SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–44
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Improving Lazy Abstraction for SCR Specifications through Constraint Relaxation

Renzo Degiovanni[1,3]*, Pablo Ponzio[1,3], Nazareno Aguirre[1,3] and Marcelo Frias[2,3]

[1]*Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Argentina.*
[2]*Departamento de Ingeniería Informática, Instituto Tecnológico Buenos Aires, Buenos Aires, Argentina.*
[3]*Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina.*

SUMMARY

Formal requirements specifications, e.g. SCR (Software Cost Reduction) specifications, are challenging to analyze using automated techniques such as model checking. Since such specifications are meant to capture *requirements*, they tend to refer to real-world magnitudes often characterized through variables over large domains. At the same time, they feature a high degree of non determinism, as opposed to other analysis contexts such as (sequential) program verification. This makes model checking of SCR specifications difficult even for symbolic approaches. Moreover, automated abstraction refinement techniques such as counterexample guided abstraction refinement fail in many cases in this context, since the concrete state space is typically large, and reaching specific states of interest may require complex executions involving many different states, causing these approaches to perform many abstraction refinements, and making them ineffective in practice.

In this paper, an approach to tackle the above situation, through a two-stage abstraction, is presented. The specification is first relaxed, by disregarding the constraints imposed in the specification by physical laws or by the environment, before being fed to a CEGAR (Counterexample Guided Abstraction Refinement) procedure, tailored to SCR. By relaxing the original specification, shorter spurious counterexamples are produced, favouring the abstraction refinement through the introduction of fewer abstraction predicates. Then, when a counterexample is concretizable with respect to the relaxed (concrete) specification but it is spurious with respect to the original specification, an efficient though incomplete refinement step is applied to the constraints, to cause the removal of the spurious case.

This approach is experimentally assessed, comparing it with related techniques in the verification of properties and in automated test case generation, using various SCR specifications drawn from the literature as case studies. The experiments show that this new approach runs faster and scales better to larger, more complex specifications than related techniques. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

*Correspondence to: Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto. Ruta Nac. No. 36 Km. 601, Río Cuarto (5800), Argentina. E-mail: rdegiovanni@dc.exa.unrc.edu.ar

Prepared using stvrauth.cls *[Version: 2010/05/13 v2.00]*

1. INTRODUCTION

The requirements process, consisting of thoroughly describing the features that software artifacts must possess, is recognized as an important phase in the development of quality software [23, 37, 48]. This phase often involves some requirements specification framework or language specifically designed for the task, whose usage leads to better requirements elicitation, e.g., by exposing imprecisions in the description of software features, missing cases, and even contradictions among different requirements. Thus, a main benefit of a rigorous requirements process lies in its potential to expose errors at early stages of software development, when these are easier and less expensive to correct.

Various approaches aim at aiding in eliciting, describing and organizing requirements, most of which are based on informal notations (e.g., the approaches by Stevens et al. [47] an Maiden and Alexander [42]). Some approaches, on the other hand, are based on *formal* notations, leading to inherently unambiguous specifications, better suited for automated or semi-automated analyses. Software Cost Reduction (SCR) [32, 27, 25] is a formal methodology for describing software requirements. SCR's distinctive feature is its *tabular* notation, that allows one to organize large and unstructured requirements into smaller and well structured tables, yielding more modular requirements specifications that are easier to understand and mantain. SCR is useful to describe the interactions between a software system and its operating environment, as well as the particular features imposed by the nature of the environment itself. SCR has been successfully used for eliciting the requirements of many safety critical applications, including an aircraft's operational flight program [32], a submarine's communications system [26], the control software of a nuclear power plant [50], among others [9, 13, 39]. It has also been used as part of a process for developing human-centric decision systems [30], and as part of an approach to the development of trustworthy autonomous systems [31], and to derive event-based transition systems from goal oriented requirements models [40]. In addition, several tools supporting automated analysis of SCR specifications have been developed, most notably the toolset introduced by Heitmeyer et al. [29], which performs type-checking, consistency checking (no contradictory requirements, no missing cases, etc.), simulation of user provided scenarios, and other analysis tasks. Two particularly difficult analyses regarding SCR specifications are *test case generation*, i.e., producing executions of the system from the specification of the requirements, and the verification of properties of the specification. The former is very important for contrasting the expected behaviour of the system, as specified by the formal requirements, and the actual behaviour of the system once it is implemented (it is in fact an instance of the so called *model based testing* approach). The latter is very useful as a way of checking properties that are expected to emerge as a consequence of the requirements. For both these analyses, a typical mechanism employed in the context of SCR specifications is *model checking*, including some rather sophisticated approaches (e.g., those introduced by Bultan and Heitmeyer [12]).

A main limitation in automated analyses is the *state explosion problem*: automated analyses are at least polynomial on the size of the state space of a specification, and such size grows exponentially with the complexity of the specification (e.g, increasing number of variables or the size of their domains, increasing number of components in a specification). In requirements specifications, there is typically a need to refer to real-world magnitudes, which are often formally

Prepared using stvrauth.cls
DOI: 10.1002/stvr

characterized through variables over large domains, increasing the size of the state space. Also, since they correspond to early phases of software development, they often feature a high degree of non determinism, compared to other more concrete descriptions, such as designs and source code. Thus, model checking requirements specifications is a challenging task, even for symbolic approaches. Approaches that use *abstraction* as a way of tackling state explosion, by producing more abstract versions of a specification that are better suited for analysis, have also been employed for requirements analysis. However, these mechanisms also suffer from known limitations. Some approaches require *manual* abstractions [29, 22]. Heitmeyer et al. [29] perform ad hoc abstractions on specifications, while Gargantini and Heitmeyer [22] propose manually shrinking the domains of numeric variables before analysis. Ad hoc abstractions require careful observation of the specification and the property to be verified, or the goal to be reached in the case of test case generation, by an experienced engineer, a time consuming and expensive activity; on the other hand, proving a property true or reaching a goal in a specification with manually reduced domains does not guarantee that the property holds or the goal is actually reached in the original specification. This is an important problem, in particular in relation to test case generation, where the produced test cases are used to contrast expected behaviour with the actual system behaviour; when these scenarios come from manually abstracted specifications, they will need to be (manually) concretized in order to be contrastable with the system.

Techniques that automatically refine abstractions for analysis, e.g., Counterexample Guided Abstraction Refinement (CEGAR), also have difficulties to be applied in the domain of requirements specifications, since the concrete state space in such specifications is typically large, and reaching specific states of interest may require complex executions involving many different states, causing these approaches to perform many abstraction refinements, and making them ineffective in practice.

In this paper, an approach to tackle the above situation, through a two-stage abstraction, is presented. The specification is first relaxed, by disregarding the constraints imposed in the specification by physical laws or by the environment, before being fed to a CEGAR procedure, tailored to SCR. Essentially, the first stage is motivated by the observation that, by disregarding environmental constraints imposed on monitored (numerical) variables of SCR requirements specifications, one obtains a *relaxed* specification, adequate for the verification of properties that do not need to precisely track the validity of such environmental constraints. At the same time, by relaxing the original specification, shorter spurious counterexamples are produced, favouring the abstraction refinement through the introduction of fewer abstraction predicates. The second stage is a standard CEGAR procedure [33], designed to exploit inherent characteristics of SCR specifications such as the division of the state space in mode classes, to achieve better performance. It is complete with respect to the relaxed specifications it is fed with. However, concretizable counterexamples may still be spurious with respect to the original specification. These receive a *lightweight* treatment, that only allows it to remove spurious cases when infeasibility can be blamed on atomic transitions, through the addition of transition invariants that cause their removal, constituting an *incomplete* refinement of the relaxed specification.

The presented technique is evaluated for verifying properties of requirements specifications and generating tests from specifications, on a number of case studies taken from the literature. These experiments show that this technique outperforms previous approaches (including previous work by authors of this article [19], by orders of magnitude). Moreover, these cases show that indeed many

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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

(a) Mode class mcPressure

Modes	Events	
High	Never	@F(mcPressure=High)
TooLow, Permitted	@T(mBlock=On) when mReset=Off	@T(mcPressure=High) OR @T(mReset=On)
tOverridden	True	False

(b) Term tOverridden

Modes	Conditions	
High, Permitted	True	False
TooLow	tOverridden	NOT tOverridden
cSafetyInjection	Off	On

(c) Controlled var. cSafetyInjection

Figure 1. Tabular specification of the Safety Injection System

properties do not need to precisely track the validity of environmental constraints, witnessed by the very low number of refinements required on the relaxed specifications, and that the lightweight refinement of the relaxed specification is powerful enough to handle the assessed specifications, not falling into its source of incompleteness in any of these cases.

The remainder of this article is organized as follows. In section 2 the SCR method is discussed, as well as the motivation for this work. In Section 3 the lazy abstraction approach for the analysis of SCR specifications is presented, which serves as a core of the whole analysis approach, described in Section 4. Later on, in Section 5, the experimental evaluation is presented. Finally, Section 6 discusses related work, while the conclusions are presented in Section 7.

2. SOFTWARE COST REDUCTION

Software Cost Reduction (SCR) is a method and a language for describing software requirements [28]. A distinctive feature of SCR is its *tabular* notation for describing requirements, resulting in specifications that are more modular, and easier to understand and maintain. SCR is a formal language with a precisely defined semantics. An SCR requirements specification describes how the system must interact with its environment in order to accomplish its desired goal, e.g., ensuring that some property always holds in the environment. In SCR, the system is conceived as an actuator that reads the values of some quantities of interest from the environment via its *monitored* (input) variables, and based on such inputs dictates which values the system must produce on its *controlled* (output) variables, which in turn produce changes in output devices to (indirectly) alter the environment to accomplish the system's goal. From the point of view of the system, the environment nondeterministically triggers input events, via (feasible) changes in the monitored variables. The system is aware of the environmental alterations associated to input events through changes in the monitored variables, and constantly reacts to these alterations producing values for controlled variables, to mantain the intended properties in the environment. Formally, the requirements are described in SCR by means of a mathematical relation among the system's monitored and controlled variables, called **REQ**, defining, for fixed values of monitored variables, the accepted values for the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

system's controlled variables. Furthermore, SCR allows one to describe constraints of the system's environment by means of another (mathematical) relation, called **NAT**.

As an example, that will be used as a running case study throughout this article, consider the Safety Injection System (SIS) [17] and whose SCR specification is depicted in Figure 1. SIS is in charge of partially controlling a nuclear power plant by tracking the water pressure of a cooling subsystem via a monitored variable `mWaterPres`, and the state of two user-controlled switches, one that prevents the system from being engaged (monitored variable `mBlock`), and another that reactivates the system after being blocked (monitored variable `mReset`). SIS requirements (relation **REQ**) should state that when the water pressure level is too low (dangerous), the system must administer a "safety injection" that takes the water pressure back to a normal level. Thus, SIS sets controlled variable `cSafetyInjection` to true to indicate that a safety injection must be applied. Constraints on the SIS environment (relation **NAT**) may state, for instance, that due to physical laws the water pressure levels can vary between 0 and 5000, and that the sensoring equipment guarantees that the difference between two consecutive measurements of the water pressure level cannot be greater than 10.

SCR also introduces the notions of *modes* and *mode classes*. A *mode class* is, essentially, an internal controlled variable that allows the system to maintain state information, whose possible values are the *modes*. For example, SIS defines mode class `mcPressure`, which introduces modes `TooLow`, `Permitted` and `High`. These modes indicate that water pressure levels are currently too low (dangerous), normal and high, respectively.

SCR specifications are *typed*, variables can only take values from their corresponding typesets. A mode class' typeset is the set of the modes it introduces. For the above example, mode class `mcPressure` defines modes `TooLow`, `Permitted` and `High`; monitored variables `mBlock` and `mReset` are of type $\{$`On`, `Off`$\}$, and `mWaterPres` is an integer ranging from `0` to `5000`; controlled variable `cSafetyInjection` is boolean. SCR also features the definition of constants; in SIS, constants `Low` (`900`) and `Permit` (`4000`) represent the thresholds for entering modes `TooLow` and `High`, respectively.

The syntax adopted for basic events in SCR is `@T(cond)`, where `cond` is a logical predicate. Its semantics is that the system (atomically) transitions from a state where `cond` does not hold to a state satisfying `cond`, i.e., it intuitively reads as "`cond` becomes true". For example, `@T(mWaterPress >= Low)` (see the first row in Figure 1(a)) states that the water pressure level just became greater than or equal to `Low` (it was less than `Low` in the previous state). Expression `@F(cond)` is the dual of `@T(cond)` ("`cond`" just became false). The events that occur in the system environment and affect the values of monitored variables are called *input events*. They drive the execution of the system (i.e., the system reacts to input events).

The relation that governs the intended behaviour of the system, namely **REQ**, is captured in SCR through a set of tables. Tables can be either *mode transition*, *event* or *condition* tables. A *mode transition table* describes how a system's mode class changes in response to events. SIS' mode transition table, corresponding to mode class `mcPressure`, is shown in Figure 1(a). For example, the first row states that when the system is in mode `TooLow`, and monitored variable `mWaterPres` is increased reaching or exceeding `Low` (i.e., input event `@T(mWaterPres >= Low)` occurs), then the system transitions into mode `Permitted`. An *event table* defines how a variable (other than a mode class) changes its value in response to events. The only *event table* for SIS is shown

Prepared using stvrauth.cls
DOI: 10.1002/stvr

in Figure 1(b). It defines a *term* (a kind of auxiliary internal variable of the specification) called `tOverridden`. `tOverriden` is true when the system actions are overridden, i.e., when the block switch is pressed and the reset switch is not, while the mode is not `High`. Figure 1(b) shows that `tOverriden` takes the value `True` (row 3, column 2) when the mode is `TooLow` or `Permitted` (row 2, column 1) and event `@T(mBlock=On) when mReset=Off` occurs (row 2, column 2). This last expression is an example of a *conditioned event*, which is triggered only when `mBlock` becomes `On` while `mReset` is `Off`. Finally, a *condition table* defines a variable or term as a function of other variables in the current state of the system. The condition table of SIS, shown in Figure 1(c), defines the controlled variable `cSafetyInjection` in terms of the values of `mcPermitted` and `tOverriden`. For example, the table states that the safety injection must be applied (`cSafetyInjection` is `On` in row 3, column 3) when the system is in mode `TooLow` (row 2, column 1) and the system actions are not overridden (row 2, column 3).

2.1. SCR semantics

In this section the standard SCR semantics is briefly discussed. The reader is referred to the work of Heitmeyer et al. [29] for further details. In section 3.1 a slightly different presentation of this SCR semantics, that fits better the introduction of the lazy predicate abstraction algorithm for SCR analysis, is provided. Throughout this section a fixed SCR specification *Spec* is assumed.

Formally, the *system* modelled by *Spec* is represented by a labelled transition system (LTS) $\Sigma_{Spec} = (S, S_0, E, T)$, where S is the (finite) set of system states, $S_0 \subseteq S$ are the initial states, and E are the input events that it observes from its environment. The transition relation $T \subseteq S \times E \times S$ is defined by the SCR tables and the **NAT** relation of *Spec*. If the system is in state $s \in S$ and input event $e \in E$ is triggered, T indicates how to construct the new system state s' (the system response to e). Due to the constraints imposed on SCR tables, T is *deterministic*, i.e., given a source state s and an event e, SCR tables and **NAT** constraints deterministically define $s' = T(s, e)$. In addition, T is *partial*, since not all the input events are reacted to in all states. Moreover, SCR tables define a dependency relation D between entities (xDy iff y is involved in the table defining x), which must be a partial order.

There are some assumptions related to the occurrence of input events in the environment. First, input events are triggered *nondeterministically* in the environment. At any given state, any enabled input event might occur. An event is *enabled* if it is allowed to occur at the current state, according to its definition, and satisfies **NAT**. For instance, assuming that **NAT** states that sensoring intervals guarantee that `mWaterPres` cannot change more than 10 units in two consecutive measurements, `@T(mWaterPres>900)` is only enabled in states in which `mWaterPres` $\in$ [891..900] (`@T(c)` represents the fact that `c` becomes true in the state, i.e., `c` must be false for the event to be enabled). Second, it is assumed that input events are triggered one at a time. In other words, exactly one input event occurs at each system transition. This is often called the *One-Input Assumption* in the literature [28].

Let us be more precise about the state space of the system. As mentioned before, there are four kinds of entities in SCR: mode classes, terms, monitored and controlled variables. Each entity has an associated datatype, a finite set of values that the entity can take. Function TY is defined to map each entity to its corresponding datatype. In the case of a mode class m, TY maps it to $\{M_1, \ldots, M_n\}$, the set of modes it defines. A *state* $s \in S$ is a total function mapping each entity x to a value in

Prepared using stvrauth.cls
DOI: 10.1002/stvr

$$T_{\texttt{cSafetyInjection}}(\texttt{mcPressure}, \texttt{tOverriden}) =$$
$$\begin{cases} \texttt{Off} & \text{if } \texttt{mcPressure} = \texttt{High} \vee \texttt{mcPressure} = \texttt{Permitted} \vee \\ & \quad (\texttt{mcPressure} = \texttt{TooLow} \wedge \texttt{tOverriden}) \\ \texttt{On} & \text{if } \texttt{mcPressure} = \texttt{TooLow} \wedge \neg\texttt{tOverriden} \end{cases}$$

Figure 2. `cSafetyInjection`'s table function

$TY(x)$; $s(x)$ denotes x's value in s. Since only finite datatypes are allowed in SCR, S is a finite set. Notice that, as each entity must have a unique value in a given state s, each mode class m must have a *unique* mode $MD_m(s)$ in s (when the system has only one mode class, the subscript m in MD_m) is dropped. Without loss of generality, it is assumed that every SCR specification *Spec* defines a single mode class.

2.1.1. Tables. Intuitively, each SCR table T_x defines a dependent entity r_x; when an input event occurs, T_x describes the value of r_x in either the current state or the next state (the latter denoted by r'_x). More precisely, if $r_1, \ldots, r_k$ are all the dependencies of r_x (i.e., $(r_x, r_j) \in D$ for $1 \leq j \leq k$), then T_x defines a total function $F_x : TY(r_1) \times \ldots \times TY(r_k) \rightarrow TY(r_x)$, called the *table function* for T_x. Being total functions, the tables ensure that for each state and input event exactly one successor state exists (notice that the target state can be the same as the source state, when the corresponding table does not prescribe a change in the value of the defined entity). Thus, the whole set of SCR tables in *Spec* describes the transition relation T from Σ_{Spec}, viewing it as a total function by mapping a state s to itself when an event is not reacted to.

A *condition table* T_c defines the value of an entity r_c as a function of the values of other entities in the current state (i.e., it does not depend on events). The SCR method requires that T_c guarantees *completeness* and *disjointness* [28], implying that in every state $s \in S$, T_c must assign exactly one value to r_c. As an example, consider the condition table in Figure 1(c) that defines controlled variable `cSafetyInjection`; this variable depends on mode class `mcPressure` and term `tOverriden`, and its table function $T_{\texttt{cSafetyInjection}}$ is as shown in Figure 2.

