Bug Hunting with False Negatives*

Jens Calamé[1], Natalia Ioustinova[1], Jaco van de Pol[1,2], and Natalia Sidorova[2]

[1] Centrum voor Wiskunde en Informatica,
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
[2] Eindhoven University of Technology,
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
jens.calame@cwi.nl,ustin@cwi.nl,
jaco.van.de.pol@cwi.nl,n.sidorova@tue.nl

Abstract. Safe data abstractions are widely used for verification purposes. Positive verification results can be transferred from the abstract to the concrete system. When a property is violated in the abstract system, one still has to check whether a concrete violation scenario exists. However, even when the violation scenario is not reproducible in the concrete system (a false negative), it may still contain information on possible sources of bugs.
Here, we propose a bug hunting framework based on abstract violation scenarios. We first extract a violation pattern from one abstract violation scenario. The violation pattern represents multiple abstract violation scenarios, increasing the chance that a corresponding concrete violation exists. Then, we look for a concrete violation that corresponds to the violation pattern by using constraint solving techniques. Finally, we define the class of counterexamples that we can handle and argue correctness of the proposed framework.
Our method combines two formal techniques, model checking and constraint solving. Through an analysis of contracting and precise abstractions, we are able to integrate overapproximation by abstraction with concrete counterexample generation.

1 Introduction

Abstractions [5,6,7,9,13,18] are widely used to reduce the state space of complex, distributed, data-oriented and thus large systems for verification purposes. We focus on abstractions that are used to check satisfaction rather than the violation of properties. These abstractions are constructed in such a way that we can transfer positive verification results from the abstract to the concrete model, but not the negative ones. Counterexamples found on the abstract system may have no counterpart in the concrete system. We further refer to this kind of counterexamples as *false negatives*. False negatives are usually used to refine the abstraction and iteratively call the model checking algorithm on the refined abstraction [4,10,17].

* Part of this research has been funded by the Dutch BSIK/BRICKS project.

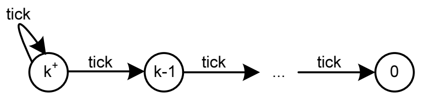

Fig. 1. Abstracted timer

In this paper, we consider false negatives in the context of *data abstractions*, i.e. abstractions that substitute actual data values by abstract ones and operations on concrete data by operations on abstract data, depending on the property being verified. We use the timer abstraction from [8] as an illustrating example in this paper. This abstraction leaves all values of a discrete timer below k unchanged and maps all higher values to the abstract value k^+. Note that the deterministic time progress operation *tick* (decreasing the values of active timers by one), becomes non-deterministic in the abstract model (see Fig. 1). But this abstraction allows us to only regard the k smallest values and the constant k^+ in order to prove that a property holds for any value n.

Consider a system, where every timer setting $set(n)$ is followed by n *tick* steps before the timer is set again, for some constant value n. Being set to a value n above k, the abstract timer can do an arbitrary number of *tick* steps, before it reaches value $k-1$. From there, it decreases until it expires at 0.

We now use this k^+ timer abstraction to verify an action-based *LTL* property $\Box(a \rightarrow \Diamond b)$ and obtain the following trace as a counterexample for the abstract system: $a.set(k^+).tick^3.b.(a.set(k^+).tick^2.d)^\star$. Note that the timer abstraction affected the parameter of the *set* action, so that the number of *tick* steps following $set(k^+)$ is not fixed anymore. This trace obviously is a *false negative* since it does not reflect any possible trace of the original system (remember the constant n).

Assuming that the trace $a.set(n).tick^n.b.(a.set(n).tick^n.d)^\star$ exists in the original system, the false negative still contains a clue for finding this concrete counterexample. We can relax the found abstract counterexample by using the information that the operations on timers are influenced by the timer abstraction and check whether the concrete system contains a trace matching the pattern $a.any^\star.b.(a.any^\star.d)^\star$ where *any* represents any action on timers. We call such a pattern a *violation pattern*. Note that any trace matching the violation pattern violates our property of interest. The pattern contains a cyclic part, and it is more restrictive than the negation of the property. Therefore, when enumerative model checking is concerned, it is easier to find a trace of the concrete system satisfying the pattern than one that violates the property.

In this paper, we propose a framework that supports the bug hunting process described in the above example. In this framework, we apply a combination of abstraction, refinement and constraint solving techniques to process algebraic specifications. The framework is illustrated in Fig. 2 where $\mathcal{M}$ denotes the concrete system, $\mathcal{M}^\alpha$ stands for an abstraction of $\mathcal{M}$, ϕ is the property in question and ϕ^α is its abstraction. When checking whether the abstract system satisfies the abstract property, we may obtain a counterexample having no counterpart in

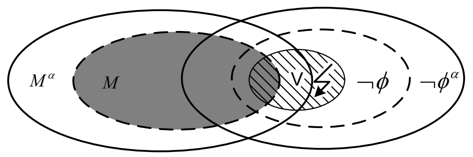

Fig. 2. Violation pattern approach

the concrete system (the set $(\mathcal{M}^\alpha \backslash \mathcal{M}) \cap \neg\phi$). Given the counterexample, we relax actions influenced by the data abstraction and construct a violation pattern that represents a set of traces violating the property and resembling the counterexample. For this to work, we need an accurate analysis of contracting and precise abstractions [16]. In short, contracting abstractions abstract a system property in a way, that less traces fulfill this property, while precise abstractions do not affect fulfilling traces.

To check whether there is a concrete trace matching the violation pattern, we transform the violation pattern and the specification of the concrete system into a constraint logic program. Subsequently, a constraint solver is used to find a concrete trace matching the violation pattern, if such a trace exists.

The rest of the paper is organized as follows: In the remainder of this section, we compare our work with related work. In Section 2, we define the class of systems we are working with. Furthermore, we define a next-free action-based *LTL* (*ALTL*) and extend it by data (*eALTL*). In Section 3, we work out abstractions of labeled transition systems and of *eALTL* properties. In Section 4, we present a taxonomy of counterexamples, of which we select the false negatives to build up a bug hunting framework and discuss its correctness in Section 5. In Section 6, we give an example for the implementation of this framework. Finally, we conclude with Section 7.

Related work

First, we compare our method with the more traditional CEGAR approach (Counter-Example-Guided Abstraction Refinement) [4,17], which has recently been extended to state- and event-based software by the ComFoRT framework [3]. In both methods, abstractions preserve properties in one direction only: if the abstract system satisfies the property, so does the concrete system; a counterexample may however be a real one or a false negative. In the CEGAR method, the abstraction is refined based on abstract counterexamples, and model checking is iteratively applied to the refined abstractions of the system. Our method is to generalize false negatives and then to find violations in the concrete specification, which are similar to the original false negative. Note that in principle both methods can be combined: given a false negative, one could search for a concrete violation using our method. If it is found, the CEGAR loop can be terminated early. If still no concrete counterexample is found, one can proceed by refining the abstraction as in the CEGAR approach and iterate verification.

