studies presented in [10], where a thorough comparison between different model checkers used for test generation is carried out. Therein, a number of test criteria, such as *table coverage* and *modified condition decision coverage*, relevant to tabular specifications are used. We generate test cases for the same criteria, and refer

the reader to [10] for a description of these. The case studies in [10], some of which we also use in this paper, are models available in the literature, regarding a cruise control system (`ccs`), a safety injection system (`sis`), an aircraft's autopilot (`autopilot`), and a car overtaking protocol for coordinating smart vehicles (`car3prop`). Our models differ, for part of the experimentation, from the ones used in [10]; for assessing the benefits that exploiting characteristics of tables provides, compared with lazy abstraction, we use essentially the same models as in [10], which have been manually simplified (by using smaller constants and numeric ranges, mostly). However, in order to compare our approach with model checking test generation we use versions of the case studies which are larger than those in [10], i.e., where the sizes of constants and numeric ranges are larger (basically, at the level of abstraction of the textual descriptions of the corresponding systems). For instance, `autopilot reduced` deals with integer variables in the range [0..500], while the original `autopilot`, which we use for comparison with model checkers, has these same variables over the range [0..10000]. As we mentioned previously, we are interested in this because, especially for validation and verification, it is important to generate test cases at the same level of abstraction expected by the user, and/or used in the implementation. All the experiments were run on an 2.6GHz Intel Core 2 Duo with 3GB of RAM (2.5GB maximum memory set for the analysis tools), running GNU/Linux 2.6.

The following table compares, for three case studies (`sis`, `car3prop` and `autopilot`), standard predicate abstraction (PA, i.e., support predicates are globally shared by regions, modes are not part of the initial abstraction, no modularisation of the transition relation with respect to monitored variables/modes), with PA plus support predicates local to modes and these considered in the initial abstraction (PA + m.), PA plus modularised transition relation with respect to monitored variables/modes (PA + mt.), and finally our approach (PA + m. + mt.). The data corresponds to the total number of test predicates for a test criterion, the number of runs of each technique, and the corresponding numbers of **c**overed (i.e., those for which a test case was generated), **i**nfeasible (those that the corresponding technique identified as unrealisable), and **u**ncovered test predicates (i.e., errors, those in which the corresponding technique is inconclusive). For the covered test predicates, we also indicate between parentheses the number of traces, since a single trace can cover several test predicates. When any individual run of any of the processes executed for over an hour, it was stopped and the corresponding test predicate marked as uncovered. Notice that we marked the models `autopilot` and `sis` as "reduced", meaning that they have been manually simplified with respect to the original description, by using small constants and numerical ranges. These models are the same as those used in [10] (`car3prop` is not reduced because it has no numerical constants or ranges).

CS	car3prop (498 TPs.)			sis reduced (91 TPs.)			autopilot reduced (409 TPs.)		
	runs	c/i/u	time	runs	c/i/u	time	runs	c/i/u	time
PA	110	402(14)/90/**6**	33620	19	80(4)/11/**0**	8496	21	395(7)/10/**4**	23497
PA + m.	114	401(17)/96/**1**	8858	21	80(10)/11/**0**	5319	81	347(19)/10/**52**	113401
PA + mt.	118	402(22)/74/**22**	69197	20	80(9)/11/**0**	13032	51	389(31)/10/**10**	22201
PA + m. + mt.	119	402(29)/96/**0**	1795	23	80(12)/11/**0**	4724	54	399(44)/10/**0**	3951

In our approach, support predicates are local to modes instead of regions. We argued that, due to the structure of tables, regions with the same mode tend to share the support predicates. In order to validate this hypothesis, we have randomly chosen a few test predicates, and gathered the number of savings in predicate discovery associated with the "support predicates local to regions" approach. Basically, we measure for standard lazy abstraction and our approach, the number of nodes visited and introduced support predicates, and for standard lazy abstraction also the support predicate with largest number of "rediscoveries" for different regions with the same mode. These results are summarised in the following table.

CS	Lazy			Lazy + m. + mt.	
	nodes	predicates	most repeated	nodes	predicates
car3prop	41	94	30	76	11
car3prop	48	75	33	75	8
sis reduced	916	158	50	936	65
sis reduced	113	25	8	113	17
autopilot reduced	1037	126	21	1192	32
autopilot reduced	965	126	21	1120	32

Models involving numeric variables over large ranges, such as `sis`, are those in which "support predicates local to regions" provides more benefits.