An *event table* T_e defines an entity r_e in terms of events. T_e then typically must consider both "old" (previous state) and "new" (current state) values of entities that r_e depends on. SCR requires that T_e guarantees *disjointness*, enforcing that two different table rows cannot simultaneously be satisfied. Also, according to SCR semantics, different rows must yield different values for r_e. Finally, if no event considered by T_e takes place (in the circumstances that T_e expects), then T_e maintains the same value of r_e in the next state (i.e., $r'_e = r_e$). As an example, consider the event table of Figure 1(b), defining term `tOverriden`. This term depends on the values of `mBlock`, `mBlock'`, `mReset`, `mReset'`, `mcPressure`, `mcPressure'` and `tOverriden`, and the table function $T_{\texttt{tOverriden}}$ is as shown in Figure 3.

A *mode transition table* T_m is a particular case of an event table, defining how the system transitions within modes of a mode class r_m when events occur. Thus, the table function F_m for T_m is defined similarly to event tables. In addition to the conditions imposed on event tables, T_m has to meet a *reachability* condition ensuring that all modes in $TY(r_m)$ can be reached from its prescribed initial mode.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

$$T_{\texttt{tOverriden}}(\texttt{mBlock}, \texttt{mReset}, \texttt{mcPressure}, \texttt{tOverriden}, \texttt{mBlock'}, \texttt{mReset'}, \texttt{mcPressure'}) =$$

$$\begin{cases} true & \text{if } (\texttt{mcPressure} = \texttt{TooLow} \wedge \texttt{mBlock'} = \texttt{On} \wedge \texttt{mBlock} = \texttt{Off} \wedge \texttt{mReset} = \texttt{Off}) \vee \\ & \quad (\texttt{mcPressure} = \texttt{Permitted} \wedge \texttt{mBlock'} = \texttt{On} \wedge \texttt{mBlock} = \texttt{Off} \wedge \texttt{mReset} = \texttt{Off}) \\ false & \text{if } (\texttt{mcPressure} = \texttt{TooLow} \wedge \texttt{mReset'} = \texttt{On} \wedge \texttt{mReset} = \texttt{Off}) \vee \\ & \quad (\texttt{mcPressure} = \texttt{Permitted} \wedge \texttt{mReset'} = \texttt{On} \wedge \texttt{mReset} = \texttt{Off}) \vee \\ & \quad (\texttt{mcPressure'} \neq \texttt{High} \wedge \texttt{mcPressure} = \texttt{High}) \vee \\ & \quad (\texttt{mcPressure'} = \texttt{High} \wedge \texttt{mcPressure} \neq \texttt{High}) \\ \texttt{tOverriden} & \text{otherwise} \end{cases}$$

Figure 3. tOverriden's table function

2.1.2. Executions, reachable states, and the verification of invariant properties. Let $\Sigma_{Spec} = (S, S_0, E, T)$ be *Spec*'s associated LTS. For set $F \subseteq E$, F^* (resp. F^+) is defined to be the set of all sequences (resp. non empty sequences) of events in F. The following notation is introduced. For every $e \in F$ and $es \in F^*$, $e \cdot es$ is the sequence resulting from prepending e to the beginning of es, $\#es$ is the length of es, and $es \uparrow i$ is the sequence containing the first i elements of es. For states $s, s' \in S$ and event $e \in F$, $s \overset{e}{\rightsquigarrow} s'$ denotes $T(s, e) = s'$. Furthermore, for monitored variable mv, $s \overset{\texttt{mv}}{\rightsquigarrow} s'$ denotes that there exists an input event e modifying mv ($e \in \texttt{mv}$) such that $s \overset{e}{\rightsquigarrow} s'$. The definition of $\rightsquigarrow$ is lifted to sequences of events in the usual way: $s \overset{\lambda}{\rightsquigarrow} s'$ iff $s = s'$ (λ represents the empty sequence), and $s \overset{e \cdot es}{\rightsquigarrow} s'$ iff there exists a state s'' such that $s \overset{e}{\rightsquigarrow} s'' \wedge s'' \overset{es}{\rightsquigarrow} s'$. In addition, $s \overset{F^*}{\rightsquigarrow} s'$ (resp. $s \overset{F^+}{\rightsquigarrow} s'$) stands for $\exists es \in F^* \mid s \overset{es}{\rightsquigarrow} s'$ (resp. $\exists es \in F^+ \mid s \overset{es}{\rightsquigarrow} s'$). Hence, for monitored variable mv, $s \overset{\texttt{mv}^+}{\rightsquigarrow} s'$ indicates that there exists a non empty sequence of events that modify monitored variable mv that makes the system transition from s to s'.

An *execution* of Σ_{Spec} is a sequence $es \in E^*$ of input events, starting at an initial state $s_0 \in S_0$. Given an execution σ, $\#\sigma$ is the number of states in σ. Also, $\sigma.i$ yields σ's i-th state, for $0 \leq i < \#\sigma$, i.e., if $s_0 \overset{es \uparrow i}{\rightsquigarrow} s_i$, then $\sigma.i = s_i$.

The set *Reach*(Σ_{Spec}) of *reachable states* of Σ_{Spec} is defined as the ending states of executions: *Reach*$(\Sigma_{Spec}) = \{s \mid \exists s_0 \in S_0 . s_0 \overset{E^*}{\rightsquigarrow} s\}$. An *invariant property* is a predicate P over S that holds in every $s \in$ *Reach*(Σ_{Spec}). A *transition property* is a predicate P_T over $S \times S$, that must be true in every pair (s, s') of consecutive reachable states, i.e., states such that there exists an execution σ and value i such that $\sigma.i = s$ and $\sigma.(i+1) = s'$.

A traditional way for verifying that an invariant property P holds in Σ_{Spec} is to prove that the set *Reach*(Σ_{Spec}) does not intersect with the set of states satisfying $\neg P$. Notice that proving that any overapproximation (superset) of *Reach*(Σ_{Spec}) does not intersect with $\neg P$ also suffices to prove P valid in Σ_{Spec}. As feasible transitions only lead to states in *Reach*(Σ_{Spec}), then proving a transition property P_T valid can be done by proving that no transition executed from states satisfying *Reach*(Σ_{Spec}), or any overapproximation of it, satisfies $\neg P_T$. The computation of overapproximations is the core of abstract interpretation based approaches for verification, like the work presented in this paper.

2.2. Abstraction based analysis for SCR and the numerical variables problem

Instead of directly dealing with the verification of a reachability property on a potentially very large concrete LTS, abstraction based analyses propose to do so on an *abstract* state space, to increase scalability. That is, given an LTS $\mathcal{S} = (S, \Sigma, \rightarrow)$, an abstraction based approach proposes

Prepared using stvrauth.cls
DOI: 10.1002/stvr

considering *regions*, i.e., abstract states which represent sets of states from S, and abstract transitions that overapproximate the $\rightarrow$ transitions between the states constituting the regions. More precisely, a *region structure* for LTS $\mathcal{S}$ is a structure $\mathcal{R} = (R, \bot, \sqcup, \sqcap, \mathit{pre}, \mathit{post}, [.])$, consisting of a set R of regions, where each region r represents a set $[r]^\dagger$ 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 ($\mathit{pre}(r, l)$ returns the largest region such that, from all its states and traversing arcs labelled by l one arrives at states in r; $\mathit{post}(r, l)$ returns the largest set of states that can be reached from states in r through transitions labelled by l), respectively. An abstraction structure $\mathcal{A} = (\mathcal{R}, \mathit{pre}^{\mathcal{A}}, \mathit{post}^{\mathcal{A}}, \trianglelefteq)$ for an LTS $\mathcal{S}$ complements a region structure $\mathcal{R}$ for $\mathcal{S}$ with abstract pre and post operators $\mathit{pre}^{\mathcal{A}}$ and $\mathit{post}^{\mathcal{A}}$, and a precision preorder $\trianglelefteq$, such that $\mathit{pre}(r, l) \subseteq \mathit{pre}^{\mathcal{A}}(r, l)$, $\mathit{post}(r, l) \subseteq \mathit{post}^{\mathcal{A}}(r, l)$ (i.e., abstract transitions overapproximate concrete transitions between sets of states), assuming that $r \subseteq r'$ denotes $[r] \subseteq [r']$. Moreover, $\mathit{pre}^{\mathcal{A}}$ and $\mathit{post}^{\mathcal{A}}$ must be monotonic with respect to $(\trianglelefteq \cap \subseteq)$, i.e., the precision preorder $\trianglelefteq$ intuitively indicates how close the abstract pre and post operators are to the concrete ones, for given regions.

The above described general abstraction setting can be instantiated in various ways, in particular through an effective one known as *predicate abstraction*. In predicate abstraction, regions are characterized by sets of state properties called *support predicates*, and the concretization is simply the set of states satisfying the corresponding predicates. More precisely, if P is 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: $\bot = (false, \emptyset)$; $(\varphi, \Gamma) \sqcup (\varphi', \Gamma') = (\varphi \vee \varphi', \Gamma \cup \Gamma')$; $(\varphi, \Gamma) \sqcap (\varphi', \Gamma') = (\varphi \wedge \varphi', \Gamma \cup \Gamma')$; $\mathit{pre}((\varphi, \Gamma), l) = (\varphi_l^{pre}, \Gamma_{ls})$ where φ_l^{pre} is the 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 concretization $[(\varphi, \Gamma)]$ 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. $\mathit{post}^{A_P}((\varphi, \Gamma), l)$ 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 $\varphi_i \Rightarrow \mathit{pre}(\gamma, l)$. The operator pre^{A_P} is defined in a similar way.
- $\trianglelefteq$ is defined in the following way: $(\varphi, \Gamma) \trianglelefteq (\varphi', \Gamma')$ iff $\Gamma \supseteq \Gamma'$.

An important fact about abstraction based analyses, including predicate abstraction ones, is that the process is sound but incomplete with respect to unreachability. That is, if a state is determined unreachable in the abstract state space, it is guaranteed that it is unreachable in the concrete state space; but, a state may be reachable in the abstract state space while being unreachable in the concrete state space. This fact leads to possibly having *spurious* counterexamples, i.e., abstract counterexamples that cannot be "concretized" (reproduced in the concrete state space). These spurious counterexamples can be removed by *refining* the abstraction, adding appropriate

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

Prepared using stvrauth.cls
DOI: 10.1002/stvr

abstraction predicates that make such counterexamples cease, making the abstraction closer to the concrete state space. These new abstraction predicates can actually be computed automatically from spurious counterexamples, e.g. via interpolation, leading to automated counterexample guided abstraction refinement (CEGAR). Of course, the larger the set of abstraction predicates, the more concrete the abstract state space, and the more expensive the analysis becomes, so it is important to maintain the set of abstraction predicates small. Notice that the above characterization of predicate abstraction, called *lazy predicate abstraction* and introduced in [33], allows it to maintain abstraction predicates *local* to the regions, thus allowing one to refine only parts of the abstract state space, as required, during a CEGAR process.

For illustration purposes, let us consider the following simple analysis scenario. Suppose that one wants to check whether mode `Permitted` is reachable in the SIS specification or not, starting from a particular initial state in which `mcPressure = TooLow` $\wedge$ `mWaterPres=14`. The initial abstraction will have a single abstraction predicate, namely $\phi_0 : \texttt{mcPressure} = \texttt{Permitted}$, and the abstract state space is as shown in Fig. 4(a). A first abstract trace reaching mode `Permitted` is shown in Fig. 4(b), which is clearly spurious, since according to **NAT** an input event can increase `mWaterPres` in at most 10 units at a time. In this case, the abstraction may be refined by introducing a new abstraction predicate. Many approaches have been proposed to (automatically) suggest such predicates. In the present case, when using an interpolation based approach implemented on the MathSAT tool, the first abstraction predicate that is added to remove the spurious counterexample is `mWaterPres <= 24`, which leads to the refined abstraction in Fig 4(c). Now, for the refined abstraction, a new spurious counterexample is obtained, shown in Fig. 4(d). Clearly, since at least 89 input events modifying `mWaterPres` must take place for the system to transition to `Permitted` from the initial state, the process will need to "discover" 88 abstraction predicates (`mWaterPres=i`, with $\texttt{i} \in \{24, 34, 44, \ldots, 894\}$) before obtaining a (minimal) concretizable abstract counterexample, thus proving the reachability of mode `Permitted`.

Since the size of the abstract model constructed in any predicate abstraction based approach grows exponentially with respect to the number of abstraction predicates [24], requiring 88 abstraction predicates makes this simple analysis scenario rather difficult. Notice how the restrictions imposed by `NAT` affect the number of abstraction predicates required to produce concretizable abstract counterexamples; for instance, if `NAT` asserts that `mWaterPres` can change in at most 5 units in two consecutive measures, then the previous simple abstraction predicate based analysis will require 176 abstraction predicates. Clearly, numerical monitored variables with large domains that vary in small leaps (small compared to the size of the domain) are problematic for abstraction based approaches. Since monitored variables with these characteristics appear frequently in SCR specifications, as acknowledged by cases reported in the literature [12, 29, 21].

Let us now elaborate on the importance of a *lazy* abstraction approach, in contrast with a standard predicate abstraction mechanism. Consider the abstract state transition representation of the safety injection example, shown in Figure 5(a). This abstract transition system corresponds to an abstraction built from only two abstraction predicates, namely `mcPressure = TooLow` and `mWaterPres=14`, similar to the example used previously in this section; solid arrows correspond to valid abstract state transitions triggered by changes in the water pressure, while dotted lines indicate infeasible transitions, that had to be checked in the construction of the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

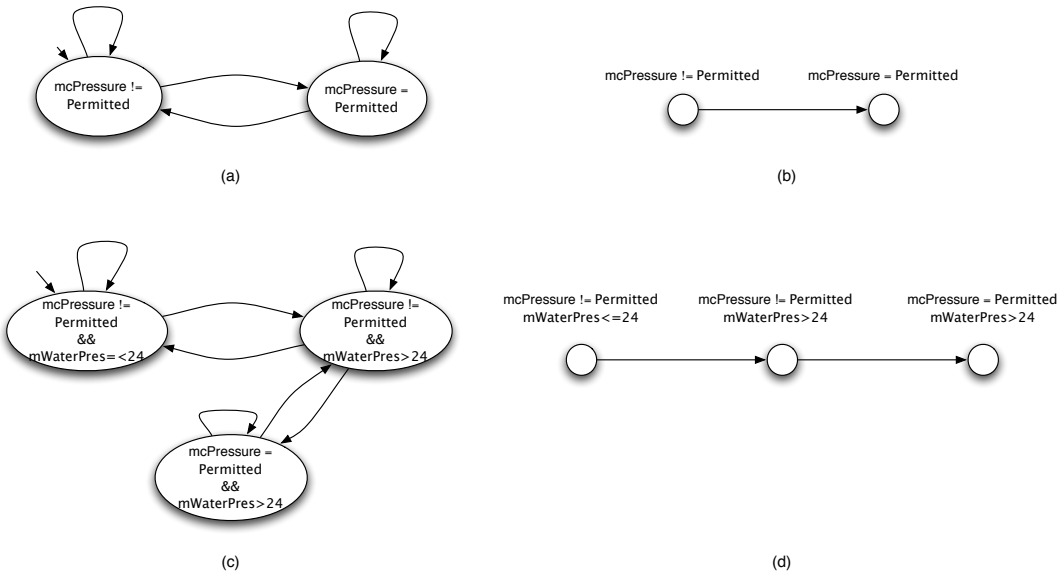

Figure 4. A simple predicate abstraction and abstract counterexamples for the SIS specification.

abstract model. Suppose that, while trying to concretize an abstract counterexample reaching state `mcPressure != TooLow` $\wedge$ `mWaterPres != 14` in two steps, the abstraction predicate `mWaterPres <= 24` is discovered. A lazy abstraction approach would introduce this new predicate only in states in which `mcPressure != TooLow` (the mode where the abstract trace failed to be concretized), leading to a new abstract transition system, shown in Figure 5(b). In this abstract transition system, infeasible states, that need to be checked for feasibility, are crossed, and again valid and infeasible transitions are depicted with solid and dotted arrows, respectively. On the other hand, if a standard predicate abstraction approach is used, then the new abstraction predicate will be used in *all* states, leading to more states and more detailed transitions, as shown in Figure 5(c). Notice how, even for this very small example, the number of states and transitions that need to be checked for feasibility, increases significantly.

Lazy abstraction, originally introduced by Henzinger et al. [33], has proved to have an important impact in abstraction based verification tools. This is acknowledged by the effectiveness of tools like BLAST [7], and the increased scalability observed in various applications of the concept of lazy abstraction [6, 46, 1, 15, 52].

3. LAZY PREDICATE ABSTRACTION FOR SCR

The end of the previous section motivates the approach that is introduced here, to analyze SCR requirements specifications via (lazy) predicate abstraction. Given a specification *Spec*, and some invariant property ϕ to verify on *Spec*, the approach starts by disregarding the **NAT** constraints in it, obtaining a *relaxed* specification *RSpec*. Then, a lazy predicate abstraction is performed, to attempt to verify ϕ on *RSpec*; since *RSpec* is weaker than *Spec*, the satisfaction of ϕ on *RSpec* implies its

Prepared using stvrauth.cls
DOI: 10.1002/stvr

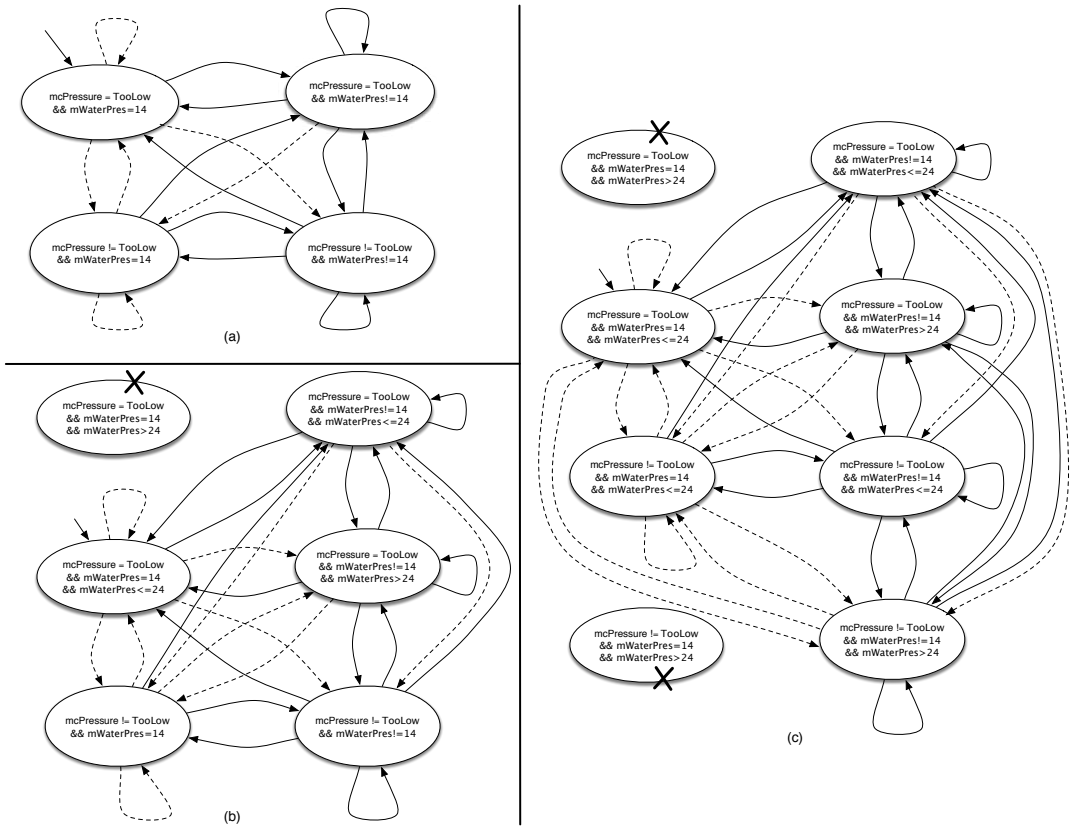

Figure 5. A simple example illustrating the difference between lazy abstraction and predicate abstraction.

satisfaction on *Spec*. The lazy predicate abstraction applied on *RSpec* is, in essence, a traditional lazy abstraction with automated counterexample guided abstraction refinement, tailored to exploit specific characteristics of SCR specifications.