For counterexamples that have been produced when model checking the abstract model, it has to be determined whether they represent real system defects. In [21], the problem of automating this analysis has been addressed. For this purpose, the authors propose two techniques: model checking on choice-free paths and abstract counterexample guided concrete simulation. In [20], an approach based on test generation is proposed for searching for concrete instances of abstract counterexamples. Only counterexamples for safety properties are addressed by those approaches, i.e. it works only for finite counterexamples, while we deal with infinite traces. Unlike these approaches, we look for a concrete trace that does not match a counterexample itself, but a violation pattern that has been generated from it.

Finally, [15] and [22] are orthogonal to ours, because there model checking methods are proposed that rely on a refinement of an *underapproximation* of the system behavior. These methods aim at the falsification of a desired property and apply a refinement when no counterexample is found. In contrast, we try to prove the property and, if we do not succeed, try to find a concrete counterexample.

2 The Specification Framework

We did our research in the setting of the process-algebraic language μCRL [14]. As graphical notation, we will use symbolic transition systems (*STS*s, cf. [24]). A specification $\mathcal{S}$ over an alphabet of actions Act (defined below), is given as the parallel composition $\Pi_{i=1}^{n} P_i$ of a finite number of processes. A process definition P is given by a four-tuple $(Var, Loc, Edg, (\ell_{\text{init}}, \eta_{\text{init}}))$, where Var denotes a finite set of variables, and Loc denotes a finite set of *locations* ℓ, or control states. A mapping of variables to values is called a valuation; we denote the set of valuations by $Val = \{\eta \mid \eta\colon Var \to \mathbb{D}\}$. We assume standard data domains such as $\mathbb{N}$ or $\mathbb{B}$. The set $Expr$ denotes the set of *expressions*, built from variables and function symbols in the usual way. An expression can be evaluated to a value, given a valuation for the variables. We write $\mathbb{D}$ when leaving the data-domain unspecified and silently assume all expressions to be well-typed. The initial location and valuation are given by $(\ell_{\text{init}}, \eta_{\text{init}})$. The set $Edg \subseteq Loc \times Act \times Loc$ denotes the set of edges. An *edge* describes changes of configurations specified by an *action* from Act.

Let $Event$ be a set of system events (cf. channel names, action names). As actions, we distinguish (1) the *input* of an event s together with a local variable to which the received value can be assigned, (2) the *output* of an event s together with a value described by an expression, and (3) *internal actions*, like assignments. Every action is *guarded* by a boolean expression g. This guard decides, whether the action may be executed (when the guard evaluates to true) or not. So we define the set Act to consist of: $g \triangleright ?s(x)$, $g \triangleright !s(e)$, or $g \triangleright \tau, x := e$, resp., and we use $\iota, \iota' \ldots$ when leaving the action unspecified. For an edge $(\ell, \iota, \hat{\ell}) \in Edg$, we write more suggestively $\ell \to_{\iota} \hat{\ell}$.

Examples of specifications can be found in Fig. 6 later in this paper. There, the system on the left-hand side awaits an input $in(x)$, with a variable x that will be instantiated at runtime. Depending on the value of x, the system will then output the event *out* with either the value of x or 0.

Before we define the semantics of our specifications, we introduce the notion of labeled transition systems and traces.

Definition 1 (Total *LTS*). *A* labeled transition system *(LTS) is a quadruple* $\mathcal{M} = (\Sigma, Lab, \Delta, \sigma_{\text{init}})$ *where* Σ *is a set of* states*,* Lab *is a set of* action labels*,* $\Delta \subseteq \Sigma \times Lab \times \Sigma$ *is a labeled* transition relation *and* $\sigma_{\text{init}} \in \Sigma$ *is the* initial state*. A* total LTS *does not contain any deadlocks.*

Further we write $\sigma \rightarrow_\lambda \sigma'$ for a triple $(\sigma, \lambda, \sigma') \in \Delta$ and refer to it as a λ-*step* of $\mathcal{M}$. For the rest of the paper, we assume *LTS*s to be *total*.

Definition 2 (Traces). *Let* $\mathcal{M} = (\Sigma, Lab, \Delta, \sigma_{\text{init}})$ *be an LTS. A* trace β *of* $\mathcal{M}$ *is a mapping* $\beta \colon \mathbb{N} \setminus \{0\} \rightarrow Lab$*, such that there is a mapping* $\beta' \colon \mathbb{N} \rightarrow \Sigma$ *and for any* $i, (i+1) \in \mathbb{N} \colon \beta'[i] \rightarrow_{\beta[i+1]} \beta'[i+1] \in \Delta$ *with* $\beta'[0] = \sigma_{\text{init}}$*. We further refer to the suffix of* β *starting at* $\beta[i]$ *as* β^i*. By* $[\![\mathcal{M}]\!]_{\text{trace}}$*, we denote the set of all traces in* $\mathcal{M}$*.*

The step semantics of $\mathcal{S}$ is given by an *LTS* $\mathcal{M} = (\Sigma, Lab, \Delta, \sigma_{\text{init}})$. Here, the set of states is $\Sigma := Loc \times Val$ with the initial state $\sigma_{\text{init}} := (\ell_{\text{init}}, \eta_{\text{init}}) \in \Sigma$. The (possibly infinite) set of labels is $Lab := \{s(d) \mid s \in Event, d \in \mathbb{D}\}$. Finally, the transitions $\Delta \subseteq \Sigma \times Lab \times \Sigma$ are given as a labeled transition relation between states. The labels differentiate internal actions and communication steps, either input or output, which are labeled by an event and a value being transmitted, i.e. τ, $?s(v)$ or $!s(v)$, respectively.

Receiving an event s with a communication parameter x, $\ell \rightarrow_{g \triangleright ?s(x)} \hat{\ell} \in Edg$, results in updating the valuation $\eta_{[x \mapsto v]}$ according to the parameter of the event and changing current location to $\hat{\ell}$. The possible input values are limited by the guard. Output, $\ell \rightarrow_{g \triangleright !s(e)} \hat{\ell} \in Edg$, is guarded, so sending a message involves evaluating the guard and the expression according to the current valuation. It leads to the change of the location of the process from ℓ to $\hat{\ell}$. Assignments, $\ell \rightarrow_{g \triangleright \tau, x:=e} \hat{\ell} \in Edg$, result in the change of a location and the update of the valuation $\eta_{[x \mapsto v]}$, where $[\![e]\!]_\eta = v$. Assignment transitions are labeled by the corresponding action label τ. Firing such a transition also involves evaluating the guard and the expression according to the current valuation.