We now compare our technique, referred to in the tables as "Lazy abs. +", with model checkers used for test generation. Our experiments are based on those presented in [10], so we compare our technique with Spin, NuSMV, Cadence SMV and SAL. The four case studies used for the assessment are the ones mentioned above, using larger more realistic constants and numeric ranges, compared to the models used in [10]. We have run these tools with a variety of settings, and we report the best result obtained for each tool, in the table at the end of this section. When a tool is not mentioned for one of the case studies, that is because it performed significantly worse than other model checkers. As opposed to the previous experiments, and because we have increased the sizes of the models, we set the timeout for covering single test predicates to 3 hours, and the total analysis time for a test criterion to 2 days. We report the number of individual runs for each technique, the covered, uncovered (nor covered, nor identified as infeasible, referred to as errors) and infeasible test predicates, the total time for the generation (in seconds), and the largest trace produced. For our technique, we also mention the number of automated refinements that were necessary. As our experimental results show, our technique is able to deal with the generation in cases in which model checkers fail to do so. Let us explain our assessment of the experiments reported in the table at the end of this section. Notice that, whenever Spin is able to generate a test case, it does so very fast, but being an explicit state model checker, it generally runs out of memory quickly for models with large numerical domains. In specifications such that `car3prop` and `sis`, with either small ranges for integer variables or relatively few interleavings (due to having a small number of monitored variables), model checkers perform very well, being able to generate test cases much faster than our technique. On the other hand, for specifications such as `ccs` and `autopilot`, with large ranges for numerical variables and several monitored variables (leading to a higher degree

of interleaving), model checkers perform poorly and our technique shows better profit.

	Runs	Tests			Time	Max. Trace	Rfnmts.
		Cov.	Errors	Infeas.			
autopilot (409 test predicates)							
Spin	169	288(48)	121	0	2351	168	-
NuSMV	409	0	409	0	timeout	-	-
Cad. SMV	378	36(5)	373	0	timeout	13	-
SAL/SMC	296	116(3)	293	0	timeout	6	-
Lazy abs. +	62	390(43)	10	9	164003	127	7099
ccs (582 test predicates)							
Spin	582	0	582	0	timeout	-	-
Cad. SMV	582	0	582	0	timeout	-	-
Lazy abs. +	120	494(32)	0	88	312	8	167
car3prop (498 test predicates)							
NuSMV	142	402(46)	0	96	261.45	13	-
Cad. SMV	160	402(64)	0	96	77.58	10	-
SAL/BMC	133	402(37)	15	81	56.32	11	-
Lazy abs. +	125	402(29)	0	96	1795	9	1713
sis (91 test predicates)							
Spin	19	80(8)	0	11	145.72	23515	-
NuSMV	27	80(16)	0	11	995.23	406	-
Cad. SMV	31	80(20)	0	11	420.34	402	-
SAL/SMC	27	80(16)	0	11	37.94	403	-
Lazy abs. +	24	80(13)	0	11	32742	402	2955

5 Conclusion and Future Work

We have identified a number of characteristics inherent to formal tabular requirements specifications that can be exploited for improving automated analysis for test generation from these specifications. Indeed, we argued that certain features of these specifications can be exploited for defining an abstraction process able to effectively generate test cases, i.e., runs of the specification, corresponding to a variety of test criteria relevant to tables. Basically, the identified characteristics have to do with typical elements used by the engineer in the construction of the formal specification, and the inherent behavioural model of tables. We have compared our developed approach with test case generation from tables using some model checkers, as well as with a standard abstraction approach not exploiting the identified characteristics of tables. Our experimental results show that the identified characteristics play a significant role in the scalability of abstraction employed in test case generation from tables, and that the resulting technique is able to deal better with some tabular specifications which, due to their complexity with respect to the size of numerical ranges and constants present in the model, and not handled well by some model checkers. The motivation for dealing with "larger" specifications is straightforward: it contributes to scalability in this analysis, and facilitates the validation and verification activities, for which it is important that the generated tests maintain the level of abstraction/detail expected by users, and present in the implementation. Although abstraction has been successfully applied for test case generation via model checking, it has generally been applied in "control intensive" domains [8]. In contrast, requirements specifications are not control intensive, thus constituting a challenging domain to apply abstraction for test case generation. As our experiments show, directly

applying lazy abstraction for test case generation in the context of requirements specifications does not perform well, so our variant, exploiting characteristics of tables, shows its profit.

As future work, we plan to explore the use of terms for the abstraction, complementing our current use of mode classes; this is motivated by the fact that, according to [6], terms typically capture historical information, as modes do. We also plan to apply the presented approach for the verification of specifications, i.e., to guaranteeing the properties associated with the well formedness of tables, as well as the verification of state and transition invariants over tabular specifications. We are also extending the ideas presented in this paper to the framework for describing behaviours over tabular specifications presented in [1].