Notice that the approach consists of two abstraction steps. The first is the relaxation resulting from the removal of the **NAT** constraints; the second is the lazy predicate abstraction. One may then obtain three kinds of counterexamples: *(i)* spurious counterexamples resulting from the lazy predicate abstraction, i.e., executions that are feasible in the abstract model but infeasible in *RSpec*; *(ii)* spurious counterexamples resulting from the removal of **NAT** constraints, i.e., executions that are both feasible in the abstract model and in *RSpec*, but are infeasible in *Spec*; and *(iii)* real counterexamples, i.e., abstract counterexamples that can be reproduced both in *Spec* and *RSpec*, and thus constitute actual violations to the property ϕ. Different approaches are employed to deal with the first two, which are described later on in the paper.

Since, despite its importance, the first abstraction is rather straightforward, let the current focus be on the second abstraction, the lazy predicate abstraction tailored to SCR specifications. This abstraction, whose preliminary version was originally introduced by Degiovanni et al. [19], maintains the same precision as the concrete specification as far as the *modes* are concerned, while

Prepared using stvrauth.cls
DOI: 10.1002/stvr

having a degree of precision in all other respects that may vary for different modes, i.e., is *lazy* (in the sense of the term as used by Henzinger et al. [33]) with respect to modes.

Let us provide a high level overview of how this abstraction process works, with references to the rest of the section, where further technical details are provided. The abstraction process can be captured as an algorithm, LPA-SCR, that takes as inputs an SCR specification *Spec*, and an invariant property φ_{prop} that one wants to prove true in *Spec* (alternatively, reachability analyses like those required for test generation can be performed with exactly the same mechanism). LPA-SCR attempts to construct an abstract reachability tree for *Spec*, by using the abstract transition relation $post^{\mathcal{A}}$ to expand forwardly the successors of already created abstract states. During the construction of the tree, LPA-SCR checks that any newly created abstract state satisfies φ_{prop}. If LPA-SCR constructs the whole abstract reachability tree with all states satisfying φ_{prop}, then it has computed an overapproximation of the set of reachable states of *Spec*, that in turn constitutes a proof that φ_{prop} holds in *Spec*. However, LPA-SCR might find abstract states where φ_{prop} does not hold. An abstract execution (a path in the abstract reachability tree) $\sigma_{\mathcal{A}}$ ending in a state where φ_{prop} does not hold constitutes an *abstract counterexample*. LPA-SCR invokes a decision procedure (MathSAT) to figure out if $\sigma_{\mathcal{A}}$ encodes a real (concrete) execution t of *Spec* violating φ_{prop}. If such a t exists, LPA-SCR returns it as a witness of the violation of φ_{prop} and terminates. If there is no concrete execution for $\sigma_{\mathcal{A}}$, $\sigma_{\mathcal{A}}$ is determined to be a *spurious counterexample*. In this case, LPA-SCR employs an interpolation based process, further discussed in section 3.5, to extract support predicates from $\sigma_{\mathcal{A}}$, that refine the current abstract reachability tree and get rid of $\sigma_{\mathcal{A}}$. After the refinement, LPA-SCR resumes the construction of the abstract reachability tree trying to verify φ_{prop} in *Spec*. LPA-SCR falls in the well-known category of counterexample guided abstraction refinement algorithms (CEGAR), that is, it realizes an abstraction algorithm that increases the precision of the abstract model as needed, using the information encoded in spurious counterexamples.

The rest of this section is devoted to the technical details of the abstraction process, including a precise definition for algorithm LPA-SCR. This abstraction is better introduced if the SCR semantics is provided in a different way, modularizing the global transition relation of a specification with respect to modes and input events, as presented below. A fixed SCR specification *Spec* is assumed, with mode class $M = \{m_1, \ldots, m_k\}$, whose associated LTS is referred to as $\Sigma_{Spec} = (S, S_0, E, T)$.

3.1. Mode explicit SCR semantics

The mode explicit semantics for SCR defined here splits each input event $e = (\mathtt{mv}, v, v')$ into several input events with fixed modes for the current and next states when e is triggered. In this way, in the mode explicit semantics, after triggering an event in the current state, it is known exactly what the mode of the next state will be. The reachable states of the mode explicit and the traditional semantics are guaranteed to be the same (Theorem 3.3). Hence, an invariant property is satisfied in the mode explicit semantics of *Spec* if and only if it is satisfied in the original semantics; then, the abstraction algorithm referring to the mode explicit semantics can be safely implemented.

A *mode-explicit input event* is a triple $e_m = (m, e, m')$, where m, m' are modes, and $e = (\mathtt{mv}, v, v')$ is an input event in the original semantics. Triggering mode-explicit event e_m takes the system from state s to s' if $MD(s) = m$ (the mode of s is m), $MD(s') = m'$ (the mode of s' is m'), $s(\mathtt{mv}) = v$ and $s'(\mathtt{mv}) = v'$. The set of *all* mode-explicit input events is denoted by $E^{M \times M}$; by considering all possible combinations of events with source/target modes, it is guaranteed that

Prepared using stvrauth.cls
DOI: 10.1002/stvr

no behaviours are lost with respect to the original semantics. The expression $(m, \mathtt{mv}, m')$ denotes the set of mode-explicit input events modifying monitored variable $\mathtt{mv}$ that transition from a state in mode m to a state in mode m', i.e., $(m, \mathtt{mv}, m') = \{(m, e, m') \in E^{M\times M} \mid \exists (v, v') \in \gamma_{\mathtt{mv}} \cdot e = (\mathtt{mv}, v, v') \in E\}$.

The *mode-explicit semantics* for SCR is given by the LTS $\Sigma^{M\times M}_{Spec} = (S, S_0, E^{M\times M}, T)$. Notice that here, the set of input events $E^{M\times M}$ is used in the definition of the LTS, instead of E as in the original semantics. For each input event $e_m \in \Sigma^{M\times M}_{Spec}$, $T(s, e_m) = s'$ is denoted by $s \overset{e_m}{\rightsquigarrow} s'$. Also, for monitored variable $\mathtt{mv}$, $s \overset{\mathtt{mv}}{\rightsquigarrow} s'$ indicates that there exists a mode-explicit input event $e_m \in (MD(s), \mathtt{mv}, MD(s'))$ such that $s \overset{e_m}{\rightsquigarrow} s'$.

The following results prove that $\Sigma^{M\times M}_{Spec}$ and Σ_{Spec} produce the same sets of reachable states.

Lemma 3.1
For every execution σ of $\Sigma_{Spec} = (S, S_0, E, T)$ there exists an execution σ_m of $\Sigma^{M\times M}_{Spec} = (S, S_0, E^{M\times M}, T)$ such that $\#\sigma = \#\sigma_m$, and $\sigma.i = \sigma_m.i$ for each $0 \le i \le \#\sigma + 1$.

Proof
The proof proceeds by induction on the length of σ. For the base case, if $\#\sigma = 0$, $\sigma : s_0 \overset{\lambda}{\rightsquigarrow} s_0$, and the mode-explicit execution satisfying the conditions of this lemma is $\sigma_m : s_0 \overset{\lambda}{\rightsquigarrow} s_0$. For the inductive case, assume that $\sigma : s_0 \overset{es}{\rightsquigarrow} s$ with $es \in E^*$, $\#\sigma = k$, and that $s \overset{e}{\rightsquigarrow} s'$, where $e = (\mathtt{mv}, v_1, v_2)$. By the inductive hypothesis, there exists $\sigma_m : s_0 \overset{es_m}{\rightsquigarrow} s$, for some event sequence $es_m \in (E^{M\times M})^*$, such that $\#\sigma = \#\sigma_m$, and $\sigma.i = \sigma_m.i$ for each $0 \le i \le \#\sigma + 1$. Let us call $m_1 = MD(s), m_2 = MD(s')$. Then, event $e_m = (m_1, e, m_2) \in E^{M\times M}$ is enabled in s and $s \overset{e_m}{\rightsquigarrow} s'$, completing the proof. □

Lemma 3.2
For every execution σ_m of $\Sigma^{M\times M}_{Spec} = (S, S_0, E^{M\times M}, T)$ there exists an execution σ of $\Sigma_{Spec} = (S, S_0, E, T)$ such that $\#\sigma_m = \#\sigma$, and $\sigma_m.i = \sigma.i$ for each $0 \le i \le \#\sigma_m + 1$.

Proof
Again, the proof proceeds by induction on $\#\sigma$. The base case can be proved as in Lemma 3.1. For the inductive case, assume that $\sigma_m : s_0 \overset{es_m}{\rightsquigarrow} s$ with $es_m \in (E^{M\times M})^*$, $\#\sigma_m = k$, and that $s \overset{e_m}{\rightsquigarrow} s'$, where $e_m = (m_1, e, m_2)$, $m_1 = MD(s), m_2 = MD(s')$, $e = (\mathtt{mv}, v_1, v_2)$. By the inductive hypothesis, there exists $\sigma : s_0 \overset{es}{\rightsquigarrow} s$, for some event sequence $es \in E^*$, such that $\#\sigma_m = \#\sigma$, and $\sigma_m.i = \sigma.i$ for each $0 \le i \le \#\sigma + 1$. But clearly $s \overset{e}{\rightsquigarrow} s'$ for $e = (\mathtt{mv}, v_1, v_2) \in E$, and the Lemma holds. □

Theorem 3.3
For every SCR specification *Spec*, *Reach*($\Sigma^{M\times M}_{Spec}$)=*Reach*($\Sigma_{Spec}$).

Proof
Follows from Lemmas 3.1 and 3.2 and the definition of *Reach*. □

3.2. Modularizing the transition relation

In order to improve the abstraction algorithm, the transition function T of $\Sigma^{M\times M}_{Spec}$ is *modularized*. That is, a set of simpler functions $T_1, \ldots, T_k$ is derived from T, that when combined are equivalent to T, but that can be used separately in the process of abstraction, allowing for an improvement in analysis performance. Two different modularizations are proposed, described below.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

3.2.1. Event-based modularization. Notice that if a monitored variable mv is changed in a transition, by the One-Input assumption all the other monitored variables cannot change during the same transition. Furthermore, events that do not depend on mv are not triggered in this situation (formally, event e is said to depend on mv if e refers to some entity n such that n is mv or $(n, \texttt{mv}) \in D_{new}+$). Thus, for each monitored variable mv, a simplified version $T_{\texttt{mv}}$ of the transition function T is computed by removing formulas of the table functions coming from events that do not depend on mv. So, when an event $e \in \texttt{mv}$ is triggered, $T_{\texttt{mv}}$ is equivalent to T, i.e., $T_{\texttt{mv}}(s, e) = T(s, e)$ for any state $s \in S$. Notice that this modularisation step can only be performed with event and mode transition tables, as condition tables are not described in terms of events.

As an example, recall table functions $F_{\texttt{tOverriden}}$ and $F_{\texttt{mcPressure}}$, defining term tOverriden and mode class mcPressure, respectively (section 2.1). The modularization of $F_{\texttt{tOverriden}}$ and $F_{\texttt{mcPressure}}$ with respect to events mReset are as follows:

$$\texttt{tOverriden'} =$$

$$F_{\texttt{tOverriden}_{\texttt{mReset}}}(\texttt{mBlock}, \texttt{mReset}, \texttt{mcPressure}, \texttt{tOverriden}, \texttt{mBlock'}, \texttt{mReset'}, \texttt{mcPressure'}) =$$

$$\begin{cases} \mathit{false} & \text{if } (\texttt{mcPressure} = \texttt{TooLow} \wedge \texttt{mReset'} = \texttt{On} \wedge \texttt{mReset} = \texttt{Off}) \vee \\ & \quad (\texttt{mcPressure} = \texttt{Permitted} \wedge \texttt{mReset'} = \texttt{On} \wedge \texttt{mReset} = \texttt{Off}) \\ \texttt{tOverriden} & \text{otherwise} \end{cases}$$

$$\texttt{mcPressure'} = F_{\texttt{mcPressure}_{\texttt{mReset}}}(\texttt{mWaterPres}, \texttt{mcPressure}, \texttt{mWaterPres'}) = \begin{cases} \texttt{mcPressure} & \mathit{true} \end{cases}$$

Notice that the only event in the table defining tOverriden (Figure 1(b)) that depends on mReset is @T(mReset = On) (second row second column of the table). Thus, this is the only event considered in the modularized table function $F_{\texttt{tOverriden}_{\texttt{mReset}}}$. On the other hand, none of the events in the mode transition table defining mcPressure (Figure 1(a)) depend on mReset (mReset cannot trigger a mode change). Hence, $F_{\texttt{mcPressure}_{\texttt{mReset}}}$ always returns the original value of mcPressure.

Considering that $\texttt{mv}_1, \ldots, \texttt{mv}_j$ are the monitored variables of *Spec*, the abstraction algorithm first modularizes T as described above, obtaining $T_{\texttt{mv}_1}, \ldots, T_{\texttt{mv}_j}$. It is important to remark that this process is completely automated, as the dependencies (relation $D_{new}+$ and the dependency relation for events) can be automatically determined from a syntactic analysis of *Spec* [10]. Then, when computing the abstract successors for an abstract state with respect to event $T_{\texttt{mv}_i}$ $(1 \leq i \leq j)$, the algorithm selects the corresponding $T_{\texttt{mv}_i}$ to be applied. As evidenced by the example above, the modularized transition functions can be encoded more succinctly as logic formulas than the whole transformation function T. This presents two advantages for the abstraction algorithm: the computation of abstract successors becomes faster (as it calls a decision procedure with a smaller formula), and the precision of the computed abstract model is increased, as in practice the interpolation based refinement procedure (section 3.5) computes stronger interpolants with the reduced formulas (section 5).

3.2.2. Mode-based modularization. The mode explicit semantics enables the abstraction algorithm to perform a different modularization of the transition relation, exploiting the fact that, when a mode explicit input event is triggered, it is known exactly which modes participate in the previous and next system states. For every pair m, m' of modes, a simpler transition relation $T_{m,m'}$ will be derived from T by removing all the events and conditions that cannot occur when transitioning

Prepared using stvrauth.cls
DOI: 10.1002/stvr

from m to m'. That is, from a pair m and m' of source and target modes, $T_{m,m'}$ is constructed by discarding from T all rows that involve events that do not start at m, or that do not transition to m'. As for the previous modularization, the abstraction algorithm will employ $T_{m,m'}$ to compute the abstract successors when an event $e = (m, \cdot, m')$ is triggered (e modifies any monitored variable, but the modes of the previous and next states are m and m', respectively). Consider for instance that event $e = (\texttt{High}, \cdot, \texttt{High})$ is triggered, that is, being in mode `High`, a change is produced by event e but the system stays in mode `High`. $F_{\texttt{cSafetyInjection}_{\texttt{High,High}}}$ and $F_{\texttt{tOverriden}_{\texttt{High,High}}}$ for table functions $F_{\texttt{cSafetyInjection}}$ defining `cSafetyInjection` (Figure 1(c)) and $F_{\texttt{tOverriden}}$ defining `tOverriden` (Figure 1(b)) are as follows:

$$\texttt{cSafetyInjection} =$$
$$F_{\texttt{cSafetyInjection}_{\texttt{High,High}}}(\texttt{mcPressure}, \texttt{tOverriden}) = \begin{cases} \texttt{Off} & \text{if } \texttt{mcPressure} = \texttt{High} \end{cases}$$

$$\texttt{tOverriden'} =$$
$$F_{\texttt{tOverriden}_{\texttt{High,High}}}(\texttt{mBlock}, \texttt{mReset}, \texttt{mcPressure}, \texttt{tOverriden}, \texttt{mBlock'}, \texttt{mReset'}, \texttt{mcPressure'}) =$$
$$\begin{cases} \texttt{tOverriden} & \textit{true} \end{cases}$$

Notice how in these cases, only formulas with source mode `High` are kept. Observe in particular that in $F_{\texttt{tOverriden}}$ there is only one formula with source mode `High`, which consists of the event `@F(mcPressure=High)`. However, `@F(mcPressure=High)` is clearly not enabled when the target mode is `High`, thus $F_{\texttt{tOverriden}_{High,High}}$ always returns the old value of `tOverriden`.

Mode-based and event-based modularizations can be combined to produce smaller parts of the transition relation T, "slicing" it in relation to modes and events of interest. Indeed, the abstraction used in this paper applies a modularization by events first, and then the mode-based modularization. That is, for each input event $e = (m, \texttt{mv}, m')$, a transition relation $T_{m,\texttt{mv},m'}$ is obtained by first computing $T_{\texttt{mv}}$ as in the previous section, and then applying the mode-based modularization to $T_{\texttt{mv}}$.

3.3. The Abstraction Setting

As mentioned previously, the abstraction algorithm presented in this paper is based on the lazy predicate abstraction framework presented in [33]. Lazy abstraction allows different degrees of precision in different parts of the abstract model (i.e., different sets of support predicates), as opposed to previous predicate abstraction approaches where abstraction predicates were considered global [24]. Since lazy predicate abstraction was originally devised to analyze C programs [33], it localizes support predicates to *program locations*. Thus, to construct an abstract state with location l, lazy abstraction only considers the support predicates corresponding to l. In addition, the abstraction refinement algorithm of lazy abstraction is defined in such a way that newly discovered abstraction predicates are added only to the program locations where they are necessary.