2.1 *ALTL* with data (*eALTL*)

To specify properties of a system, we propose a data extension for action-based Linear Temporal Logic (*ALTL* [12]). This logic specifies system properties in terms of events parameterized with data. Here, we first define *action formulae*, their satisfaction and then define *extended ALTL*, *eALTL*.

Definition 3 (Action Formulae). *Let x be a variable from Var, expr be a boolean expression from Expr, a be an event from Event, then the syntax of an action formula ζ is defined as follows:*

$$\zeta ::= \top \mid \{a(x) \mid expr(x)\} \mid \neg\zeta \mid \zeta \wedge \zeta$$

We will use $a(x)$ as an abbreviation for $\{a(x) \mid \mathsf{true}\}$ and $a(d)$ as an abbreviation for $\{a(x) \mid x = d\}$. We do not impose any limitations on the set of boolean expressions.

Definition 4 (Interpretation of an action formula). *Let $act \in Lab$ and ζ be an action formula, then the satisfaction of ζ on act is defined as follows:*

$$\begin{array}{ll} act \models \top & \textit{always } (\mathsf{true}) \\ act \models \{a(x) \mid expr(x)\} & \textit{if there exists some } d \in \mathbb{D} \textit{ s.t.} \\ & act = a(d) \textit{ and } [\![expr]\!]_{[x \mapsto d]} = \mathsf{true} \\ act \models \zeta_1 \wedge \zeta_2 & \textit{if } act \models \zeta_1 \textit{ and } act \models \zeta_2 \\ act \models \neg\zeta & \textit{if not } act \models \zeta \end{array}$$

Definition 5 (*eALTL* Formulae). *Let ζ be an action formula. The syntax of* eALTL *formulae is defined by the following grammar:*

$$\phi ::= \zeta \mid \neg\phi \mid \phi \wedge \phi \mid \phi \mathbf{U} \phi$$

Definition 6 (Semantics of *eALTL*). *Let β be a (infinite) trace, ϕ, ϕ_1, ϕ_2 be* eALTL *formulae, ζ be an action formula then*

$$\begin{array}{ll} \beta \models \zeta & \textit{if } \beta[1] \models \zeta \\ \beta \models \neg\phi & \textit{if not } \beta \models \phi \\ \beta \models \phi_1 \wedge \phi_2 & \textit{if } \beta \models \phi_1 \textit{ and } \beta \models \phi_2 \\ \beta \models \phi_1 \mathbf{U} \phi_2 & \textit{if there exists } k \in \mathbb{N} \textit{ such that} \\ & \textit{for all } 0 \le i < k : \beta^i \models \phi_1 \textit{ and } \beta^k \models \phi_2 \end{array}$$

Let $\mathcal{M} = (\Sigma, Lab, \Delta, \sigma_{\text{init}})$ be an *LTS*. We say that $\mathcal{M} \models \phi$ iff $\beta \models \phi$ for all traces β of $\mathcal{M}$ starting at σ_{init}. We introduce the following shorthand notations: $\bot$ for $\neg\top$; $\Diamond\phi$ for $\top\mathbf{U}\phi$; $\Box\phi$ for $\neg\Diamond\neg\phi$; $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$; $\phi_1 \Rightarrow \phi_2$ for $\neg\phi_1 \vee \phi_2$; $\phi_1 \mathbf{R} \phi_2$ for $\neg(\neg\phi_1 \mathbf{U} \neg\phi_2)$. *eALTL* is suitable to express a broad range of property patterns like occurrence, bounded response or absence [11]. For our further work on abstracting properties of systems, we will require that property formulae are in positive normal form, i.e. all negations are pushed inside, right before action formulae.

3 Abstraction of Systems and Properties

In this section, we present an abstraction mechanism based on homomorphisms as in [5,16], and adapted to an action-based setting. Abstracting a system leads to a smaller state space which can thus be examined easier. However, model checking an abstracted system also requires the abstraction of the properties that have to be checked. We will first present the abstraction of systems and then the abstraction of *eALTL* properties.

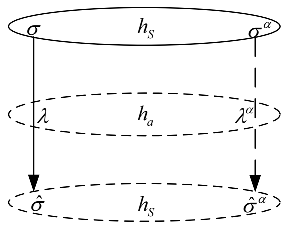

Fig. 3. Abstraction requirement for *LTS*s

3.1 Abstraction of a system

The basis for the abstraction is a homomorphism $\alpha = \langle h_s, h_a\rangle$ defining two abstraction functions which regard states and actions of an *LTS* [5,23]. The function $h_s : \Sigma \rightarrow \Sigma^\alpha$ maps the states of a concrete system $\mathcal{M}$ to abstract states. The function $h_a : Lab \rightarrow Lab^\alpha$ does the same with action labels of $\mathcal{M}$.

Definition 7. *Let abstraction* $\alpha = \langle h_s, h_a\rangle$ *for automaton* $\mathcal{M} = (\Sigma, Lab, \Delta, \sigma_{\text{init}})$ *be given. We define* $\alpha(\mathcal{M})$ *to be* $(\Sigma^\alpha, Lab^\alpha, \Delta^\alpha, h_s(\sigma_{\text{init}}))$*, where* $\sigma^\alpha \rightarrow_{\lambda^\alpha} \hat{\sigma}^\alpha \in \Delta^\alpha$ *if and only if* $\sigma \rightarrow_\lambda \hat{\sigma} \in \Delta$*, for some* σ*,* $\hat{\sigma}$ *and* λ *such that* $h_s(\sigma) = \sigma^\alpha$*,* $h_s(\hat{\sigma}) = \hat{\sigma}^\alpha$*, and* $h_a(\lambda) = \lambda^\alpha$*.*

Now, we define a *homomorphic relation on traces*, $\equiv_\alpha \subseteq Lab^\star \times Lab^{\alpha\star}$, which relates concrete traces from $Lab^\star$ to their abstract counterparts in $Lab^{\alpha\star}$.