Acknowledgements

The authors would like to thank Angelo Gargantini, who kindly provided the model checking specifications of the SCR models we used in our experiments, as well as a prototype tool to automate test generation from tables using various model checkers. This greatly simplified our experimental evaluation. We would also like to thank the anonymous referees for their helpful comments.

This work was partially supported by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grant PICT 2006 No. 2484. The third author's participation was also supported through ANPCyT grant PICT PAE 2007 No. 2772.

References

1. N. Aguirre, M. Frias, M. Moscato, T. Maibaum and A. Wassyng, *Describing and Analyzing Behaviours over Tabular Specifications Using (Dyn)Alloy*, in Proc. of FASE 2009, LNCS 5503, Springer, 2009.
2. J. Atlee and J. Gannon, *State-Based Model Checking of Event-Driven System Requirements*, IEEE Trans. Software Eng. 19(1), IEEE Press, 1993.
3. D. Beyer, A. Chlipala, T. Henzinger, R. Jhala and R. Majumdar: *Generating Tests from Counterexamples*, in Proc. of ICSE 2004, IEEE, 2004.
4. R. Bharadwaj and C. Heitmeyer, *Model Checking Complete Requirements Specifications Using Abstraction*, Automated Software Engineering 6(1), Springer, 1999.
5. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio and R. Sebastiani, *The MathSAT 4 SMT Solver*, in Proc. of CAV 2008, LNCS 5123, Springer, 2008.
6. T. Bultan and C. Heitmeyer, *Applying Infinite State Model Checking and Other Analysis Techniques to Tabular Requirements Specifications of Safety-Critical Systems*, Design Automation for Embedded Systems, 12(1-2), 2008.
7. S. Chaki, E. Clarke, A. Groce, S. Jha and H. Veith. *Modular Verification of Software Components in C*, Trans. on Software Engineering 30(6), IEEE, 2004.
8. E. Clarke, A. Gupta, H. Jain and H. Veith, *Model Checking: Back and Forth between Hardware and Software*, in Verified Software: Theories, Tools, Experiments, LNCS 4171, Springer, 2008.
9. X. Feng, D. Parnas, T. Tse and T. O'Callahan, *A Comparison of Tabular Expression-Based Testing Strategies*, IEEE Transactions on Software Engineering (to appear). Also available at `http://www.cs.hku.hk/research/techreps/document/TR-2009-19.pdf`.

10. G. Fraser and A. Gargantini, *An Evaluation of Model Checkers for Specification Based Test Case Generation*, in Proc. of ICST 2009, LNCS, Springer, 2009.
11. A. Gargantini and C. Heitmeyer, *Using Model Checking to Generate Tests from Requirements Specifications*, in Proc. of ESEC/FSE 1999, LNCS, Springer, 1999.
12. C. Heitmeyer, R. Jeffords and B. Labaw, *Automated consistency checking of requirements specifications*, Trans. on Soft. Eng. and Methodology, 5(3), ACM, 1996.
13. C. Heitmeyer, M. Archer, R. Bharadwaj and R. Jeffords, *Tools for constructing requirements specifications: the SCR Toolset at the age of nine*, Computer Systems: Science & Engineering, 20(1), 2005.
14. K. Heninger, J. Kallander, D. Parnas and J. Shore, *Software Requirements for the A-7E Aircraft*, NLR Memorandum Report 3876, US Naval Research Lab., 1978.
15. T. Henzinger, R. Jhala, R. Majumdar and G. Sutre, *Lazy abstraction*, in Proc. of POPL 2002, ACM, 2002.
16. T. Henzinger, R. Jhala, R. Majumdar and K. McMillan, *Abstractions from proofs* in in Proc. of POPL 2004, LNCS, Springer, 2004.
17. N. Leveson, M. Heimdahl, H. Hildreth and J. Reese, *Requirements Specifications for Process-Control Systems*, Trans. on Software Engineering, 20(9), IEEE, 1994.
18. K. McMillan, *Interpolation and SAT-Based Model Checking*, in Proceedings of the 15th International Conference on Computer Aided Verification CAV 2003, LNCS 2725, Springer, 2003.
19. D. Parnas and J. Madey, *Functional Documentation for Computer Systems*, Science of Computer Programming, 25(1), Elsevier, 1995.
20. N. Tillmann and J. Halleux, *Pex-White Box Test Generation for .NET*. In Proc. of TAP 2008, LNCS, Springer, 2008.
21. W. Visser, C. Păsăreanu and S. Khurshid, *Test input generation with Java PathFinder*, in Proc. of ISSTA 2004, ACM, 2004.