In SCR specifications, mode classes define a partition of the set of system's states, and one introduces modes to split the system's state when the system needs to react differently to events according to the mode of operation. Therefore, it is proposed here to keep the same set of abstraction predicates for all states that share the same mode, in an SCR specification. The abstraction refinement procedure for the algorithm is also designed to add discovered predicates "locally", in the sense that they will refine all states within the same mode.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Following [33], the abstract domain used by the lazy abstraction approach algorithm is defined as follows, using LIA formulas (the language of the quantifier free linear integer arithmetic) to characterize abstract states. LIA is chosen because it is decidable, and because there exists an interpolation algorithm for LIA that is used for the abstraction refinement procedure (section 3.5).

Let $\Sigma_{Spec}^{M\times M}$ be the mode explicit LTS associated with *Spec*. An abstract state of the abstraction approach presented in this paper will be defined as a set of *mode explicit atomic regions*, defined as follows.

Definition 3.1. A *mode explicit atomic region* is a tuple $a = (m, \varphi, \Pi)$ composed of a mode $m \in M$, a function mapping modes to support (abstraction) predicates $\Pi : M \mapsto 2^{LIA}$, and a boolean formula φ over the predicates of $\Pi(m)$.

Intuitively, region $a = (m, \varphi, \Pi)$ represents the set of states with mode m that satisfy φ, denoted with the region's concretization $[a]$. Notice that the modes in abstract regions are maintained explicit. The abstraction approach performs an abstraction process that is *precise* with respect to modes, and allows for different degrees of precision for states with different modes.

In addition, for any mode explicit atomic regions (m_1, φ_1, Π_1) and (m_2, φ_2, Π_2), the following operations are defined:

- $[(m_1, \varphi_1, \Pi_1)] = [m_1 \wedge \varphi_1]$, where $[f]$, for $f \in LIA$, denotes the set of states described by f.
- $(m_1, \varphi_1, \Pi_1) \sqcup (m_2, \varphi_2, \Pi_2) = (m_1, \varphi_1 \vee \varphi_2, \lambda m.\Pi_1(m) \cup \Pi_2(m))$ if $m_1 = m_2$; $\perp$ otherwise.
- $(m_1, \varphi_1, \Pi_1) \sqcap (m_2, \varphi_2, \Pi_2) = (m_1, \varphi_1 \wedge \varphi_2, \lambda m.\Pi_1(m) \cup \Pi_2(m))$ if $m_1 = m_2$; $\perp$ otherwise.
- If $e_m = (m_1, \mathtt{mv}, m_1')$, $post((m_1, \varphi_1, \Pi_1), e_m) = (m_1', \varphi_{e_m}^{post}, \Pi_{ls})$, where $\varphi_{e_m}^{post}$ is the strongest postcondition of φ_1 with respect to event e_m, and $\Pi_{ls}(m_1')$ is the least superset of $\Pi(m_1')$ which contains all the predicates in $\varphi_{e_m}^{post}$. Otherwise, if $e_m = (m_2, \cdot, \cdot)$ and $m_1 \neq m_2$, then $post((m_1, \varphi_1, \Pi_1), e_m) = \perp$.

Notice that the strongest postcondition $\varphi_{e_m}^{post}$ can be defined by the following LIA formula:

$$\begin{aligned}\varphi_{e_m}^{post} &= \varphi_1 \wedge e_m \wedge T_{m_1,\mathtt{mv},m_1'} \\ &= \varphi_1 \wedge (m_1 \wedge \mathtt{mv}' \neq \mathtt{mv} \wedge \gamma_{\mathtt{mv}}(\mathtt{mv}, \mathtt{mv}') \wedge m_1') \wedge T_{m_1,\mathtt{mv},m_1'}\end{aligned}$$

where primed variables denote the value of entities in the state returned by $post$, and the predicate $\gamma_{\mathtt{mv}}(\mathtt{mv}, \mathtt{mv}')$ indicates that monitored variable $\mathtt{mv}$ changed satisfying the **NAT** constraints.

Definition 3.2. Let A be the set of mode explicit atomic regions. A *mode explicit region structure* $\mathcal{R} = (R, \perp, \sqcup, \sqcap, post, [.])$ for $\Sigma_{Spec}^{M\times M}$ consists of:

- $R \subseteq 2^A$. That is, each mode explicit region $r \in R$ is a set of mode explicit atomic regions.
- $[r] = \bigcup_{a\in r}[a]$, for all $r \in R$.
- For $r_1, r_2 \in R$, the following definitions are considered:

Prepared using stvrauth.cls
DOI: 10.1002/stvr

- $r_1 \sqcup r_2 = r_1 \cup r_2 \cup \{a_1 \sqcup a_2 \mid a_1 \in r_1, a_2 \in r_2\}$,
- $r_1 \sqcap r_2 = \{a_1 \sqcap a_2 \mid a_1 \in r_1, a_2 \in r_2\}$,
- $post(r_1, e_m) = \bigcup_{a \in r_1} post(a, e_m)$.

$\mathcal{R}$ above is the algorithm's abstract domain. Notice that it is very similar to the abstract domain of the lazy predicate abstraction for C programs presented by R. Jhala [38]. The main differences are that, in the algorithm as presented in this paper, program locations are replaced by modes of the specification in atomic regions, and the call stack is eliminated, as there are no procedure abstractions in SCR.

In order to construct abstract states, the following abstraction function is employed.

Definition 3.3. The *abstraction function* $\alpha : LIA \times 2^{LIA} \mapsto LIA$ is defined by:

$$\alpha(\varphi, P) = \bigwedge_{p \in P} \{p | \varphi \rightarrow p\} \wedge \bigwedge_{p \in P} \{\neg p | \varphi \rightarrow \neg p\}$$

Intuitively, α takes a formula φ and a set of abstraction predicates P as inputs, and returns a formula representing the *predicate abstraction* [24] of φ with respect to P. Formula $\alpha(\varphi, P)$ always returns a (LIA) predicate involving a conjunction of predicates and negations of predicates from P. Notice that $\alpha(\varphi, P)$ is an abstraction of the original φ, in the sense that the states it represents is a superset of the states described by φ. $\alpha(\varphi, P)$ is said to overapproximate φ, i.e., $[\varphi] \subseteq [\alpha(\varphi, P)]$.

To implement α, one needs to be able to decide whether a predicate $p \in P$ or its negation are implied by φ. The MathSAT SMT solver [11] is an example of a tool that is suitable for this task.

The abstraction structure that is the basis of the present approach can now be introduced.

Definition 3.4. A *mode explicit abstraction structure* is a tuple $\mathcal{A} = (\mathcal{R}, post^{\mathcal{A}}, \trianglelefteq)$, where:

- $\mathcal{R}$ is a mode explicit region structure,
- For $r \in R$ and mode explicit input event $e_m = (m_1, \mathtt{mv}, m_2)$, the abstract strongest postcondition operator is defined by $post^{\mathcal{A}}(r, e_m) = \bigcup_{a \in r} post^{\mathcal{A}}(a, e_m)$. For an atomic mode explicit region $a = (m, \varphi, \Pi)$, $post^{\mathcal{A}}$ is defined by:
  - $post^{\mathcal{A}}((m, \varphi, \Pi), e_m) = (m_2, \alpha(\varphi^{post}_{e_m}, \Pi(m_2)), \Pi)$ if $m = m_1$.
  - $post^{\mathcal{A}}((m, \varphi, \Pi), e_m) = \bot$ if $m \neq m_1$.
- For region r, $r.\Pi$ is defined as $\bigcup_{(\cdot,\cdot,\Pi) \in r} \Pi$. For $r_1, r_2 \in R$ with $\Pi_1 = r_1.\Pi$ and $\Pi_2 = r_2.\Pi$, the precision preorder is defined by: $r_1 \trianglelefteq r_2$ iff $\forall m \in M. \Pi_2(m) \subseteq \Pi_1(m)$

Intuitively, for a mode explicit event $e_m = (m_1, \mathtt{mv}, m_2)$, $post^{\mathcal{A}}((m_1, \varphi, \Pi), e_m)$ returns a region with e_m's target mode m_2, and the set of states obtained by abstracting the result of executing the concrete $post$ on φ, with respect to the set of abstraction predicates associated with mode m_2.

Based on the proof strategy by Henzinger et al. [33], the following theorems guarantee that the above-introduced abstraction of $Spec$ characterizes an overapproximation of $Spec$'s behaviours.

Theorem 3.4

For any $r \in R, e_m \in E_{M \times M}, post(r, e_m) \sqsubseteq post^{\mathcal{A}}(r, e_m)$.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Proof
To prove this result, it has to be proved that $[post(r, e_m)] \subseteq [post^{\mathcal{A}}(r, e_m)]$. Then, if for any mode explicit atomic region $a \in r$, $[post(a, e_m)] \subseteq [post^{\mathcal{A}}(a, e_m)]$, the theorem follows by definition of $[.]$. Let us prove that $[post(a, e_m)] \subseteq [post^{\mathcal{A}}(a, e_m)]$.

Let $a = (m, \varphi, \Pi)$ and $e_m = (m', \mathtt{mv}, m'')$. If $m \neq m'$, by definition we have $post(a, e_m) = \bot$ and $post^{\mathcal{A}}(a, e_m) = \bot$, and therefore $[post(a, e_m)] \subseteq [post^{\mathcal{A}}(a, e_m)]$. If $m = m'$, then $[post(a, e_m)] = [(m'', \varphi_{e_m}^{post}, \Pi_{ls})] = [m'' \wedge \varphi_{e_m}^{post}]$ and $[post^{\mathcal{A}}(a, e_m)] = [(m'', \alpha(\varphi_{e_m}^{post}, \Pi(m'')), \Pi)] = [m'' \wedge \alpha(\varphi_{e_m}^{post}, \Pi(m''))]$. Since α is an abstraction function, then $[\varphi_{e_m}^{post}] \subseteq [\alpha(\varphi_{e_m}^{post}, \Pi(m''))]$. Hence, $[m'' \wedge \varphi_{e_m}^{post}] \subseteq [m'' \wedge \alpha(\varphi_{e_m}^{post}, \Pi(m''))]$ and the proof is finished. □

Theorem 3.5
If $r, r' \in R$, such that $r \trianglelefteq r'$, then for any $e_m \in E_{M \times M}$, $post^{\mathcal{A}}(r, e_m) \trianglelefteq post^{\mathcal{A}}(r', e_m)$.

Proof

Let $\Pi_r = r.\Pi$ and $\Pi_{r'} = r'.\Pi'$ be the mappings from modes to support predicates of regions r and r', respectively. As $r \trianglelefteq r'$, by definition of $\trianglelefteq$, for all mode $m \in M$, $\Pi_r(m) \subseteq \Pi_{r'}(m)$. By definition of $post^{\mathcal{A}}$, the region $post^{\mathcal{A}}(r, e_m)$ will contain the same mapping of r, i.e., Π_r. Similarly, the mapping of region $post^{\mathcal{A}}(r', e_m)$ will be the same of region r', i.e., $\Pi_{r'}$. Then, as $\Pi_r(m) \subseteq \Pi_{r'}(m)$, by definition of $\trianglelefteq$, it follows that $post^{\mathcal{A}}(r, e_m) \trianglelefteq post^{\mathcal{A}}(r', e_m)$. □

Theorem 3.6
If $r, r' \in R$, such that, $r \sqsubseteq r'$, then for any $e_m \in E_{M \times M}$, $post^{\mathcal{A}}(r, e_m) \sqsubseteq post^{\mathcal{A}}(r', e_m)$.

Proof
Suppose an atomic region $(m, \varphi, \Pi) \in post^{\mathcal{A}}(r, e_m)$. By definition of $post^{\mathcal{A}}$, there must exist an atomic region $a \in r$ such that $post^{\mathcal{A}}(a, e_m) = (m, \varphi, \Pi)$. Because of the hypothesis $r \sqsubseteq r'$, the atomic region a must also belong to region r' (i.e., $a \in r'$). Then, $(m, \varphi, \Pi) = post^{\mathcal{A}}(a, e_m) \in post^{\mathcal{A}}(r', e_m)$. □

3.4. SCR abstraction algorithm

The SCR abstraction algorithm just presented, shown in Algorithm 1, is a version of the original lazy predicate abstraction put forward by Henzinger et al. [33] tailored to SCR specifications. It employs the mode explicit abstraction structure of definition 3.4, i.e., its abstract domain consists of the mode explicit region structures of definition 3.2, and operator $post^{\mathcal{A}}$ is used to compute the successors of abstract states (regions). In addition, the present abstraction assumes that the transition relation of *Spec* has been modularized (by modes and events) following the steps given in section 3.2, before executing the algorithm.

Let us further explain some details of the LPA-SCR algorithm. In this algorithm, a node of the abstract reachability tree is denoted by $n : r$, where n is the node's name, and r the abstract region (definition 3.2) associated with the node. In addition, Π is a function storing the set of abstraction predicates currently associated with each mode. Let $init$ be the predicate describing the initial state of *Spec*, and assume that m_{init} is the mode of state $init$. In line 2, LPA-SCR initializes Π with φ_{prop} as the only available predicate for each mode (except for m_{init} that additionally includes

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Algorithm 1 SCR abstraction algorithm

1: **function** LPA-SCR
2: $\Pi = \{m_{init} \to \{init\}\} \cup \lambda m.\{\varphi_{prop}\}$, for all $m \in M$
3: $r_0 = \{(m_{init}, \alpha(init, \Pi(m_{init})), \Pi)\}$
4: create an unmarked root node $r : r_0$
5: **while** there are unmarked nodes **do**
6: pick an unmarked node $n : r_n$
7: **if** there exists $(\cdot, \varphi_n, \cdot) \in r_n$ such that φ_{prop} does not appear positively in φ_n **then**
8: - - the property may not hold at node n
9: - - $\sigma_{\mathcal{A}}$ is the abstract error trace starting at the root r and ending at n
10: $\sigma_{\mathcal{A}} = r : r_0 \xrightarrow{\sigma} n : r_n$
11: - - check if there is a feasible concrete error trace t represented by $\sigma_{\mathcal{A}}$
12: - - if infeas., $\sigma_{\mathcal{A}}$ is spurious, $\Pi' : M \to 2^{LIA}$ are the discovered interpolants
13: $(feasible, t, \Pi') = solve(\sigma_{\mathcal{A}})$
14: **if** $feasible$ **then**
15: - - $\sigma_{\mathcal{A}}$ contains concrete error trace t
16: **return** t
17: **else**
18: - - $\sigma_{\mathcal{A}}$ is spurious
19: let $n' : r'_n$ be the oldest ancestor of n in $\sigma_{\mathcal{A}}$ such that
20: there exists $(m'_n, \cdot, \cdot) \in r'_n$ with $\Pi'(m'_n) \neq \emptyset$
21: let $n'' : r''_n$ be the parent of $n' : r'_n$ in $\sigma_{\mathcal{A}}$
22: - - refine the abstract reachability tree starting at n''
23: drop the subtrees of n''
24: unmark n''
25: - - update Π to include the discovered interpolants Π'
26: $\Pi = \lambda m.(\Pi(m) \cup \Pi'(m))$
27: **else if** $r_n \sqsubseteq r'_n$ for some node $n' : r'_n$ marked as uncovered **then**
28: - - n is covered by n'
29: mark n as covered
30: **else**
31: - - construct the abstract successors of n
32: **for** each $e_m = (m, \mathtt{mv}, m')$, such that $m, m' \in M$ and $\mathtt{mv}$ is a monitored variable **do**
33: $r'_n = post^{\mathcal{A}}(r_n, e_m)$
34: **if** $r'_n \neq \bot$ **then**
35: create an unmarked node $n' : r'_n$ and an edge $n : r_n \xrightarrow{e_m} n' : r'_n$
36: mark n as uncovered
37: **return** region $\bigsqcup\{r_u | u : r_u$ is a node marked as uncovered$\}$

predicate $init$). Π will grow monotonically throughout the execution, as abstraction predicates will be added to Π each time the refinement algorithm discovers new predicates to get rid of spurious counterexamples. For simplicity, Π is made a global variable in the algorithm (instead of being part of each region), and it is assumed that the operator $post^{\mathcal{A}}$ employs Π to compute abstract successors.

Lines 3 and 4 construct the root node $r : r_0$ of the abstract reachability tree, where r_0 consists of an atomic region with mode m_{init}, predicate $\alpha(init, \Pi(m_{init}))$ (the predicate abstraction of $init$ with respect to the predicates in $\Pi(m_{init})$), and (the initial) Π as support predicates.

After the initialization, in the main loop (lines 5–35), LPA-SCR performs an iterative process, where each step consists of either: (i) deciding if the abstract counterexample found (a node violating φ_{prop}) represents a concrete counterexample, or the abstract model has to be refined to get rid of

Prepared using stvrauth.cls
DOI: 10.1002/stvr

this spurious violation, (ii) marking a node of the tree as *covered* due to its region already being computed elsewhere in the tree, or (iii) expanding the abstract reachability tree by constructing the abstract successors of a node. During this process, the algorithm *marks* already visited nodes to avoid processing them twice. When the abstract successors of a node are built, the node is marked as *uncovered*. Conversely, a node is marked as *covered* when its region has been computed in another part of the tree. When the whole abstract reachability tree has been computed, the regions of the nodes marked as uncovered determine the set of reachable states of the tree, which constitute a proof of the validity of φ_{prop} in *Spec*.

In line 6 the algorithm takes an unmarked node $n : r_n$ and decides to perform (i), (ii) or (iii) above, depending on the characteristics of r_n. In case (i), the condition in line 7 is true, that is, φ_{prop} does not appear positively in r_n's predicate. In such case, we have an abstract counterexample $\sigma_{\mathcal{A}}$ starting at the root of the tree and ending at n –see line 10. Then, the algorithm needs to invoke the decision procedure with $\sigma_{\mathcal{A}}$ as input, to find out if there exists a concrete execution violating φ_{prop} encoded in $\sigma_{\mathcal{A}}$. This is carried out by function $solve$ in line 12, that represents an invocation to the decision procedure (the SMT solver MathSAT). The result of executing *solve* is a triple $(feasible, t, \Pi')$. If there exists a concrete execution violating φ_{prop} contained in $\sigma_{\mathcal{A}}$, $feasible$ is set to true, and the witness concrete execution stored in variable t (Π' is undefined in this case). Then, as $feasible$ is true, LPA-SCR returns the execution t witnessing that φ_{prop} does not hold in *Spec* (lines 13-15). Otherwise, if $\sigma_{\mathcal{A}}$ is spurious, $solve$ sets $feasible$ to false, and returns in Π' a mapping with the predicates needed to get rid of the spurious execution. As mentioned earlier, the computation of Π' is carried out by the interpolation based refinement algorithm of section 3.5. If $\sigma_{\mathcal{A}}$ is spurious, in lines 18-19 the algorithm finds out the oldest ancestor n' of n in $\sigma_{\mathcal{A}}$ that needs to be refined, that is, an n' such that Π' does not contain any new predicate to add to any of the nodes in the path from the root r to n'. Then, in lines 20-22 LPA-SCR drops the subtrees that must be refined taking into account the newly discovered predicates Π', that is, the subtrees starting from the parent n'' of n'. In addition, in line 23 n'' is unmarked, so it becomes eligible again for the computation of its abstract successors in a later iteration, but this time including the new set of predicates allowing to discard $\sigma_{\mathcal{A}}$. Finally, in line 25 Π is updated to include the newly discovered predicates Π'. Notice that the predicates in Π' are added locally to the modes where they were discovered, whereas in the original lazy abstraction approach they are added locally to the corresponding program locations [33].