Definition 8 (Trace Inclusion w.r.t. α). *Let* $\alpha = \langle h_s, h_a\rangle$ *be a homomorphism. For a trace* β *of* $Lab^\star$ *and trace* β^α *of* $Lab^{\alpha\star}$*, we say* $\beta \equiv_\alpha \beta^\alpha$ *iff for all* $i \in \mathbb{N}$: $\beta^\alpha[i] = h_a(\beta[i])$*.*

We say that $\mathcal{M} \subseteq_\alpha \mathcal{M}^\alpha$ *iff for every trace* β *of* $\mathcal{M}$ *there exists a trace* β^α *of* $\mathcal{M}^\alpha$ *such that* $\beta \equiv_\alpha \beta^\alpha$*.*

It is well known that homomorphic abstractions lead to overapproximations. Notably, the abstract system covers at least the traces of the concrete system:

Lemma 9. *Let* $\mathcal{M}$ *be an* LTS *with homomorphism* α*. Then* $\mathcal{M} \subseteq_\alpha \alpha(\mathcal{M})$*.*

It is often more convenient to apply abstractions directly on a system specification $\mathcal{S}$ than on its transition system $\mathcal{M}$. Such an abstraction on the level of $\mathcal{S}$ is well-developed within the *Abstract Interpretation* framework [6,7,9]. Abstract Interpretation imposes a requirement on the relation between the concrete specification $\mathcal{S}$ and its abstract interpretation $\mathcal{S}^\alpha$. This takes the form of a safety requirement on the relation between data and operations of the concrete system and their abstract counterparts (we skip the details). Each value of the concrete domain $\mathbb{D}$ is related by a data abstraction function h_d to a value from the abstract domain $\mathbb{D}^\alpha$. For every operation (function) f on the concrete data domain, an abstract function f^α is defined, which overapproximates f. For reasons of simplicity, we assume f to be a unary operation. Furthermore, we apply only data abstraction. This means that the names of actions in a system are not affected by the abstraction, i.e. $h_a(a(d)) = a(h_d(d))$ such that two actions $a(x)$ and $b(y)$ cannot be mapped to the same abstract action.

However, applying abstractions directly on a system's specification $\mathcal{S}$ rather than on its *LTS* leads to a loss of precision. Let $\mathcal{S}^\alpha$ be the abstract interpretation of $\mathcal{S}$, and let $\mathcal{M}^\alpha$ and $\mathcal{M}$ be their underlying *LTS*s. It is well known that $\mathcal{M}^\alpha$ is only an overapproximation of $\alpha(\mathcal{M})$ (cf. [5]). In particular, we will still have trace inclusion up to α: $\mathcal{M} \subseteq_\alpha \alpha(\mathcal{M}) \subseteq_\alpha \mathcal{M}^\alpha$.

3.2 Abstraction of *eALTL* formulae

The abstraction of *eALTL* formulae is based on the notions of *contracting and precise abstractions* as it has been introduced in [16]. In a contracting abstraction, a property ϕ^α holds for a trace β^α iff the property ϕ holds for *all* concrete traces β with $\beta^\alpha = \alpha(\beta)$. Note that for soundness of abstract model checking, we need contracting abstractions. This does, however, not imply that all properties that hold for the original system, *must* also hold in the abstract system (see Fig. 4, ellipse vs. hatched square). In *precise* abstractions, this cannot happen.

Definition 10 (Contracting and Precise Abstraction). *Let ϕ be a property over an action alphabet λ. Its abstraction ϕ^α is*

- contracting *iff:* $\forall \beta \in Lab^\star : \alpha(\beta) \models \phi^\alpha \Rightarrow \beta \models \phi$.
- precise *iff:* $\forall \beta \in Lab^\star : \alpha(\beta) \models \phi^\alpha \Leftrightarrow \beta \models \phi$.

In the following, we define an abstraction of *eALTL* formulae that is guaranteed to be contracting. We assume all formulae to be in positive normal form.

Definition 11 (Abstraction of Action Formulae). *Action formulae as defined in Def. 3 are abstracted as follows:*

$$\begin{aligned}
\alpha(\top) &:= \top \\
\alpha(\{a(x) \mid expr(x)\}) &:= \{a(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \to expr(x))\} \\
\alpha(\neg\{a(x) \mid expr(x)\}) &:= \bigvee_{b \neq a} \{b(x^\alpha)\} \vee \{a(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \to \neg expr(x)\} \\
\alpha(\zeta_1 \wedge \zeta_2) &:= \alpha(\zeta_1) \wedge \alpha(\zeta_2)
\end{aligned}$$

The abstraction of *eALTL* formulae is more straightforward, since we do not have to regard negations on this level.

Definition 12 (Abstraction of *eALTL* Formulae). eALTL *formulae as defined in Def. 5 are abstracted as follows:*

$$\begin{aligned}
\alpha(\phi_1 \wedge \phi_2) &:= \alpha(\phi_1) \wedge \alpha(\phi_2) \\
\alpha(\phi_1 \mathbf{U} \phi_2) &:= \alpha(\phi_1) \mathbf{U} \alpha(\phi_2)
\end{aligned}$$

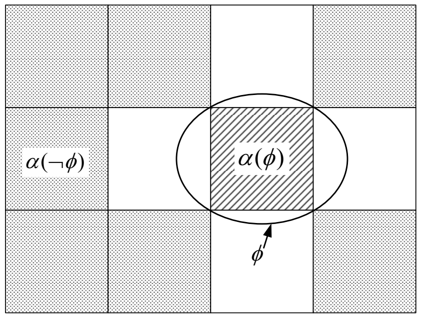

Fig. 4. Contracting Abstraction

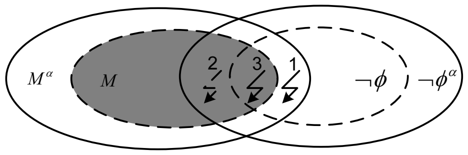

Fig. 5. Classification of counterexamples

In order to have precise abstractions, we need a restriction on the homomorphism α. We define that α is *consistent* with ϕ, iff for all action formulae ζ occuring in ϕ, $\{h_a(act)|act \models \zeta\} \cap [\![\neg\alpha(\zeta)]\!] = \emptyset$, i.e. the hatched square and the ellipse in Figure 4 coincide.

Lemma 13. *If α is consistent with ϕ, then $\alpha(\phi)$ is precise.*

4 Classification of Counterexamples

We can now explain model checking by abstraction for *eALTL* formulae. Let a specification $\mathcal{S}$ (with underlying *LTS* $\mathcal{M}$) and an *eALTL* property ϕ be given. Let us investigate whether a contracting abstraction α suffices for our needs. We compute $\alpha(\phi)$ and $\mathcal{S}^\alpha$, generate its underlying *LTS* $\mathcal{M}^\alpha$ and use a model checking algorithm to check $\mathcal{M}^\alpha \models \phi^\alpha$. If this holds, we can derive by our previous results, that also $\mathcal{M} \models \phi$, without ever generating $\mathcal{M}$. If it does not hold, we obtain a counterexample. Here we provide a classification of abstract counterexamples and demonstrate their relationship with contracting and precise abstractions of *eALTL* formulae.