If node $n : r_n$ does not violate φ_{prop}, LPA-SCR considers case (ii) above. Thus, in line 26 LPA-SCR checks if there exists another uncovered (marked) node $n' : r'_n$ such that the set of states represented by r'_n includes those represented by r_n (denoted by $r_n \sqsubseteq r'_n$). Notice that this is a very simple syntactic check: if a support predicates p in r'_n is set to true (resp. false), then p should be true (resp. false) in r_n too. When $r_n \sqsubseteq r'_n$, LPA-SCR does not need to continue expanding n, since all of n's successors must be checked to satisfy φ_{prop} in its own part of the reachability tree, and the set of states defined by any successor of n' is included in the set of states defined by some successor of n (more precisely, for any event $e_m \in E_{M\times M}$, $post^A(r_n, e_m)$ has to satisfy φ_{prop}, and $post^A(r_n, e_m) \sqsubseteq post^A(r'_n, e_m)$). In this case, n is said to be *covered* by n', and LPA-SCR marks n (as covered) so it is not picked up again in future iterations.

Finally, when cases (i) and (ii) do not apply, in line 29 LPA-SCR starts the computation of the successors of node $n : r_n$. This corresponds to case (iii) above. Thus, in lines 31-34 LPA-SCR creates an unmarked node $n' : r'_n$ with $r'_n = post^A(r_n, e_m)$ for each event $e_m = (m, \mathtt{mv}, m')$, where

Prepared using stvrauth.cls
DOI: 10.1002/stvr

$m, m' \in M$ and $\texttt{mv}$ is a monitored variable. In addition, it adds an edge to the reachability tree from n to n' (line 34). Then, n is marked as uncovered to indicate that its abstract successors have been computed (line 35).

In the next section, the refinement algorithm implemented by the $solve$ procedure is explained in greater detail.

3.5. Interpolation based abstraction refinement

To get rid of spurious counterexamples, LPA-SCR employs an interpolation based abstraction refinement approach, introduced first for the original lazy abstraction by Henzinger et al. [34]. In this section, the details of how this refinement approach is applied to remove the kind of spurious counterexamples that appear in LPA-SCR, are explained. The reader interested in learning more about interpolation is referred to the work of Henzinger et al. [34].

Definition 3.5. Let φ_1, φ_2 be two formulas such that their conjunction is unsatisfiable. An *interpolant* for (φ_1, φ_2) is a formula ψ such that: (i) φ_1 implies ψ, (ii) $\psi \wedge \varphi_2$ is unsatisfiable, (iii) ψ only contains variables that appear both in φ_1 and φ_2.

For some logics (including LIA), interpolants can be extracted automatically from the proof of unsatisfiability of $\varphi_1 \wedge \varphi_2$, in an efficient way.

Let us now discuss how LPA-SCR uses interpolation to get rid of spurious counterexamples. Assume that $\sigma^{\mathcal{A}}$ is a spurious counterexample returned by LPA-SCR, and let $\sigma^{\mathcal{A}} = \sigma_1^{\mathcal{A}} : \sigma_2^{\mathcal{A}}$ be a splitting of $\sigma^{\mathcal{A}}$ into a disjoint prefix $\sigma_1^{\mathcal{A}}$ and suffix $\sigma_2^{\mathcal{A}}$. In addition, let φ_1 (resp. φ_2) be an encoding of prefix (resp. suffix) $\sigma_1^{\mathcal{A}}$ ($\sigma_2^{\mathcal{A}}$) as a logical formula. Notice that $\sigma^{\mathcal{A}}$ is unsatisfiable since it is spurious. Hence, $\varphi_1 \wedge \varphi_2$ is unsatisfiable as well. Thus, an interpolant ψ can be computed for (φ_1, φ_2). By (i) above, ψ represents an overapproximation of the states obtained by executing prefix $\sigma_1^{\mathcal{A}}$ (and it is clearly true after the execution of $\sigma_1^{\mathcal{A}}$). By (ii), it follows that suffix $\sigma_2^{\mathcal{A}}$ cannot be executed starting from states satisfying ψ. Therefore, ψ is the abstraction predicate we need to rule out $\sigma_1^{\mathcal{A}} : \sigma_2^{\mathcal{A}}$.

The actual implementation of $solve$ (invoked in line 12 of Algorithm 1 to get rid of $\sigma^{\mathcal{A}}$) repeats the above process for each feasible splitting of $\sigma^{\mathcal{A}}$ (into nonempty prefixes and suffixes). Let $\sigma^{\mathcal{A}}$ be composed of nodes $n_1 : r_1, n_2 : r_2, \ldots, n_k : r_k$. Because $\sigma^{\mathcal{A}}$ is an execution computed by LPA-SCR, there must exist events $e_1, \ldots, e_{k-1}$ such that $post^{\mathcal{A}}(r_i, e_i) = r_{i+1}$, for $i = 1 \ldots k-1$. An invocation of $solve$ with $\sigma^{\mathcal{A}}$ yields formulas $\psi_1, \psi_2, \ldots, \psi_{k-1}$, such that each ψ_j ($j = 1 \ldots k-1$) is an interpolant for prefix $n_1 : r_1 \ldots n_j : r_j$ and suffix $n_{j+1} : r_{j+1} \ldots n_k : r_k$. $\psi_1, \psi_2, \ldots, \psi_{k-1}$ are the abstraction predicates required to refine the current reachability tree and rule out $\sigma^{\mathcal{A}}$.

The encoding of spurious counterexamples into logical (LIA) formulas will be used to build formulas (φ_1, φ_2) for prefixes and suffixes, that will then be fed to the decision procedure that carries out the computation of interpolants. Let $\sigma^{\mathcal{A}}$ be as above. Assume that each region r_i contains exactly one mode explicit atomic region, i.e., $r_i = \{(m_i, \varphi_i, \Pi)\}$. The encoding of $\sigma^{\mathcal{A}}$ as a LIA formula is: $(m_1 \wedge \varphi_1) \wedge (m_2 \wedge \varphi_2) \wedge \ldots \wedge (m_k \wedge \varphi_k)$. That is, each state of $\sigma^{\mathcal{A}}$ adds a conjunction of its mode and its predicate to the resulting formula. Notice that this encoding can be applied to any prefix and suffix of $\sigma^{\mathcal{A}}$.

Now, assume that $solve(\sigma^{\mathcal{A}})$ yields formulas $\psi_1, \psi_2, \ldots, \psi_{k-1}$. Intuitively, each ψ_i is an overapproximation of the execution $n_1 : r_1, n_2 : r_2, \ldots, n_i : r_i$, at abstract state n_i. But as $r_i = \{(m_i, \varphi_i, \Pi)\}$ ($i = 1 \ldots k-1$), and LPA-SCR's abstraction is precise with respect to modes, m_i

Prepared using stvrauth.cls
DOI: 10.1002/stvr

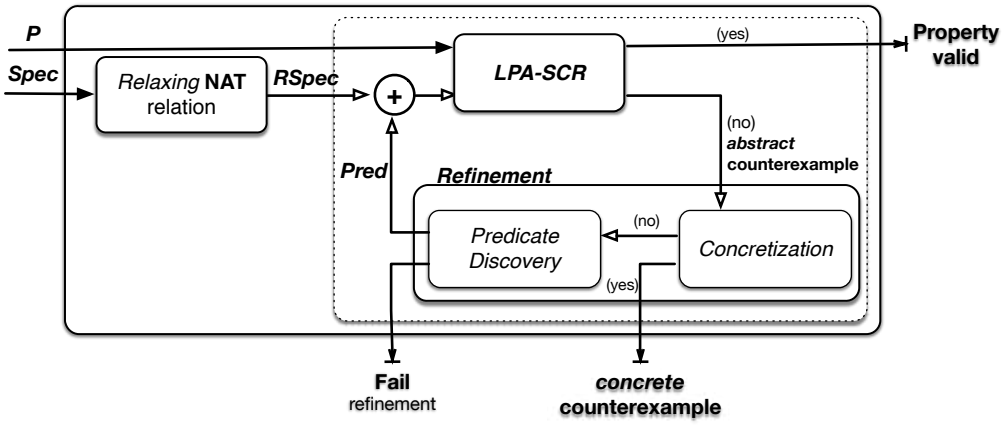

Figure 6. Overview of our approach.

is the mode where ψ_i is *relevant* for discarding $\sigma^{\mathcal{A}}$. Thus, $solve$ returns the computed interpolants as a mapping Π', such that for all m_i, $\Pi'(m_i)$ is the set of all interpolants that are relevant at mode m_i. In this way, LPA-SCR avoids adding the interpolants as global abstraction predicates. Indeed, in LPA-SCR the abstraction predicates are local to the modes where they are useful for ruling out $\sigma^{\mathcal{A}}$. This allows LPA-SCR to speed up the computation of abstract successors, as it is often the case that a lesser number of (relevant) abstraction predicates have to be considered when they are added locally to modes. The original interpolation based refinement algorithm for LPA is similar, but it employs program locations instead of modes to localize abstraction predicates.

The assumption made in this paper, that there is only one atomic region per abstract state, enables LPA-SCR to simplify the formulas that will be fed to the decision procedure, which in turn allows for the computation of simpler interpolants. Thus, the implementation of LPA-SCR enforces this assumption (although it could be dropped without compromising correctness).

It is important to remark that the implementation of $solve$ employs the MathSat SMT solver [11] to compute interpolants (for LIA formulas).

4. ABSTRACTION WITH CONSTRAINT RELAXATION

The overall analysis approach proposed in this paper has been described before, with its lazy abstraction component analyzed in detail. It remains to describe, in more detail, the constraint relaxation that is performed prior to abstraction. Figure 6 depicts the main stages of the overall approach. In this Figure, it can be seen that the process takes an SCR specification *Spec* and an invariant property *P* as inputs. It then tries to verify that *P* holds in *Spec*, in a completely automated way. As a result of the verification process it either returns that *P* holds (property valid in the Figure), or P does not hold and an execution of *Spec* that violates *P* is generated, or the analysis is inconclusive (*Fail*).

A verification attempt consists of two main steps. First, a *relaxed* specification *RSpec* is constructed, by removing **NAT** constraints of the monitored numeric variables of *Spec*. This step is

Prepared using stvrauth.cls
DOI: 10.1002/stvr

discussed in detail below, in section 4.1. Then, the counterexample guided abstraction refinement procedure (CEGAR) described in the previous section is executed, trying to prove that *RSpec* satisfies *P*.

The CEGAR procedure (Algorithm 1 of section 3.4) is invoked with *RSpec* and *P* as inputs. If LPA-SCR proves that *P* is valid in *RSpec*, then *P* is also valid in *Spec* (Theorem 4.1 of section 4.1 states that *RSpec* is a conservative abstraction of *Spec*). Hence, the whole process terminates informing the user that *P* holds. Otherwise, LPA-SCR returns an abstract counterexample: an execution σ^R of *RSpec* that may or may not represent an execution of the original *Spec* violating *P*. (This happens because an abstract counterexample of *RSpec* might involve input events that are forbidden by *Spec*'s **NAT**.)

A second CEGAR process is in charge of dealing with abstract counterexamples σ^R that were deemed concretizable with respect to *RSPec*. This is implemented by the refinement module of Figure 6. First, the concretization leverages existent model checking techniques [10] to explore the state space of abstract counterexample σ^R, looking for an actual execution σ of *Spec* violating *P*. If such a σ exists, the process terminates returning σ as a witness of the violation of *P*. The details for the concretization procedure are given in section 4.2. If σ^R does not encode a concrete execution of *Spec*, it is said that σ^R is spurious. The predicate discovery module tries to learn a logical predicate $Pred$ from spurious σ^R (also using model checking), such that $Pred$ encodes the cause of the σ^R's spuriousness. Furthermore, when added to *RSpec*, the discovered predicate $Pred$ discards σ^R as a valid execution of *RSpec* (and possibly other spurious counterexamples), allowing LPA-SCR to start over. However, the refinement algorithm is not complete, and it might fail in some cases. When this happens, the process must return *Fail*, as it cannot continue trying to prove P.

The remainder of this section is devoted to the technical details of the constraint relaxation phase, as well as the spuriousness checking of *RSPec* counterexamples with respect to *Spec*.

4.1. Relaxing NAT constraints

Recall from previous sections that the behaviour of a monitored variable `mv` can be modelled as a binary relation γ_{mv}, called the transition relation for `mv`. For example, the transition relation for variable `mWaterPres` in *SIS* is as follows:

$$\gamma_{\texttt{mWaterPres}} = \{(x, x') \mid x' \neq x \wedge 0 \leq |x' - x| \leq 10 \wedge x, x' \in \{0 \dots 5000\}\}$$

In this formula, constraint $0 \leq |x' - x| \leq 10$ is imposed by *SIS*' **NAT**. The relaxation step consists of removing the **NAT** constraints that limit the behaviour of monitored numeric variables. In the *SIS* example, removing the aformentioned **NAT** constraint yields a new SCR specification, *RSIS*, where the behaviour of `mWaterPres` is described by relation $\gamma^R_{\texttt{mWaterPres}}$ below:

$$\gamma^R_{\texttt{mWaterPres}} = \{(x, x') \mid x' \neq x \wedge x, x' \in \{0 \dots 5000\}\}$$

This achieves the desired effect of letting `mWaterPres` vary arbitrarily within its domain.

In general, let γ_{mv} below be the transition relation of monitored numeric variable mv in *Spec*:

$$\gamma_{mv} = \{(x, x') : x' \neq x \wedge \mathbf{NAT}(x, x') \wedge x, x' \in TY(mv)\}$$

DOI: 10.1002/stvr

where **NAT**(x, x') is a constraint imposed by **NAT** and $x, x' \in TY(mv)$ enforces the right types for x and x'. The relaxation step removes **NAT**(x, x') from γ_{mv}, producing a new specification γ^R_{mv} as below:

$$\gamma^R_{mv} = \{(x, x') : x' \neq x \wedge x, x' \in TY(mv)\}$$

This process applies the above procedure for each monitored numeric variable mv of *Spec*, to produce *RSpec*. If the LTS for *Spec* is $\Sigma_{Spec} = (S, S_0, E, T)$ and $\mathtt{mv}_1, \ldots, \mathtt{mv}_k$ are *Spec*'s monitored variables, then the LTS for *RSpec* is $\Sigma_{RSpec} = (S, S_0, E^R, T)$, with:

$$\mathtt{mv}^R_i = \{(\mathtt{mv}_i, v, v') \mid (v, v') \in \gamma^R_{mv_i}\} \qquad\qquad E^R = \bigcup_{i=1}^{k} \mathtt{mv}^R_i$$

Notice that, for monitored variable $\mathtt{mv}_i$ that is unconstrained by **NAT** we have that $\gamma_{mv_i} = \gamma^R_{mv_i}$. In contrast, if γ_{mv_i} was altered by the relaxation, then $\gamma_{mv_i} \subseteq \gamma^R_{mv_i}$. Therefore, $\mathtt{mv}_i \subseteq \mathtt{mv}^R_i$ for $i = 1 \ldots k$, and in turn $E \subseteq E^R$. Intuitively, this means that all the input events of *Spec* are events of *RSpec* and that there might be events of *RSpec* that are not events of *Spec*. For example, $e = (\mathtt{mWaterPres}, 14, 904) \in E^R$ (e is an event of *RSIS*) but $e \notin E$ (e is infeasible in *SIS*).

By the above, an execution σ^R of *RSIS* that contains infeasible events of *SIS* is not a valid execution of *RSIS*. Conversely, all the executions σ of *SIS* are valid in *RSIS*. This implies that *RSpec* is an *abstraction* of *Spec*: it possesses the same executions of *Spec* plus many others. It should be clear now that the reachable states of *Spec* are a subset of the reachable states of *RSpec*.

Theorem 4.1
Let $\Sigma_{Spec}, \Sigma_{RSpec}$ be the LTS associated with *Spec* and *RSpec*, respectively.
Then, *Reach*$(\Sigma_{Spec}) \subseteq$ *Reach*(Σ_{RSpec})

Thus, *RSpec* is a *conservative abstraction* of *Spec*: any invariant property that holds in *RSpec* also holds in *Spec*. For this reason, when our analysis process proves an invariant property *P* in *RSpec* it can conclude that *P* holds in *Spec*.

As discussed earlier, and evidenced by the experiments in section 5, the relaxation of **NAT** greatly speeds up the execution of LPA-SCR, as a significantly smaller number of abstraction predicates are required for proving properties of *RSpec* (recall the discussion at the end of section 2). However, executing LPA-SCR with *RSpec* and property P might return an execution σ^R of *RSpec* violating P. An execution σ^R is called an *RSpec counterexample*, that might or might not encode a valid execution of *Spec* violating P. In the next section, it is discussed how this approach analyzes RSpec counterexamples, building real counterexamples of *Spec*, or getting rid of them if they are spurious.

4.2. Dealing with RSpec counterexamples

The general form of an RSpec counterexample, returned by LPA-SCR during an attempt to prove property P, is shown below:

$$\sigma^R = s^{\mathcal{A}}_0 \; mv^R_0 \; s^{\mathcal{A}}_1 \; mv^R_1 \ldots mv^R_{m-1} \; s^{\mathcal{A}}_m \tag{1}$$

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Furthermore, $S_0 \subseteq [s_0^{\mathcal{A}}]$, $post^{\mathcal{A}}(s_i, mv_i) = s_{i+1}$ for all i, and P must not appear positively in φ_m (otherwise σ^R would not represent a violation of P).

Counterexamples σ^R of *RSpec* are dealt with differently depending on whether they only involve feasible events of *Spec* or not. In the first case, one has $mv_i^R = mv_i$ for any monitored variable mv_i. Hence, σ^R only represents valid executions of *Spec*. The $solve$ procedure of LPA-SCR (Algorithm 1) can be employed to generate a valid execution of *Spec* violating *P* from σ^R. Notice that σ^R must represent some valid execution of *RSpec*, otherwise it would have been removed by the refinement algorithm of LPA-SCR.

In the second case, there is $mv_i^R \in E^R$ such that $mv_i^R \notin E$, that is, σ^R involves at least an event of *RSpec* that is infeasible in *Spec*. Therefore, σ^R might represent executions that are infeasible in *Spec* due to **NAT**. Thus, using $solve$ to generate a concrete execution from σ^R might produce a spurious counterexample: an execution that is infeasible in *Spec*.

Model checking is employed to generate only feasible executions of *Spec* from the latter type of abstract counterexamples, avoiding false positives. To simplify the explanation of the technique, let us assume that there is only one event mv_j in (1) such that $mv_j^R \notin E$ (extending the technique for the general case is trivial). Then, (1) can be rewritten as:

$$\sigma^R = s_0^{\mathcal{A}}\ mv_0\ s_1^{\mathcal{A}} \ldots s_j^{\mathcal{A}}\ mv_j^R\ s_{j+1}^{\mathcal{A}} \ldots s_{m-1}^{\mathcal{A}}\ mv_{m-1}\ s_m^{\mathcal{A}} \tag{2}$$

All feasible executions of *Spec* represented by σ^R, denoted by $[\sigma^R]$, have to be explored to find out if there exists one that violates *P*.

Let us formally define what it means for an execution σ of *Spec* to be represented by σ^R.

$$\sigma \in [\sigma^R] \Leftrightarrow \exists s_0 \in [s_0^{\mathcal{A}}], \ldots, s_k \in [s_k^{\mathcal{A}}] \mid s_0 \overset{mv_0}{\rightsquigarrow} s_1 \ldots s_j \overset{mv_j{}^+}{\rightsquigarrow} s_{j+1} \ldots s_{k-1} \overset{mv_{k-1}}{\rightsquigarrow} s_k$$

Notice that the effect of mv_j^R ($\in E^R$) is "simulated" by executing a sequence of events corresponding to $mv_j \in E$: this is denoted by $s_j \overset{mv_j{}^+}{\rightsquigarrow} s_{j+1}$ (see section 2.1.2 for the definition of $mv_j{}^+$). Also, observe that due to the definition of $[\sigma^R]$, the resulting σ is an execution of *Spec*.

This process encodes $[\sigma^R]$ in Promela, the input language of the model checker SPIN, and then asks SPIN to find a counterexample $\sigma \in [\sigma^R]$ of P. The encoding consists of a modified version of the approach introduced by Bharadwaj and Heitmeyer [10] for the analysis of whole SCR specifications. The approach by Bharadwaj and Heitmeyer [10] is discussed in section 4.2.2. In section 4.2.1, an example illustrating the concepts introduced in this section is also provided. Furthermore, this example will be employed in section 4.2.3, where the encoding of counterexamples in Promela is described.

4.2.1. An example. Let P be the following (invalid) *SIS* property: `mcPressure` $=$ `Permitted` $\wedge$ `tOverriden` $\rightarrow$ `mWaterPres` $\neq$ `Permit-1`. Feeding LPA-SCR with *RSIS* and P yields the abstract counterexample σ^R_{RSIS} of Figure 7.

In this figure, below each abstract state $s_i^{\mathcal{A}}$, the formula yielded by $[s_i^{\mathcal{A}}]$ is indicated, describing all the states represented by $s_i^{\mathcal{A}}$. The `pc=i` annotations in the Figure can be ignored for the moment.

σ^R_{RSIS} indicates how to obtain an execution σ of *SIS* ($\sigma \in [\sigma^R_{RSIS}]$) violating P. It states that one must start at the initial state (represented by $s_0^{\mathcal{A}}$ in the Figure) and modify monitored variable

Prepared using stvrauth.cls
DOI: 10.1002/stvr

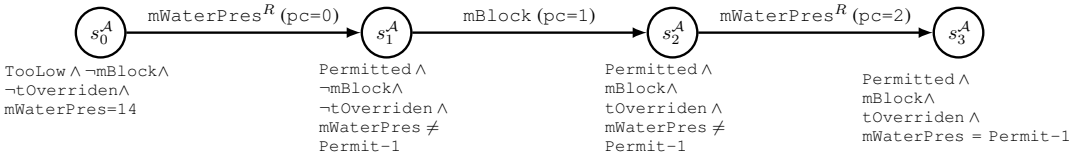

Figure 7. Graphical representation of abstract counterexample σ^R_{RSIS}.

mWaterPres several times, until a state with mode Permitted and where mWaterPres is not equal to Permit-1 yet, is reached (represented by s_1^A). Then, the block switch has to be activated, making the system behaviour overridden in the next state (tOverriden holds at s_2^A). Notice that P is satisfied at states characterized by s_2^A. But, modifying mWaterPres to make it equal to Permit-1 leads to a state where P does not hold (represented by s_3^A).

Section 4.2.3 describes the proposed encoding for $[\sigma^R_{RSIS}]$. Let us now discuss the original SCR encoding in Promela put forward by Bharadwaj and Heitmeyer [10].

4.2.2. Encoding SCR specifications in Promela. This section explains the approach by Bharadwaj and Heitmeyer [10] by means of an example, using the *SIS* specification. For further details, the reader is referred to the work of Bharadwaj and Heitmeyer [10]. A snippet of the translation of *SIS* into Promela is shown in Figure 8.

Variable declarations are omitted in Figure 8 due to space restrictions. However, it is worth mentioning that two SPIN variables (with the right type) must be declared for each *SIS* entity, storing the pre and post values of the entity at each transition of the system. For example, the Promela code involves variables mWaterPres and mWaterPresP (of numeric type), denoting the pre and post values of mWaterPres, respectively. In addition, it is assumed that all variables are initialized to their values in the *SIS* initial state.

Recall that the system described by an SCR specification is always reacting to input events triggered in its environment. Hence, the behaviour of the system is modelled as an infinite loop: see lines 1-44 of Figure 8. Each iteration of the loop atomically performs the following sequential steps: a unique input event (modifying a monitored variable) is triggered nondeterministically (lines 3-19); the SCR transition function (defined by the tables) is used to compute the next state of the system, by propagating the changes carried out by the input event to the remaining entities (lines 21-39); the new values for variables become the current values, so the next iteration starts at the newly computed state (lines 40-41). The **atomic** keyword of line 2 makes SPIN to execute each iteration indivisibly.

The translation heavily employs Promela's **if..fi** statements. These are typical nondeterministic guarded alternative command: exactly one guarded command whose guard is true at the current state is picked for execution. True guards can be ommited, as is the case with the outer **if** of lines 3-19. Thus, each of the inner **if**'s of lines 4-7, 8-11, 12-18 model the execution of a different input event. For example, the new value of mBlock can be set to Off in line 5, if it is currently On (the guard is mBlock!=Off), and vice versa in line 6.

To understand the translation of the SCR transition function, consider the available mode transitions when the current mode is TooLow, depicted in lines 23-26 of the Figure. Line 24 states that when the water pressure is currently below constant Low and it becomes greater or equals to

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
1   do ::
2    atomic{
3      if /* an input event is triggered nondeterministically */
4      :: if /* mBlock is modified */
5         :: mBlock!=Off -> mBlockP = Off
6         :: mBlock!=On -> mBlockP = On
7         fi;
8      :: if /* mReset is modified */
9         :: mReset!=Off -> mResetP = Off
10        :: mReset!=On -> mResetP = On
11        fi;
12     :: if /* mWaterPres is modified respecting NAT */
13        :: mWaterPresP = mWaterPres + 1
14        :: mWaterPresP = mWaterPres - 1
15        :: ...
16        :: mWaterPresP = mWaterPres + 10
17        :: mWaterPresP = mWaterPres - 10
18        fi;
19     fi;
20    d_step{ /* encoding of the transition relation defined by tables */
21     if /* mode transition table */
22     :: mcPressure == TooLow ->
23        if
24        :: (mWaterPresP >= Low) && (!(mWaterPres >= Low)) ->
25                                          mcPressureP = Permitted;
26        :: else skip;
27        fi;
28     :: mcPressure == Permitted ->
29        if
30        :: (mWaterPresP < Low) && (!(mWaterPres < Low)) ->
31                                          mcPressureP = TooLow;
32        :: (mWaterPresP >= Permit) && (!(mWaterPres >= Permit)) ->
33                                          mcPressureP = High;
34        :: else skip;
35        fi;
36     :: mcPressure == High ->
37        if
38        :: (mWaterPresP < Permit) && (!(mWaterPres < Permit)) ->
39                                          mcPressureP = Permitted;
40        :: else skip;
41        fi;
42     fi;
43     /* event and condition tables */
44     mBlock = mBlockP; mReset = mResetP; mWaterPres = mWaterPresP;
45     mcPressure = mcPressureP;
46     tOverriden = tOverridenP;
47     cSafetyInjection = cSafetyInjectionP;
48    }/* end of d_step */
49    }/* end of atomic */
50  od
```

Figure 8. Promela encoding for *SIS*.

`Low`, then the system must transition to mode `Permitted`. In any other case, the mode must be kept the same; this is achieved by Promela's **skip** command in line 26.

Some final remarks on the translation of the transition function are in order. The **d_step** keyword of line 20 orders SPIN to execute the transition function atomically and deterministically. That is, **d_step** enables SPIN to pick a fixed, deterministic way of executing any nondeterminism in the code enclosed by it, thereby improving its runtime performance. This works well for the transition

Prepared using stvrauth.cls
DOI: 10.1002/stvr

function because it is deterministic. In addition, the transition function must respect the dependency relation D of the specification. If entity e_1 depends on entity e_2, i.e., the value of e_2 must be computed before the value of e_1, then the Promela code for the table defining e_2 must appear before the code for the table defining e_1.

4.2.3. Encoding RSpec counterexamples in Promela. This section describes how the encoding of the previous section is adapted to deal with *RSpec*'s counterexamples. Again, the presentation is driven by means of an example. Let σ^R_{RSIS} be the counterexample given in Figure 7, explained in section 4.2.1. Part of the Promela code describing the set of executions (of *Spec*) represented by σ^R_{RSIS} ($[\sigma^R_{RSIS}]$) is shown in Figure 9.

As in the previous encoding, it is assumed that variables are initialized to their values at the initial state. Let w be the number of events in σ^R_{RSIS}. An integer variable `pc` such that `pc=i`, $0 \leq i < w$, is introduced to indicate that events modifying variable mv_i of σ^R_{RSIS} are being executed. Notice the `pc=i` annotations within brackets at the left of each event in Figure 7. In addition, `pc=w` holds at the last state of σ^R_{RSIS} (although `pc=3` is not shown in the Figure). Notice that the last (abstract) state of σ^R_{RSIS} represents states violating the property being verified, P. Thus, `pc=w` indicates that a violation contained in σ^R_{RSIS} has been found. Variable `pc` is set to `0` at the initial state.

The encoding of $[\sigma^R_{RSIS}]$ consists of a main loop (lines 1-41), where each iteration triggers an input event modifying the monitored variable corresponding to the current value of `pc`. At the beggining of each iteration the approach asserts that `pc!=3`, using Promela's **assert** clause (line 4). If the model checker is currently at a state that violates the assert, i.e., `pc=3`, it returns the whole execution that took it from the initial state to the current state. Due to the encoding, this is an execution of *Spec* contained in $[\sigma^R_{RSIS}]$ that violates P. Otherwise, the model checker continues its execution starting from the line following the assert.

In lines 5-9 monitored variable `mVar` is selected according to the value of `pc` (see the `pc=i` annotations in Figure 7). Then, in lines 10-27 an input event modifying `mVar` is triggered nondeterministically –the guards added to the outermost **if** ensure that only `mVar` can be modified at the current iteration.

In lines 28-32 the transition relation is employed to obtain the new system state, resulting from executing the selected event (modifying `mVar`). As before, the last step is executed atomically and deterministically (see the comments about **d_step** in the previous section).

The key step of the encoding is how `pc` is updated. This is done in lines 33-40. Figure 7 starts at an abstraction of the initial state, ${s_0}^{\mathcal{A}}$, where `pc=0`. `pc=0` has associated event `mWaterPres`^{R}, meaning that several events modifying `mWaterPres` have to be executed until ${s_1}^{\mathcal{A}}$ is reached and `pc` becomes 1. Intuitively, line 34 indicates that a feasible option when `pc=0` is to stay with `pc=0`, hence executing another event modifying `mWaterPres`. The other possible option is, when ${s_1}^{\mathcal{A}}$ has been reached, to set `pc=1`, as shown in line 35. This means that one event modifying `mBlock` has to be executed (since `mBlock` is not an event of *RSIS*), and transition to `pc=2`. This is done in line 36. The event associated to `pc=2` is `mWaterPres`^{R} again, and proceeds similarly as when `pc=0`.

If the model checker reaches a state with `pc=3` it returns a witness of P not holding in *Spec*. Otherwise, $[\sigma^R_{RSIS}]$ is spurious (it does not represent any feasible execution of *Spec*), and must be

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
1  do ::
2    atomic{
3      /* If we reach the last state a witness has been found */
4      assert (pc!=3);
5      if /* pick a monitored variable mVar according to PC */
6      :: pc==0 -> mVar=c_WP
7      :: pc==1 -> mVar=c_BL
8      :: pc==2 -> mVar=c_WP
9      fi;
10     if /* an event modifying mVar is triggered */
11     :: (mVar==c_BL) -> if /* mBlock is modified */
12                        :: mBlock!=Off -> mBlockP = Off
13                        :: mBlock!=On -> mBlockP = On
14                        fi;
15     :: (mVar==c_RS) -> if /* mReset is modified */
16                        :: mReset!=Off -> mResetP = Off
17                        :: mReset!=On -> mResetP = On
18                        fi;
19     :: (mVar==c_WP) -> if /* mWaterPres is modified */
20                         :: mWaterPresP = mWaterPres + 1
21                         :: mWaterPresP = mWaterPres - 1
22                         :: ...
23                         :: mWaterPresP = mWaterPres + 10
24                         :: mWaterPresP = mWaterPres - 10
25                         fi;
26     :: else -> break
27     fi;
28   d_step{
29     /* encoding of the transition relation defined by SCR tables */
30     mBlock = mBlockP; mReset = mResetP; mWaterPres = mWaterPresP;
31     mcPressure = mcPressureP;
32     tOverriden = tOverridenP;
33     cSafetyInjection = cSafetyInjectionP;
34   }/* end of d_step */
35     if
36     :: pc==0 -> skip
37     :: pc==0 && (mcPressure==Permitted && !mBlock &&
38                             !tOverriden && mWaterPres!=Permit-1) -> pc++
39     :: pc==1 && (mcPressure==Permitted && !mBlock &&
40                             tOverriden && mWaterPres!=Permit-1) -> pc++
41     :: pc==2 -> skip
42     :: pc==2 && (mcPressure = Permitted && mBlock &&
43                             tOverriden && mWaterPres==Permit-1) -> pc++
44     fi;
45   }/* end of atomic */
46 od
```

Figure 9. Promela encoding of the *SIS* counterexample σ^{R}_{RSIS} of Figure 7.

removed from *RSpec* for the analysis to continue the verification process. This is the topic of the next section.

A comparison of Figures 8 and 9 shows the main advantage of analyzing abstract counterexamples instead of the whole specification: the order of execution of input events is (mostly) fixed in abstract counterexamples, whereas they can be triggered in any order in the whole specification. Thus, there are significantly less interleavings to be considered by the model checker when analysing abstract counterexamples, and therefore it can achieve better runtime performance. The experimental evaluation presented in section 5 supports this claim.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Finally, it is important to remark that the ideas underlying this analysis approach are independent of the selection of a specific model checking tool, and it can be easily adapted to work with different model checkers.

4.3. Refinement phase

The refinement module of the *RSpec* analysis takes as input a spurious ***RSpec*** counterexample σ^R, deemed as spurious by the concretization approach of the previous section, and attempts to produce a predicate $Pred$ that discards σ^R when added to ***RSpec***. That is, counterexample σ^R originates from an abstract counterexample that could be concretized with respect to the relaxed specification (without NAT constraints), but which was unrealizable once NAT constraints are taken into account.

Let us now introduce some notation that will be useful in the rest of this section. Let σ^R be as in formula (1) (section 4.2). The prefix of σ^R up to state i, $i \leq m$, is denoted by $\sigma^R(..i)$. Notice that the concretization operator is well defined for prefixes of abstract counterexamples. Thus, $[\sigma^R(..i)]$ denotes the concretization of the prefix $\sigma^R(..i)$.

In order to attempt to learn $Pred$ from σ^R, the refinement approach starts looking for the cause of infeasibility of σ^R in a forward manner. That is, it starts an iterative process, initializing a variable i in 1, and finishing when $i = m$. At each iteration, the approach checks whether $[\sigma^R(..i)]$ contains at least an execution of ***Spec***, using the technique presented in the previous section (i.e., translating $[\sigma^R(..i)]$ to Promela and running SPIN over the translation). If such an execution exists, $[\sigma^R(..i)]$ is not spurious and i is increased by one to start the next iteration. Once the smallest spurious prefix $[\sigma^R(..i)]$ is found, it will proceed by considering that transition $s^{\mathcal{A}}_{i-1}\ mv^R_{i-1}\ s^{\mathcal{A}}_i$ might be the culprit.

The approach consists simply in employing a model checker, in the same way as done in the previous section, to explore $[s^{\mathcal{A}}_{i-1}\ {mv_{i-1}}^R\ s^{\mathcal{A}}_i]$, i.e., to try to find an execution σ of ***Spec*** that takes the system from a state represented by $s^{\mathcal{A}}_{i-1}$ to another state represented by $s^{\mathcal{A}}_i$, via the execution of an event modifying variable mv_{i-1}. If no such execution exists, then one can take $Pred = [s^{\mathcal{A}}_{i-1}] \wedge mv^R_{i-1} \Rightarrow \neg[s^{\mathcal{A}}_i]$; this ensures that starting at a state where $[s_{i-1}]$ holds, the execution of event mv^R_{i-1} of ***RSpec*** cannot lead to a state where $[s^{\mathcal{A}}_i]$ holds. $Pred$ is then added to ***RSpec***, allowing the analysis to get rid of the spurious σ^R. On the other hand, if there exists a sequence σ witnessing $[s^{\mathcal{A}}_{i-1}\ {mv_{i-1}}^R\ s^{\mathcal{A}}_i]$, then the assumption that the last transition was responsible of the spuriousness was misleading. In this case the process simply terminates the analysis, regarding it *inconclusive*.

Notice that this *lightweight* approach for refinement is incomplete: the last transition is considered in isolation, so the feasibility of the transition may not imply the feasibility of the whole prefix. One may of course find an appropriate refinement for ***RSpec***, to remove the spurious counterexample, but the approach would derive in a second, standard CEGAR process again. By taking this simpler approach one may be missing cases, of course. However, as shown in the experimental evaluation, for the analyzed models one seldom falls into this situation.