Given a concrete system $\mathcal{M}$, its abstraction $\mathcal{M}^\alpha$, a property ϕ and its abstraction ϕ^α, we differentiate between three classes of abstract counterexamples (see Fig. 5). Given a counterexample χ^α, we refer to a concrete trace $\chi \in [\![\mathcal{M}]\!]_{\text{trace}}$ such that $\chi \equiv_\alpha \chi^\alpha$ as a *concrete counterpart* of χ^α. The first class (see counterexample 1 in Fig. 5) consists of the counterexamples having *no* concrete coun-

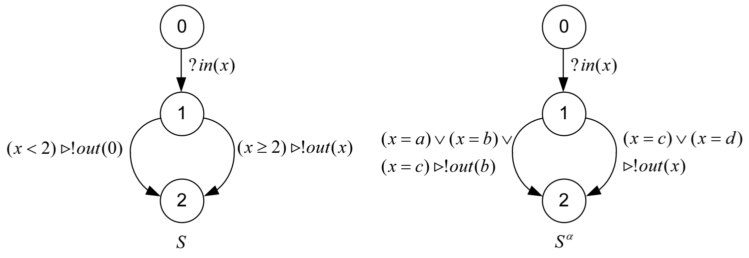

Fig. 6. Concrete and Abstracted Specifications from Example 15

terparts in the concrete system. These counterexamples are referred to as *false negatives.*

The second class (see counterexample 2 in Fig. 5) consists of counterexamples having (at least one) concrete counterpart *satisfying* the original property. We further refer to this class as *spurious counterexamples.*

The third class (see counterexample 3 in Fig. 5) consists of the counterexamples having at least one counterpart in the concrete system; moreover all concrete counterparts violate the concrete property. Counterexamples from this class are referred to as *ideal counterexamples.*

Definition 14. *Let χ^α be a counterexample obtained by verifying an abstraction ϕ^α of a property ϕ on the abstraction $\mathcal{M}^\alpha$ of a system $\mathcal{M}$ w.r.t. the homomorphism h. We distinguish the following three cases:*

1. *We call χ^α a* false negative*, if there is no $\chi \in [\![\mathcal{M}]\!]_{\text{trace}}$ such that $\chi \equiv_\alpha \chi^\alpha$.*
2. *We call χ^α a* spurious *counterexample if there exists $\chi \in [\![\mathcal{M}]\!]_{\text{trace}}$ such that $\chi \equiv_\alpha \chi^\alpha$ and $\chi \models \phi$.*
3. *Otherwise, we call χ^α an* ideal *counterexample.*

Contracting abstractions may lead to spurious counterexamples. The following example illustrates this case.

Example 15. Let $\mathcal{S}$ in Fig. 6 be the specification of a concrete system. We abstract $\mathbb{Z}$ into $\mathbb{Z}^\alpha = \{a, b, c, d\}$ where a stands for the numbers from $(-\infty, -3)$; b stands for the numbers from $[-3, 0]$; c stands for the numbers from $(0, 3]$; and d stands for the numbers from $(3, +\infty)$. By applying this abstraction to $\mathcal{S}$ we obtain $\mathcal{S}^\alpha$ (see Fig. 6).

Consider the property $\phi = \Diamond(\{out(x) \mid (x \geq 2)\})$. We compute the contracting abstraction of ϕ as follows:

$$\begin{aligned}
\phi &= \Diamond(\{out(x) \mid (x \geq 2)\}) \\
\phi^\alpha &= \Diamond(\{out(x^\alpha) \mid \forall x : h_d(x) = x^\alpha \rightarrow (x \geq 2)\}) \\
&= \Diamond(out(d))
\end{aligned}$$

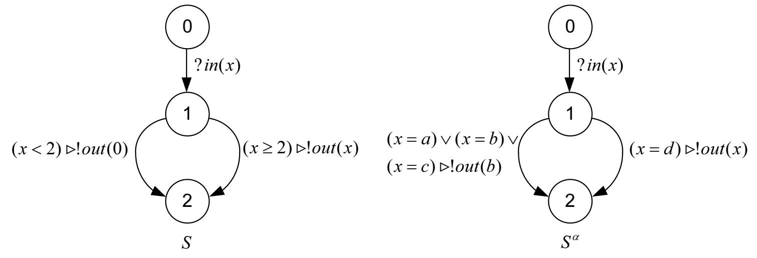

Fig. 7. Concrete and Abstracted Specifications from Example 16

Verifying ϕ^α on $\mathcal{S}^\alpha$ we may obtain the trace $in(c).out(c)$ as a counterexample, because it is a trace in S^α, but does not satisfy ϕ. However, the concrete trace $in(2).out(2)$ corresponding to the abstract counterexample satisfies $\Diamond(out(x) \wedge (x \geq 2))$. Hence, $\neg\phi^\alpha$ is not precise enough.

Such spurious counterexamples are problematic for tracking real bugs. Therefore, we will use *precise* abstractions, in order to avoid spurious counterexamples. A contracting abstraction can be made precise, by fitting the abstraction to the predicates in the specification and the formula:

Example 16. Let $\mathcal{S}$ in Fig. 7 be the specification of a concrete system. We abstract $\mathbb{Z}$ into $\mathbb{Z}^\alpha = \{a, b, c, d\}$ where the interpretation of a and b remains the same as in Example 15 while c represents the numbers from the interval $(0, 2)$ and d represents those from $[2, +\infty)$. By applying this abstraction to $\mathcal{S}$ we obtain $\mathcal{S}^\alpha$ (see Fig. 7).

Consider again the property $\phi = \Diamond(\{out(x) \mid (x \geq 2)\})$ and its abstraction $\phi^\alpha = \Diamond(out(d))$. Verifying ϕ^α on $\mathcal{S}^\alpha$ we may obtain the following counterexamples: $in(a).out(b)$, $in(b).out(b)$, and $in(c).out(b)$. In this example it is straightforward to see that any concretization of these traces is a counterexample for ϕ. So in this case, the abstraction is precise.

5 Bug Hunting with False Negatives

Counterexamples that are false negatives still have a value for detecting bugs in specifications. By relaxing them, i.e. making them even more abstract, false negatives cover a larger part of the system, which can contain bugs. In this manner, they can serve as a starting point for bug hunting.

In this section, we provide an overview of our framework for bug hunting with false negatives. This process comprises the following steps:

1. Specify a requirement as a formula ϕ of *eALTL*.
2. Choose and apply a data abstraction, which is consistent with ϕ, to the specification of the concrete system and to the concrete property.

3. Abstract counterexamples for the property are (automatically) determined using model checking.
4. Generalize the false negative further by *relaxing* actions, which are not directly relevant for our search. This results in a violation pattern. The relaxing process itself is automatic, only the counterexample and the set of directly relevant actions have to be given as input to the algorithm (see Alg. 1).
5. The concrete counterexamples are automatically computed by finding the intersection of the original system and the violation pattern.

Since the first three steps of the framework can be handled by existing data abstraction and model checking techniques, our contribution concerns the steps 4 and 5 of the framework.

5.1 Constructing a violation pattern

A counterexample that we obtain in case the property is violated on our abstract model is an infinite trace of the form $\beta_p \beta_s^\omega$ where β_p is a finite prefix and β_s^ω is a cyclic suffix with a finite *cycle base* β_s.