To learn about the spuriousness of $s^{\mathcal{A}}_{i-1}\ mv^R_{i-1}\ s^{\mathcal{A}}_i$ the approach employs a translation from SCR to the input language of the model checker ALV, similar to the one presented by Bultan and Heitmeyer [12] (and conceptually similar to the translation to Promela given in the previous section). This translation will not be discussed here; the interested reader should refer to the work of Bultan and Heitmeyer [12]. The choice of ALV over SPIN here is based on an experimental comparison of the efficiency of both model checkers when analyzing $s^{\mathcal{A}}_{i-1}\ mv^R_{i-1}\ s^{\mathcal{A}}_i$. While in the experiments SPIN was often (much) better at generating witnesses for abstract counterexamples, ALV had a

DOI: 10.1002/stvr

(large) edge in proving them spurious. This might have to do with the different natures of the tools: SPIN is an explicit model checker (it stores traversed states explicitly), whereas ALV is symbolic (it represents set of states by formulas, in an attempt to reduce the required memory space). This conclusion matches that of Bultan and Heitmeyer [12], who also compared these model checkers. Furthermore, ALV makes it easier to describe a set of initial states, as in $s^{\mathcal{A}}_{i-1}\ mv^{R}_{i-1}\ s^{\mathcal{A}}_{i}$, than SPIN, and ran faster in the experiments when a large number of initial states were present. As ALV suffers from performance problems when generating witnesses, a timeout of 1 minute is set for the analysis of $s^{\mathcal{A}}_{i-1}\ mv^{R}_{i-1}\ s^{\mathcal{A}}_{i}$. If ALV does not answer before the timeout, the refinement process fails.

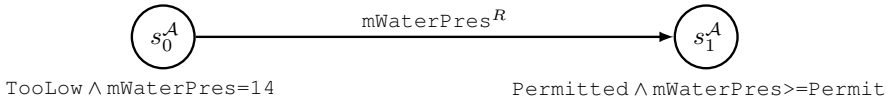

Figure 10. A spurious counterexample σ^R of *RSIS*

4.3.1. Refinement example. Consider the counterexample σ^R (of *RSIS)* shown in Figure 10, that took place during the experiments of section 5. In σ^R the system starts at the initial state, and moves to an abstract state with mode `Permitted` where `mWaterPres>=Permit` holds, by executing *RSIS*' event mWaterPres^R. However, σ^R is spurious in the original *SIS*, as `mWaterPres>=Permit` triggers a transition to mode `High` (starting from mode `Permitted`), and the only way to go back to a state in mode `Permitted` is when `mWaterPres` takes a value lower than `Permit`. ALV can prove that σ^R is infeasible in *SIS* in a few seconds. Then, the refinement process produces the following predicate that removes σ^R from *RSIS*:

$$Pred \;=\; (\texttt{mcPressure=TooLow} \wedge \texttt{mWaterPres'} \neq \texttt{mWaterPres} \Rightarrow \neg(\texttt{mcPressure'=Permitted} \wedge \texttt{mWaterPres'>=Permit}))$$

5. EXPERIMENTAL EVALUATION

The experimental evaluation is presented in order to answer the following research questions:
RQ1) Is the lazy abstraction approach (algorithm LPA-SCR) better suited than traditional lazy abstraction to analyse SCR specifications?
RQ2) is the whole analysis approach more efficient than alternative techniques to analyze SCR specifications maintaining the original level of detail?

In order to answer RQ1, in section 5.1, the effects of localizing abstraction predicates to modes and modularizing the transition relation, when analyzing SCR specifications, is evaluated. To answer RQ2, the whole analysis process is compared with alternative techniques for automated test case generation and verification of properties of SCR specifications, in Sections 5.2 and 5.3, respectively.

The evaluation is based on various case studies taken from the literature on SCR specification and analysis [21]. These are: a cruise control system (`ccs`), the safety injection system (`sis`) used as a running example throughout this article, an aircraft's autopilot (`autopilot`), and a car overtaking protocol for coordinating smart vehicles (`car3prop`). All the experiments were run on an 2.6GHz Intel Core 2 Duo PC with 3GB of RAM (2.5GB maximum memory set for the analysis tools),

Prepared using stvrauth.cls
DOI: 10.1002/stvr

running GNU/Linux 3.0. The prototype tool implementing the proposed analysis approach as well as the case studies reported here can be downloaded from `http://dc.exa.unrc.edu.ar/staff/rdegiovanni/case-studies/SCR_Analysis.zip`.

Generating Test Cases from SCR specifications

Let us briefly explain in what consists the test case generation problem for SCR specifications. Intuitively, a *test case* is a run of the SCR specification (see Section 2.1.2) that covers certain functionality. More precisely, a test case is a sequence of input events, together with the outputs that the events produce in the SCR specification. Then, these test cases can be used for *validation* (contrasting if the actual specification captures the user expected behaviour) and *verification* (checking if the implementation meets the requirements specification).

In order to use the presented approach to generate test cases for SCR specifications, we follow various different techniques based on model checking, that have been introduced in the literature [5, 21, 22]. First, one needs to build all *test predicates* corresponding to the coverage criterion of interest. Each of these test predicates characterizes a particular equivalence class of test cases, in the corresponding test criterion. Then, for each test predicate P, a *trap property* $TP = \neg P$ is generated. Then, the presented approach is executed with the property TP as input. If the approach generates a counterexample for TP, then a run that reaches a state satisfying P, i.e., a *test case* for P, has been found. Otherwise, P is an infeasible test predicate.

Several different coverage criteria have been proposed for SCR specifications. In order to assess the presented approach, the following are considered:

- *Table coverage* (T): Every cell of every table is covered at least once.
- *Split mode coverage* (SM): If a cell refers to various modes, the cell is covered for each of the modes separately.
- *Disequality split* (DS): Expressions containing disequality operators are covered for the "cases" of the disequality, e.g., $\geq$ is split into $>$ and $=$.
- *Boundary coverage* (B): Boundary values of disequalities in expressions are covered. For example, $y > C$ (with x and C integers) is split into $x = C + 1$ and $x > C + 1$.
- *Modified Condition Decision Coverage* (MCDC): Each literal (condition) is shown to independently affect the value of the expression (decision) it is part of.

5.1. Assessing the Lazy Abstraction Algorithm

The goal in this section is to measure the effect of the two key features of the lazy abstraction algorithm presented in this paper, in Section 3: modes are used for localizing abstraction predicates, and the SCR transition relation is modularized according to modes and events. Notice that here it is not the full approach what is being assessed (in particular, the effect of relaxing the NAT relation of the specification is not evaluated). The SCR lazy abstraction algorithm is evaluated using different configurations:

- LPA: Standard lazy predicate abstraction, with modes not part of the initial abstraction, support predicates local to regions, and no modularization of the transition relation.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

- LPA + m.: LPA plus modes in the initial abstraction, and support predicates local to modes.
- LPA + mt.: LPA plus a modularized transition relation.
- LPA + m. + mt.: The LPA-SCR algorithm, which employs all the optimization techniques discussed above, including m. and mt.

This experimental evaluation compares different configuration of the approach presented in this paper, to assess how its different components affect performance. In order to be able to observe the impact, models that are amenable to *all* configurations have to be used. This unfortunately is not the case for most of the original versions of the models considered in this paper: the only model that can in fact be analyzed *as is* with all configurations is `car3prop`, because it does not feature numeric domains. To have a more representative analysis the literature was sought for versions of the considered models that can be subject to all analyses. As a result of this search, *reduced* versions for two of the cases were found, namely `autopilot reduced` and `sis reduced` [21]. To have a hint of the extent of the simplification in the specifications, e.g., `autopilot reduced` deals with integer variables in the range $[0..500]$, while the original `autopilot` has these same variables over the range $[0..10000]$.

Table I summarizes the results of the experiment. The first row of the table shows the name of the case study, and in brackets, the number of test predicates to be covered. Column runs indicates the number of times a technique had to be executed in order to generate tests cases to cover all of the test predicates, for all criteria. Column c/i/u displays the number of covered (c), infeasible (i), and uncovered (u) test predicates. Covered predicates are those for which a test case could be generated, infeasible predicates are those marked unrealizable by the corresponding technique, and uncovered predicates represent cases where the technique was inconclusive (errors). For the covered test predicates, the number of runs needed is indicated in brackets, since a single run can cover several test predicates. Column time indicates how many seconds took a technique to generate tests cases for all the considered test predicates. It is important to remark that any individual run that lasted more than one hour, was stopped and marked the corresponding test predicate as uncovered.

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
LPA	110	402(14)/90/**6**	33620	19	80(4)/11/**0**	8496	21	395(7)/10/**4**	23497
LPA + m.	114	401(17)/96/**1**	8858	21	80(10)/11/**0**	5319	81	347(19)/10/**52**	113401
LPA + mt.	118	402(22)/74/**22**	69197	20	80(9)/11/**0**	13032	51	389(31)/10/**10**	22201
LPA-SCR	119	402(29)/96/**0**	1795	23	80(12)/11/**0**	4724	54	399(44)/10/**0**	3951

Table I. Assessment of our LPA-SCR algorithm.

The results in Table I point out that the LPA-SCR approach is 6X faster than standard lazy predicate abstraction (LPA). More importantly, it is able to cover all the test predicates, as opposed to the alternatives. This happens because either lazy abstraction needs to handle many (non local to modes) abstraction predicates, or it had to rediscover too many predicates (see below), making the abstract model too expensive to construct within the given time limits.

In the LPA-SCR algorithm, support predicates are local to modes instead of regions. It was argued earlier in this paper that, due to the structure of tables, regions with the same mode tend to share the support predicates. In order to validate this hypothesis, a few test predicates were randomly chosen for each case study, and gathered the total number of predicates required to generate each test case,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the maximum number of times a predicate has been rediscovered for different regions with the same mode during standard abstraction, and the number of abstract nodes constructed by each approach. These results are summarized in Table II.

CS	LPA			LPA-SCR	
	nodes	predicates	most repeated	nodes	predicates
car3prop P1	41	94	30	76	11
car3prop P2	48	75	33	75	8
sis reduced P1	916	158	50	936	65
sis reduced P2	113	25	8	113	17
autopilot reduced P1	1037	126	21	1192	32
autopilot reduced P2	965	126	21	1120	32

Table II. Predicates local to modes vs. predicates local to regions.

Clearly, LPA-SCR is improved by localizing support predicates using the modes of the specification, as this allows it to significantly reduce the number of support predicates needed, and the number of invocations to the abstraction refinement procedure. The aforementioned results are the main cause of the good behaviour (the competitive performance) of LPA-SCR against lazy abstraction, shown in Table I. It is important to remark that models involving numeric variables over large ranges, such as `sis`, are those in which standard LPA expends more effort in rediscovering support predicates. This difference becomes notoriously more evident when the original SCR specifications are analyzed (recall that here reduced versions of the models were considered), where more predicates are needed, and so, rediscovered.

5.2. Experimental Results for Test Generation

In this section, the approach REL-LPA-SCR is assessed against alternative techniques for automated test case generation of SCR specifications. In particular, a broad assessment of several model checkers for this task was presented by Fraser and Gargantini [21]. Then, the approach is compared against the model checkers used by Fraser and Gargantini [21], on the case studies already mentioned: `ccs`, `sis`, `autopilot` and `car3prop`. It is important to remark that, for the experiments in this section, the *full* specifications are considered, maintaining numeric data as in the original description of the specifications (i.e., not manually reduced versions as in the work of Fraser and Gargantini [21]). Several model checkers with different features are considered (explicit state, symbolic and bounded model checkers) for this comparison: SPIN, NuSMV, Cadence SMV and SAL. Three new tools are also considered: CPAChecker [8], a tool for predicate analysis of C programs, based on lazy abstraction and interpolation-based refinement; Randoop [44], a feedback-directed random test generation tool for Java programs; and Evosuite [20], an automated test case generation tool for Java programs based on evolutionary computation. CPAChecker is an example of a modern, efficient CEGAR-based model checker. The other two tools are examples of state-of-the-art technology for automated test case generation, that are known to perform very well in practice, and do not suffer from the scalability issues that model checkers are often subject to. To run CPAchecker, a C program per each case study was built, where the nondeterminism associated with the alterations on monitored variables is captured through auxiliary routines of the kind `__VERIFIER_nondet_int()`, that the tool provides. To run both Randoop and Evosuite, a Java class per each case study was developed, in which public methods capture the changes in

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the monitored variables; by producing tests that exercise these classes through their corresponding APIs, test sequences for the corresponding cases are produced. All classes and auxiliary programs are provided as part of the bundle for reproducing the experiments, reported earlier in this paper. These tools were run using a variety of settings, and only the best result produced by each tool is reported. When a model checking tool is not mentioned for one of the case studies, it means that it performed significantly worse than the other model checkers. All tools except for Randoop were run once per test predicate. Randoop, on the other hand, since it is not driven by coverage, was let to run for a whole hour, and then the resulting suite was evaluated, to analyze how many test predicates were covered.

Table III shows the results of this experiment. For each case study and technique, the table shows the number of times the tool was invoked (runs), the number of covered (c), infeasible (i), and uncovered (u) test predicates (when the technique is inconclusive). In each of the covered cases, it has been indicated, between parentheses, the number of tests produced by each tool, as a single test may cover several test predicates. As in the previous section, here all the test criteria are considered to obtain the test predicates (indicated in the table next to the name of each case study). The column time indicates the total running time, in seconds, for each tool. Additionally, for the whole analysis approach, the time consumed by the LPA-SCR algorithm, the time required by the model checkers to build a feasible counterexample (M.C. Conc.), and to produce the constraints that remove spurious violations (M.C. Ref.), are distinguished. Since the original (large) specifications are dealt with, the timeout for covering a single test predicate is set to 5 minutes, and a total analysis time of 5 hours for each technique. The only exception was Evosuite; a test suite per each predicate to be covered was produced; since the tool has a default timeout of 2 minutes to attempt to achieve coverage, the tool was run with this default configuration and the total generation time was summed up; the tool may achieve generation before the timeout, but continues to run to improve (e.g., reduce the size) of the test suite. While this provided an advantage to the tool, it still performed worse that other approaches, so the whole results were reported instead of only those that could be covered within the 5 hours timeout.

Let us discuss the results presented in Table III. The first issue to notice is that the presented technique is able to generate test cases for almost all predicates (only one error in the `autopilot` due to the incompleteness of the abstraction refinement approach), even when other tools fail to do so. This is more evident in large case studies, like `ccs` and `autopilot`, where the models contain many monitored variables with large numeric domains.

In the largest case study, the `ccs` model (31 variables, where 10 are numeric with ranges from 0 to 999999), the presented approach (REL-LPA-SCR) does not fail for any test predicate, whereas most other tools consistently fail due to the huge state space to be explored. SPIN is the only model checker able to generate test cases for some predicates (about 25% of the total). However, it can only handle "easy" test predicates, that is, those for which the corresponding tests can be generated by exploring just a small part of the state space (notice that SPIN never marks any predicate infeasible, since it cannot explore the entire state space). Being an explicit state model checker, SPIN is very fast in generating test cases, but it generally runs out of memory quickly for large models.

The second largest specification, `autopilot` (10 variables, 6 of those are numeric with range from 0 to 10000), shows a similar behaviour to `css`, although the model checkers can cover more test predicates in this case (close to 70% with SPIN). Despite the fact that the presented approach is

Prepared using stvrauth.cls
DOI: 10.1002/stvr

	Runs	Tests (c/i/u)	Time	LPA-SCR Time	M.C. Conc.	M.C. Ref.
autopilot (409 TPs.)						
SPIN	169	288(48)/0/**121**	2351	-	-	-
Cad. SMV	378	36(5)/0/**373**	timeout	-	-	-
SAL/SMC	296	116(3)/0/**293**	timeout	-	-	-
CPAchecker	409	275(275)/8/**126**	118323	-	-	27315
Randoop	1	201(160025)/0/**208**	3600	-	-	-
Evosuite	409	0/0/**409**	21275	-	-	-
REL-LPA-SCR	234	398(223)/10/**1**	1390	227	848	313
ccs (582 TPs.)						
SPIN	582	164/0/**418**	3456	-	-	-
Cad. SMV	582	0/0/**582**	timeout	-	-	-
CPAchecker	582	328(328)/83/**171**	183661	-	-	29545
Randoop	1	139(192073)/0/**443**	3600	-	-	-
Evosuite	582	0/0/**582**	62696	-	-	-
REL-LPA-SCR	457	482(357)/100/**0**	4657	1402	1602	1472
sis (91 TPs.)						
SPIN	19	80(8)/11/**0**	146	-	-	-
NuSMV	27	80(16)/11/**0**	995	-	-	-
Cad. SMV	31	80(20)/11/**0**	420	-	-	-
SAL/SMC	27	80(16)/11/**0**	38	-	-	-
CPAchecker	91	24(24)/11/**56**	50530	-	-	20672
Randoop	1	24(153782)/0/**67**	3600	-	-	-
Evosuite	91	0/0/**91**	6484	-	-	-
REL-LPA-SCR	70	80(59)/11/**0**	156	13	111	30
car3prop (498 TPs.)						
NuSMV	142	402(46)/96/**0**	261	-	-	-
Cad. SMV	160	402(64)/96/**0**	78	-	-	-
SAL/BMC	133	402(37)/81/**15**	56	-	-	-
CPAchecker	498	345(345)/15/**138**	130025	-	-	19947
Randoop	1	264(289)/0/**234**	3600	-	-	-
Evosuite	498	259(13526)/0/**239**	65730	-	-	-
REL-LPA-SCR	460	402(364)/96/**0**	2509	2059	450	0

Table III. Comparison between the whole analysis approach (REL-LPA-SCR) and other test generation tools in the generation of test cases for SCR.

clearly better in this case study, this is the only case in which our approach reports one error: it runs out of time (5 minutes) when SPIN is run to build a concrete counterexample. Similar to the `ccs`, notice that no model checker can mark a predicate as infeasible for the `autopilot`, due to the big state space to be explored.

However, for small specifications some model checkers can be faster than the presented approach. For example, the `sis` case study contains just three monitored variables (only one of those is numerical), so it does not represent a big problem for the model checkers. In particular, SAL (a bounded model checker) is faster than the presented approach, but the times obtained with the technique are still comparable with the other model checkers. For the `car3prop` model, all the model checkers are faster than the approach. It is important to remark that `car3prop` does not have any numeric variable, and has a large number of monitored variables. Thus, the absence of numeric variables allows model checkers to explore the full state space very efficiently. On the other hand, the presented analysis approach does not benefit from relaxing the NAT relation, and due to the characteristics of the model, it has to introduce a high number of abstraction predicates, yielding worse running times than the model checkers.