Although the counterexample χ^α may have no counterpart in the concrete system, it can contain a clue about a counterexample present in the concrete system. Therefore we transform a counterexample χ^α into a *violation pattern* $\mathcal{V}$, considering only *infinite* counterexamples.

A violation pattern is an *LTS* that accepts all traces hitting a distinguished *cyclic* state infinitely often. The violation pattern accepts only traces which are *similar* to the counterexample and *violate* the abstract property. The actions mentioned in the property are essential for the property violation. Therefore we keep this information in the violation pattern. For actions influenced by the abstraction, the order and the number of actions in a similar trace may differ from those of the counterexample. We will now first illustrate the idea of similarity on a simple example and then generalize it.

Example 17. Let us come back to the example from the introduction. Assume that we model-check the property $\Box(a \rightarrow \Diamond b)$ and obtain the abstract counterexample $a.set(k^+).tick^3.b.(a.set(k^+).tick^2.d)^\omega$ (see Fig. 8). The k^+ is in this case an abstraction of a timer: The original value of the timer is preserved up to k; any value above k is abstracted to the constant value k^+. To guarantee that the property is violated by any trace accepted by the pattern, we keep *at least* the actions a and b, because they are mentioned in the property (see Fig. 9). Since we are searching for similar traces with an infinite *cyclic* suffix β_s, we may also decide to keep information about some actions of this cycle. Here we also provide the action step d in the cycle (see Fig. 9). The actions *tick* and *set*(k^+) are not mentioned in the property and are definitely influenced by the timer abstraction. Therefore, we *relax* these actions, meaning, we allow these actions to occur an arbitrary number of times in an arbitrary order (see states 1 and 3 of the violation pattern in Fig. 9).

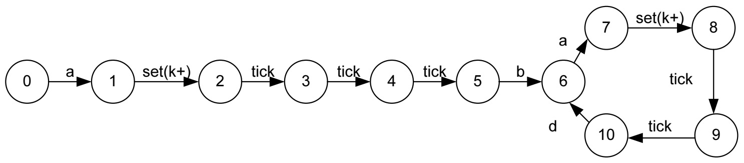

Fig. 8. A concrete counterexample

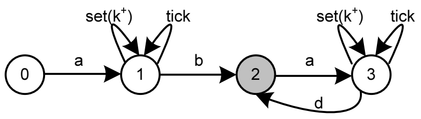

Fig. 9. The violation pattern for the counterexample

We refer to the set of action labels that we do not want to relax by Lab_{keep}. This set includes *at least* all the labels mentioned in the abstract (and also the concrete) property. In the violation pattern, we distinguish a *cyclic* state which corresponds to the first state in the cyclic suffix. The last action in the cycle base of an infinite counterexample leads to this cyclic state.

Ideally, we would like to relax more actions influenced by data abstraction. These actions can be found by applying static analysis techniques. The more actions we keep, the more concrete the counterexample is and the faster we can check whether there is a concrete trace matching the pattern. By keeping too many actions, however, we might end up with a violation pattern that specifies traces having no counterparts in the concrete system.

Definition 18 (Non-relaxed Actions). *Given a set Act^{ϕ^α} of actions appearing in a property ϕ^α, we define that some set Lab_{keep} of non-relaxed actions in a violation pattern is* consistent *if and only if $Lab_{\text{keep}} \supseteq Act^{\phi^\alpha}$.*

Lab_{keep} can optionally contain additional actions, like the last action of a cyclic suffix, or actions not influenced by the data abstraction, to make the violation pattern more specific.

Definition 19 (Violation Pattern). *Given an abstract counterexample $\chi^\alpha = \beta_p\beta_s^\omega$ and a set Lab_{keep} of non-relaxed actions, a* violation pattern *is an extended* LTS $\mathcal{V} = (\Sigma, Lab, \Delta, \sigma_{\text{init}}, \sigma_{\text{cyclic}})$ *constructed by Algorithm 1, where σ_{cyclic} is the cyclic state.*

The set of traces visiting the cyclic state infinitely often, is further referred to as the set $[\![\mathcal{V}]\!]_{\text{trace}}$ of accepted traces.

Given a counterexample $\chi^\alpha = \beta_p\beta_s^\omega$ and a set Lab_{keep} of actions to keep, Algorithm 1 constructs the violation pattern $\mathcal{V}$. The algorithm starts with creating the initial state $\sigma_{\text{init}} := 0$ of $\mathcal{V}$ and goes through $\beta_p\beta_s$. When the algorithm

Algorithm 1 Build Violation Pattern

```
Require: $\chi^\alpha = \beta_p\beta_s^\omega, Lab_{\text{keep}}$                    // trace, actions to keep
Ensure: $\mathcal{V} = (\Sigma, Lab, \Delta, \sigma_{\text{init}}, \sigma_{\text{cyclic}})$          // violation pattern
 1: $\sigma_{\text{init}} := 0;\ \Sigma := \{\sigma_{\text{init}}\};$                         // initialization
 2: $st := 0;$                                                  // current state st of $\mathcal{V}$
 3: for all $i = 1..|\beta_p\beta_s|$ do                            // for all steps of $\beta_p\beta_s$
 4:     if $\chi^\alpha[i] \notin Lab_{\text{keep}}$ then
 5:         $\Delta := \Delta \cup \{(st, \chi^\alpha[i], st)\};$                   // add a relaxed step
 6:     fi
 7:     if $i = |\beta_p| + 1$ then
 8:         $\sigma_{\text{cyclic}} := \{st\};$                  // indicate the first state of the cycle
 9:     fi
10:     if $\chi^\alpha[i] \in Lab_{\text{keep}} \vee i = |\beta_p\beta_s|$ then       // if step to be kept or last one
11:         if $i = |\beta_p\beta_s|$ then                            // if last state in cycle base
12:             $st' := \sigma_{\text{cyclic}};$                           // next state is the cyclic one
13:         else
14:             $st' := st + 1;$                                     // next state is arbitrary
15:         fi
16:         $\Sigma := \Sigma \cup \{st'\};$                                   // add a new state,
17:         $\Delta := \Delta \cup \{(st, \chi^\alpha[i], st')\};$                // add the step to the new state
18:         $st := st';$                                 // proceed with the next state of $\mathcal{V}$
19:     fi
20: od
```

encounters an action to relax, it adds a self-loop transition labeled with this action to the current state of $\mathcal{V}$. When it encounters an action to keep, it adds a transition from the current state to the (new) next state labeled by this action or, if the algorithm has reached the end of the cycle base, back to the cyclic state. The first state of β_s is assigned to σ_{cyclic}.

Lemma 20. *Let Lab_{keep} be consistent with ϕ^α, let χ^α be a counterexample for ϕ^α, and $\mathcal{V}$ be a violation pattern generated from χ^α and Lab_{keep}. Every trace $\beta^\alpha \in [\![\mathcal{V}]\!]_{\text{trace}}$ satisfies: $\beta^\alpha \not\models \phi^\alpha$.*

Proof Sketch *If a counterexample χ^α is relaxed, at least all actions from the property's alphabet are in Lab_{keep} (see Def. 18), i.e. they are not relaxed. This means, that if a trace β^α is in $[\![\mathcal{V}]\!]_{\text{trace}}$, it contains all actions from ϕ^α in the same order as they appear in the counterexample χ^α.*

Since we are considering next-free properties, the absolute position of the actions in question in the particular trace is not required, to keep violating the property ϕ^α. Preserving the cyclic state σ_{cyclic} for an infinite counterexample also allows us to preserve the cyclicity of the infinite suffix of such a counterexample.

5.2 Looking for a concrete counterexample

After we have constructed the violation pattern $\mathcal{V}$, we check whether there is a concrete counterexample $\chi = \chi_p\chi_s^\omega$, such that the corresponding abstract counterexample $\chi^\alpha \in [\![\mathcal{V}]\!]_{\text{trace}}$.

$$\text{ROUTPUT}\;\frac{\ell \rightarrow_{g \triangleright !s(e)} \hat{\ell} \in \mathit{Edg}}{s(\mathit{state}(\ell, \overline{\mathit{Var}}), \mathit{state}(\hat{\ell}, \overline{\mathit{Var}}), \mathit{param}(e)) \leftarrow g}$$

$$\text{RINPUT}\;\frac{\ell \rightarrow_{g \triangleright ?s(x)} \hat{\ell} \in \mathit{Edg}}{s(\mathit{state}(\ell, \overline{\mathit{Var}}), \mathit{state}(\hat{\ell}, \overline{\mathit{Var}}_{[x \mapsto Y]}), \mathit{param}(Y)) \leftarrow g}$$

$$\text{RASSIGN}\;\frac{\ell \rightarrow_{g \triangleright \tau, x:=e} \hat{\ell} \in \mathit{Edg}}{\tau(\mathit{state}(\ell, \overline{\mathit{Var}}), \mathit{state}(\hat{\ell}, \overline{\mathit{Var}}_{[x \mapsto e]}), \mathit{param}) \leftarrow g}$$

Table 1. From specification $\mathcal{S}$ to rule system $\mathcal{R}_{\mathcal{S}}$

For infinite counterexamples we need to check that some state of χ_s corresponds to σ_{cyclic}. We employ constraint solving [19] to find a concrete counterexample, which allows us to check this condition for infinite (but cyclic) traces, and also for certain infinite and parameterized systems.

To find a concrete trace matching the violation pattern $\mathcal{V}$, we transform the specification of the concrete system and the violation pattern into a *constraint program* and formulate a *query* to find such a trace. This transformation is similar to the one described in [2]. Note that for a concrete system with an infinite state space, it is possible that the constraint solver will not terminate. Moreover, it is possible that the only traces that match the violation pattern are spiral traces, not cyclic ones (i.e. we do have a loop with respect to control locations, but some variable is infinitely growing) and we will not be able to find them.

The transformation of the specification of the concrete system into a rule system $\mathcal{R}_{\mathcal{S}}$ is defined in Table 1. Each edge of the specification $\mathcal{S}$ is mapped into a rule $\varrho \leftarrow g$. In the rule, g is a guard and ϱ is a user-defined constraint of the form $s(\mathit{state}(\ell, \overline{\mathit{Var}}), \mathit{state}(\hat{\ell}, \overline{\mathit{Var}}'), \mathit{param}(Y))$. The first parameter *state* of the user-defined constraint describes the source states corresponding to the edge in terms of control locations of a process and valuations of process variables. The second parameter *state* describes the destination states in terms of control locations of a process and valuations of process variables. The third parameter *param* contains parameters representing input and output values. The constraint is satisfied iff the guard g is satisfied. This means, that there is a transition $(\ell, \eta) \rightarrow_{g \triangleright s(d)} (\hat{\ell}, \hat{\eta})$, if and if only the rule $s(\mathit{state}(\ell, \overline{\mathit{Var}}), \mathit{state}(\hat{\ell}, \overline{\mathit{Var}}'), \mathit{param}(Y)) \leftarrow g$ holds, for some substitution $\overline{\mathit{Var}} = \eta$, $\overline{\mathit{Var}}' = \hat{\eta}$, $Y = d$ that makes guard g become true.

In ROUTPUT, the name of the constraint coincides with the event s. Note that the values of the process variables $\overline{\mathit{Var}}$ remain unmodified and the output value is represented by the parameter Y whose value is given by the expression e. In RINPUT, the input leads to the substitution of the value of process variable x by the value of the input parameter Y. In RASSIGN, an assignment is represented by substituting the value of the process variable x by the valuation of expression e. These rules have no local parameters, so the parameter structure is empty.

Transformation of the edges of $\mathcal{V} = (\Sigma, \mathit{Lab}, \Delta, \sigma_{\text{init}}, \sigma_{\text{cyclic}})$ into the rules of the rule system $\mathcal{R}_{\mathcal{V}}$ is defined in Table 2. Here, we abbreviate $(\ell, \overline{\mathit{Var}})$ by $\vec{X}$

$$(1)\ \frac{\sigma \rightarrow_{!s(v)} \hat{\sigma} \vee \sigma \rightarrow_{?s(v)} \hat{\sigma} \vee \sigma \rightarrow_{\tau} \hat{\sigma} \qquad \sigma \neq \sigma_{\text{cyclic}}}{\begin{array}{c}\sigma(\mathit{state}(\vec{X}), \bar{C}, \bar{\beta}) \leftarrow s(\mathit{state}(\vec{X}), \mathit{state}(\vec{X}'), \mathit{param}(Y)) \wedge \\ v = \alpha(Y) \wedge \hat{\sigma}(\mathit{state}(\vec{X}'), \bar{C}, [\bar{\beta}, s(Y)])\end{array}}$$

$$(2)\ \frac{\sigma \rightarrow_{!s(v)} \hat{\sigma} \vee \sigma \rightarrow_{?s(v)} \hat{\sigma} \vee \sigma \rightarrow_{\tau} \hat{\sigma} \qquad \sigma = \sigma_{\text{cyclic}}}{\begin{array}{c}\sigma(\mathit{state}(\vec{X}), \bar{C}, \bar{\beta}) \leftarrow s(\mathit{state}(\vec{X}), \mathit{state}(\vec{X}'), \mathit{param}(Y)) \wedge \\ v = \alpha(Y) \wedge \big(\vec{X} \in \bar{C} \vee \hat{\sigma}(\mathit{state}(\vec{X}'), [\vec{X} \mid \bar{C}], [\bar{\beta}, s(Y)])\big)\end{array}}$$

Table 2. From violation pattern $\mathcal{V}$ to rule system $\mathcal{R}_{\mathcal{V}}$

and $(\hat{\ell}, \overline{\mathit{Var}}')$ by $\vec{X}'$. Intuitively, given a step of $\mathcal{V}$, a rule of $\mathcal{R}_{\mathcal{V}}$ checks whether the concrete system may make this step. The rules also take into account the information about the cyclic state and the data abstraction.