CPAchecker is included in this evaluation to have a comparison with a standard CEGAR based model checking approach. While the case studies had to be encoded as C programs (extended with

Prepared using stvrauth.cls
DOI: 10.1002/stvr

nondeterministic assignments and other elements), this is not different from other encodings in Promela, that have been described earlier in the paper. The tool showed a poor performance in comparison with the technique; we believe that the constraint relaxation, as well as the SCR-tailored abstraction, contributed to this difference. Experiments in Section 5.1 provide some evidence in this direction. The number of predicates introduced for abstraction refinement made the tool timeout in a significant number of cases. Indeed, while covered/infeasible reachability properties typically demand about 16 refinement predicates in average, those that timeout require 159 in average, confirming the motivation in the paper (the most significant difference was observed in the `sis` case study, with an average of 10 introduced predicates for those cases that finished as infeasible or covered, and an average of 329 introduced predicates for those that timeout).

Randoop and Evosuite are examples of well-regarded tools for test generation in the context of Java programs. While Randoop can produce a very large set of test cases very efficiently, the length necessary in many cases to reach some states of interest makes it very difficult for this tool to achieve good test predicate coverage. Evosuite targets coverage, so it was attempted to encode reachability of the predicates of interest as a guidance to the tool. Again, the tool could cover many cases, but it does not even meet the performance of some standard model checking tools. It seems that this particular test generation scenario is better suited for the latter kind of tools.

Finally, from Table III, it is observed that the abstraction time was significantly lower than the time spent in model checking (with the only exception of `car3prop`). This provides evidence to support the initial hypothesis, which establishes that many interesting properties can be proved without considering the specific values taken by numeric variables. However, for the properties in which the precise value of numeric variables is important, model checking has to be relied upon to generate the concrete test cases and refine the relaxed specification, a process much more expensive than the presented abstraction algorithm.

5.3. Experimental Results for Verification

Let us now concentrate in showing that REL-LPA-SCR can also be used for efficiently verifying properties of SCR specifications. Different state of the art tools, like the SCR Toolset [29] (generally using SPIN to perform the analysis) and the (infinite state) model checker ALV [12], have been used for verifying state and transition invariants for SCR specifications. However, they reported that these techniques requires a "manual" abstraction of the specification, in order to reduce the state space and be able to successfully complete the analysis of the specification. But, as these manually reduced specifications are not publicly available, the result of running the model checkers (SPIN and ALV) on the original specification are reported. CPAchecker is also included in this evaluation. Thus, the goal in this section is to evidence that REL-LPA-SCR can verify properties without any manual abstraction, as opposed to the related techniques. Since the original (large) specifications are dealt with in this section, the timeout for verifying a single property is set to 1 hour for each technique. Theorem proving based approaches are not taken into account for the comparison, because the focus is on automated verification techniques.

The state and transition invariants mentioned by Heitmeyer et al. [29], Bultan and Heitmeyer [12] and Bharadwaj et al. [9] are considered: 11 properties for the `ccs` model, 4 for the `sis` and 2 for the `autopilot`, respectively. In particular, 9 properties in the `ccs` models are valid invariants, 3 in the `sis` and 2 in the `autopilot`. Notice that `car3prop` is not included in this evaluation. This

Prepared using stvrauth.cls
DOI: 10.1002/stvr

has to do with the fact that, in the literature, no property verification scenarios were found regarding this case study. Table IV summarizes for each tool the number of properties that were verified to be valid (v), the number of generated counterexamples (c), and the number of times the tool was inconclusive (i). The time, in seconds, needed for each tool to analyzed the whole set of properties for each case study, is also reported.

The approach presented in this paper behaves considerably better than the model checkers SPIN, ALV and CPAchecker. From the point of view of efficiency, the technique is in general several orders of magnitude faster than alternative techniques. The approach is able to verify and generate counterexamples for properties in both simple and complex SCR specifications, that contain numeric variables. Notice that the only case in which this approach behaves worse than a model checker is the `sis` example, where SPIN is faster for the same reasons we explained in the previous section, when SPIN was faster for test case generation. However, SPIN fails in analyzing the bigger case studies. In the case of ALV, it shows a better behaviour than SPIN when analyzing the `ccs` case study, being able to verify 6 (simple) properties, and even to generate one counterexample (the same handled by SPIN). However, ALV's performance gets worse when analyzing specifications that contain numeric variables, as in the cases of the `autopilot` and `sis`. It is also important to remark that, from the point of view of effectiveness, the approach had in these case studies, no case reported as inconclusive.

	Props. (v/c/i)	Time
autopilot (2 properties)		
SPIN	0 / 0 / **2**	7200
ALV	0 / 0 / **2**	7200
CPAchecker	1 / 0 / **1**	3657
REL-LPA-SCR	2 / 0 / **0**	1
ccs (11 properties)		
SPIN	0 / 1 / **10**	36004
ALV	6 / 1 / **4**	14652
CPAchecker	5 / 2 / **4**	16654
REL-LPA-SCR	9 / 2 / **0**	44
sis (4 properties)		
SPIN	3/1/**0**	1
ALV	2/0/**2**	7202
CPAchecker	3 / 0 / **1**	3640
REL-LPA-SCR	3/1/**0**	5

Table IV. Comparison between the whole analysis approach (REL-LPA-SCR) and model checkers in the verification of properties for SCR specifications.

6. RELATED WORK

The SCR Toolset [29] provides a wide set of tools to perform different kinds of analysis to SCR specifications, e.g., syntax checking, consistency analysis, and the verification of properties via theorem proving and model checking [4, 12, 29]. In particular, some of these tools enable one to perform verification via model checking, which either require manual abstractions from the developer, or do not scale well. With respect to testing, the simulator in the SCR toolset allows the developer to load specific scenarios, which are in principle provided by the engineer, and check

Prepared using stvrauth.cls
DOI: 10.1002/stvr

whether certain associated assertions are violated or not in the particular executions described by the scenarios.

Gargantini and Heitmeyer [22] used a model checker for automatically obtaining executions transiting through particular *modes* of the SCR specification. Later on, Fraser and Gargantini [21] made a thorough comparison between various model checkers (symbolic, bounded, explicit state, etc.) in order to automatically generate test cases from tables, and analyzed the achieved coverage and scalability issues. With respect to test generation, the evaluation of the technique presented in this paper is based on case studies analyzed by Fraser and Gargantini, and the comparison with model checkers uses Fraser and Gargantini's framework of model checkers for test case generation from SCR tables. Bultan and Heitmeyer [12] employed the ALV infinite state model checker to analyze SCR tables. As they mentioned, ALV is not well suited for finding counterexamples, and so, neither it is for generating test cases. Then, ALV was not considering for assessing the approach presented in this paper for this task. The approach has been then compared with most of the above mentioned works, which shows that the technique is more effective than other approaches for larger requirements models.

To the best of the authors' knowledge, automated abstraction techniques have been applied to tabular requirements specifications infrequently, particularly for testing purposes. Bharadwaj and Heitmeyer [10] applied abstraction to SCR specifications, for scalability purposes related to model checking. That approach is based on the removal of irrelevant variables (slicing) to the property of interest, and the transformation of "internal" variables into monitored ones, with the aim of removing detailed variables (e.g., numerical variables). Nevertheless, they only provide experimental results using a few properties over two small SCR specifications. Later on, Heitmeyer et al. [29] assessed these abstraction techniques under a broader set of examples, mentioning the need of performing different kinds of manual ad-hoc abstractions in order to succesfully complete the analysis of some properties via model checking. In contrast to these techniques, the approach presented in this paper performs the abstraction process in a fully automated way, and is capable of dealing with the SCR specification maintaining the original level of details, as shown in section 5.3.

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 [5]. Other related and successful approaches are those by Henzinger et al. [34] and Chaki et al. [14]. The LPA-SCR algorithm of the approach presented in this paper is based on that put forward by Beyer et al. [5], which employs lazy abstraction [33] for test generation, but targets requirements specifications. As opposed to programming domains, in which abstraction was successfully applied [16], requirement specifications are not "control intensive", thus constituting a novel interesting domain. Furthermore, the LPA-SCR algorithm was especially tailored for SCR requirements specifications, exploiting their particular characteristics in order to make the analysis of interesting SCR specifications feasible.

Other automated tools such as Pex [49] and SPF [51] 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. Two

Prepared using stvrauth.cls
DOI: 10.1002/stvr

representatives of well regarded tools for automated test case generation have been considered in this paper, one that is driven by white-box criteria, namely Evosuite [20], and a black-box driven tool based on random test generation, namely Randoop [44]. The experiments showed that while these tools perform very well in achieving high coverage and mutation score in the context of test generation from code [44, 20], the setting considered in this paper seems to be inherently different, with the high degree of nondeterminism in specification that has been mentioned earlier in this paper, leading these tools to a weaker table coverage compared with model checking tools.

Finally, it is worth to mention that this paper extends and improves the work presented by Degiovanni et al. [19]. The main difference that work is the way in which specifications with large numerical domains are handled. Particularly, while Degiovanni et al. [19] proposed to split the domain of numerical variables in several smaller intervals, helping the abstraction process to converge, that solution had two major drawbacks: it is not trivial to decide how to split the intervals, and the analysis is very slow, for relatively simple specifications. The new abstraction based approach presented in this paper performs a fully automated abstraction process that, as we showed in section 5, allows us to efficiently analyze SCR specifications (maintaining the original level of detail).

7. CONCLUSIONS

The requirements process is an important phase in the development of quality software, that demands requirements specification frameworks to aid requirements elicitation, specification and refinement. Formal approaches such as SCR, the subject of this article, have an increased potential to expose ambiguities, missing cases and errors in requirements specifications, but their successful application depends greatly on having adequate and powerful automated tool support. A well known limitation in automated analysis is the so called state explosion problem, which has been tackled in various ways, including abstraction based techniques. In this paper, a number of features inherent to SCR tabular requirements specifications, that can be exploited to improve their abstraction-based automated analysis, have been identified. These observations led to a 2-stage abstraction process, that involves a first relaxation stage, where certain constraints are disregarded from the specification, and a second, more classical, counterexample guided abstraction refinement process. It is during the latter that features of SCR specifications are exploited to modularize and simplify requirements specifications for abstraction-based analysis, while the former treats a particular kind of specification element (numerical variables over large domains) to aid the latter.

The approach was assessed through a detailed comparison with alternative techniques for test generation and property verification of SCR specifications, on case studies that have been widely used in the literature. This approach showed better efficency and scalability than existing techniques, and was capable of analyzing all our case studies at their original level of detail, unlike other approaches that failed in several cases, most notably in those whose specifications involved monitored numerical variables with large domains.

The motivation for dealing with “larger” specifications is straightforward: it contributes to the scalability in this kind of analysis, and facilitates the validation and verification activities. On the one hand, for test generation it is important that the generated tests maintain the level of

Prepared using stvrauth.cls
DOI: 10.1002/stvr

abstraction/detail expected by users, and present in the implementation. Otherwise, the tests have to be manually adapted by the engineer in order to be useful in the original context; a slow, tedious an error prone process. On the other hand, properties verified in reduced models might exhibit violations in the original, larger specification. Thus, using reduced specifications is only useful to increase the engineer's confidence in the correctness of the specification, but are less effective in ensuring the absense of errors.

ACKNOWLEDGEMENTS

This material is based upon work partially supported by Argentina's ANPCyT through grants PICT 2012-1298, 2013-2624, 2015-2341 and 2015-2088, and by NPRP grant NPRP-4-1109-1-174 from the Qatar National Research Fund (a member of Qatar Foundation).

REFERENCES

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise and N. Sharygina, *Lazy Abstraction with Interpolants for Arrays*, in Proc. of LPAR 2012, LNCS 7180, Springer, 2012
2. M. Archer, C. Heitmeyer, and E. Riccobene, *Proving invariants of I/O automata with TAME*, Automated Software Engineering, 9:201232, 2002.
3. I. Alexander, N. Maiden, *Scenarios, Stories, Use Cases*, Wiley, 2004.
4. J. Atlee and J. Gannon, *State-Based Model Checking of Event-Driven System Requirements*, IEEE Trans. Software Eng. 19(1), IEEE Press, 1993.
5. D. Beyer, A. Chlipala, T. Henzinger, R. Jhala and R. Majumdar: *Generating Tests from Counterexamples*, in Proc. of ICSE 2004, IEEE, 2004.
6. D. Beyer, T. Henzinger and G. Théoduloz, *Lazy Shape Analysis*, in Proc. of CAV 2006, LNCS 4144, Springer, 2006.
7. D. Beyer, T. Henzinger, R. Jhala and R. Majumdar, *The Software Model Checker BLAST*, STTT 9(5-6), Springer, 2007.
8. D. Beyer and M. Keremoglu, *CPAchecker: A Tool for Configurable Software Verification*, in Proceedings of CAV 2011, Springer, 2011.
9. R. Bharadwaj and C. Heitmeyer, *Applying the SCR Requirements Method to a Simple Autopilot*, in Proc. of NASA Langley Formal Methods Workshop, 1997.
10. R. Bharadwaj and C. Heitmeyer, *Model Checking Complete Requirements Specifications Using Abstraction*, Automated Software Engineering 6(1), Springer, 1999.
11. 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.
12. 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.
13. R. W. Butler, *An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot*. NASA Technical Memorandum 110255. NASA Langley Research Center, May 1996.
14. 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.
15. A. Cimatti, I. Narasamdya and M. Roveri, *Boosting Lazy Abstraction for SystemC with Partial Order Reduction*, in Proc. of TACAS 2011, LNCS, Springer, 2011.
16. 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.
17. P. Courtois and D. Parnas, *Documentation for safety critical software*, in Proc. of the ICSE'93, ACM, 315-323, 1993.
18. S. Das and D. Dill, *Successive Approximation of Abstract Transition Relations*, in Proc. of the IEEE Symposium on Logic in Computer Science LICS 2001, IEEE Press, 2001.

DOI: 10.1002/stvr

19. R. Degiovanni, P. Ponzio, N. Aguirre and M. Frias, *Abstraction Based Automated Test Generation from Formal Tabular Requirements Specifications*, in Proceedings of International Conference on Tests and Proofs TAP 2011, Zurich, Switzerland, LNCS 6706, Springer, 2011.
20. G. Fraser and A. Arcuri, *EvoSuite: automatic test suite generation for object-oriented software*, in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering ESEC/FSE 2011, ACM, 2011.
21. G. Fraser and A. Gargantini, *An Evaluation of Model Checkers for Specification Based Test Case Generation*, in Proc. of ICST 2009, LNCS, Springer, 2009.
22. A. Gargantini and C. Heitmeyer, *Using Model Checking to Generate Tests from Requirements Specifications*, in Proc. of ESEC/FSE 1999, LNCS, Springer, 1999.
23. C. Ghezzi, M. Jazayeri y D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, 2002.
24. S. Graf and H. Saïdi, *Construction of abstract state graphs with pvs*, in Proc. of CAV 1997, Springer-Verlag.
25. C. Heitmeyer, *Requirements Models for Critical Systems*, Software and Systems Safety - Specification and Verification, IOS Press, 2011.
26. C. Heitmeyer and J. McLean, *Abstract requirements specifications: A new approach and its application*, IEEE TSE , 580-589, 1983.
27. C. Heitmeyer, B. Labaw, D. Kiskis, *Consistency Checking of SCR-Style Requirements Specifications*, en Proceedings de IEEE International Symposium on Requirements Engineering, York, U.K. IEEE 1995, doi: 10.1109/ISRE.1995.512546.
28. C. Heitmeyer, R. Jeffords and B. Labaw, *Automated consistency checking of requirements specifications*, Trans. on Soft. Eng. and Methodology, 5(3), ACM, 1996.
29. C. Heitmeyer, M. Archer, R. Bharadwaj and R. Jeffords, *Tools for constructing requirements specifications: the SCR Toolset at the age of ten*, Computer Systems: Science & Engineering, 20(1), 2005.
30. C. Heitmeyer, M. Pickett, E. Leonard, I. Ray, D. Aha, J. Trafton and M. Archer, *Building High Assurance Human-Centric Decision Systems*, Automated Software Engineering 22(2), Springer, 2015.
31. C. Heitmeyer and E. Leonard, *Obtaining Trust in Autonomous Systems: Tools for Formal Model Synthesis and Validation*, in Proceedings of IEEE/ACM 3rd. FME Workshop on Formal Methods in Software Engineering, IEEE CS, 2015.
32. 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.
33. T. Henzinger, R. Jhala, R. Majumdar and G. Sutre, *Lazy abstraction*, in Proc. of POPL 2002, ACM, 2002.
34. T. Henzinger, R. Jhala, R. Majumdar and K. McMillan, *Abstractions from proofs* in in Proc. of POPL 2004, LNCS, Springer, 2004.
35. G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, 1991.
36. G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*, Addison-Wesley, 2004.
37. P. Jalote, *An Integrated Approach to Software Engineering*, 3rd. Edition, Springer, 2005.
38. R. Jhala, *Lazy Abstraction*, PhD thesis, 1999.
39. J. Kirby Jr., *Example NRL/SCR software requirements for an automobile cruise control and monitoring system*, Technical Report TR-87-07, Wang Institute of Graduate Studies, 1987.
40. E. Letier, J. Kramer, J. Magee and S. Uchitel, *Deriving Event-Based Transition Systems from Goal-Oriented Requirements Models*, in Automated Software Engineering, Volume 15, Issue 2, pp 175–206, 2008.
41. N. Leveson, M. Heimdahl, H. Hildreth and J. Reese, *Requirements Specifications for Process-Control Systems*, Trans. on Software Engineering, 20(9), IEEE, 1994.
42. N. Maiden and I. Alexander, *Scenarios, stories, use cases: through the systems development life-cycle*, J. Wiley and sons, Chichester, 2004.
43. 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.
44. C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, *Feedback-Directed Random Test Generation*, in Proceedings of International Conference on Software Engineering ICSE 2007, IEEE, 2007.
45. D. Parnas and J. Madey, *Functional Documentation for Computer Systems*, Science of Computer Programming, 25(1), Elsevier, 1995.
46. B. Schlich, *Model checking of software for microcontrollers*, ACM Trans. Embedded Comput. Syst. 9(4), ACM, 2010.
47. W. Stevens, G. Myers and L. Constantine, *Structured Design*, IBM Systems Journal, 13 (2), 115-139, 1974.
48. I. Sommerville, *Software Engineering*, 8th Edition, Addison-Wesley, 2006.
49. N. Tillmann and J. Halleux, *Pex-White Box Test Generation for .NET*. In Proc. of TAP 2008, LNCS, Springer, 2008.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

50. A. J. Van Schouwen, D. Parnas, J. Madey, *Documentation of requirements for computer systems*, in Proc. of IEEE RE'93, 198-207, 1993.
51. W. Visser, C. Păsăreanu and S. Khurshid, *Test input generation with Java PathFinder*, in Proc. of ISSTA 2004, ACM, 2004.
52. Y. Vizel, O. Grumberg and S. Shoham, *Lazy abstraction and SAT-based reachability in hardware model checking*, in Proc. of FMCAD 2012, IEEE, 2012.

DOI: 10.1002/stvr