The rules in Table 2 transform the steps of a violation pattern into rules of the form: $\varrho \leftarrow \xi \wedge g_\alpha \wedge \nu$. ϱ is a user-defined constraint of the form $\sigma(\mathit{state}(\vec{X}), \bar{C}, \bar{\beta})$ specifying the source state $\mathit{state}(\vec{X})$ of the concrete system, the set of states, which are *possibly* on a cycle, in the set $\bar{C}$. This set is accumulatively constructed, and it contains concrete candidate cyclic states that match with σ_{cyclic} in the violation pattern. The third parameter, $\bar{\beta}$, contains the trace that has already been visited while examining $\mathcal{V}$ and will contain the end result.

ξ is a user-defined constraint of the form $s(\mathit{state}(\vec{X}), \mathit{state}(\vec{X}'), \mathit{param}(Y))$ as defined above. It represents a step on which the concrete system and the violation pattern can potentially synchronize.

The guard g_α checks whether the data parameters of the concrete action are a concretization of the data parameters of the abstract action.

Finally, ν determines whether and how the violation pattern has to be examined further. We will explain this in more detail shortly. Simplified, ν stops the further examination of $\mathcal{V}$, if we have reached the cyclic state of $\mathcal{V}$. Otherwise, it decides that the next step in $\mathcal{V}$ will be taken and sets the parameters accordingly.

We will now describe the rules in more detail. Rule 1 of Table 2 transforms steps of the violation pattern whose actual state σ is not the beginning of the cycle base. The step specified by the constraint $s(\mathit{state}(\vec{X}), \mathit{state}(\vec{X}'), \mathit{param}(Y))$ changes the state to $\hat{\sigma}$ in the violation pattern and to $\mathit{state}(\vec{X}')$ in the concrete system. That is captured by the constraint $\hat{\sigma}(\mathit{state}(\vec{X}'), \bar{C}, [\bar{\beta}, s(Y)])$ in ϱ. The constraint is satisfied only if both the violation pattern and the concrete system can make the specified step and the action labeling the step of the concrete system satisfies the constraint $v = \alpha(Y)$. When doing the next examination step, $\bar{C}$ is left unchanged, while the actual event s together with a concretization Y of its parameter v, is added to the examination trace $\bar{\beta}$.

Rule 2 transforms those steps of the violation pattern, which start from a state corresponding to the beginning of the cycle. If the actual corresponding state in the system is found in $\bar{C}$, the state is cyclic and has already earlier been visited during the examination. In this case, examination ends successfully. If the state is not yet in $\bar{C}$, it is *potentially cyclic*. In this case, the step is treated like in

Rule 1, just that the actual state of the system is added to $\bar{C}$. Logging potentially cyclic states and examining the violation pattern further allows us to not only detect obvious cycles, i.e. cycles in the system which are also immediately visible in the violation pattern. We can also detect those cycles, where the system spirals before entering a real cycle. In this case, the system first runs through a cycle with respect to the location, but differing in the data part of the system state, before finally returning to a previously visited state. In such a case, the cyclic state of the violation pattern is visited more than once.

The rule system $\mathcal{R}_{\mathcal{V}}$, together with the rule system $\mathcal{R}_{\mathcal{S}}$, forms the constraint program. In order to check whether we can find a concrete counterexample matching the violation pattern, we transform the pair of the initial state of the violation pattern and the initial state of the concrete system into the query $q_{\text{init}} := \sigma_{\text{init}}(\mathit{state}(\vec{X}_{\text{init}}), [], [])$ (initial state without any potentially cyclic states and without action steps yet in the counterexample trace) and ask a constraint solver, whether it finds a solution in the constraint program formed by $\mathcal{R}_{\mathcal{S}}$ and $\mathcal{R}_{\mathcal{V}}$. If yes, it provides us a counterexample as a list of actions and violation pattern states, which has been collected over the examination of $\mathcal{V}$. If constraint solving does not find a solution, we cannot give a conclusive answer and have to use e.g. abstraction refinement techniques to find out, whether the property holds on the concrete system or not.

Lemma 21. *If the query q_{init} to the rule system $\mathcal{R}_{\mathcal{V}}$ holds for some trace β, then $\beta \in [\![\mathcal{M}]\!]_{\text{trace}}$, and $\alpha(\beta) \in \mathcal{V}$.*

Proof Sketch *Assume, that q_{init} holds in rule system $\mathcal{R}_{\mathcal{V}}$ for a trace β. Then, this trace is in $[\![\mathcal{M}]\!]_{\text{trace}}$, since the conditions for the execution of particular actions in $\mathcal{R}_{\mathcal{V}}$ are based on $\mathcal{R}_{\mathcal{S}}$, an exact specification of the operational semantics of the specification language as defined in Section 2.*

The abstraction of the trace β, $\alpha(\beta)$, however, is in the violation pattern $\mathcal{V}$. The reason therefore is, that the rule system $\mathcal{R}_{\mathcal{V}}$ is generated from this violation pattern and thus only reflects steps (and inductively: traces), which appear in $\mathcal{V}$. Thus, the rule system only holds for those traces β, where $\alpha(\beta) \in \mathcal{V}$.

5.3 Correctness of the Framework

In this section, we argue the correctness of the framework, which has been worked out in the previous two subsections on the derivation of a violation pattern and the search for further counterexamples using constraint solving.

Theorem 22. *Let $\alpha = \langle h_s, h_a \rangle$ be an abstraction consistent with* eALTL*-property ϕ. Let* LTS*s $\mathcal{M}$ and $\mathcal{M}^\alpha$ be given, such that $\mathcal{M} \subseteq_\alpha \mathcal{M}^\alpha$. Furthermore, assume that the counterexample $\chi^\alpha \in [\![\mathcal{M}^\alpha]\!]_{\text{trace}}$ and $\chi^\alpha \not\models \phi^\alpha$. Let $\mathcal{V}$ be a violation pattern built from χ^α and a consistent Lab_{keep} by the algorithm in Fig. 1. Let β be a trace for which q_{init} holds, according to the constraint solving procedure defined in Subsection 5.2. Then β is a counterexample: $\beta \in [\![\mathcal{M}]\!]_{\text{trace}}$ and $\beta \not\models \phi$.*

Proof Sketch *By Lemma 21, $\beta \in [\![\mathcal{M}]\!]_{\text{trace}}$ and $\alpha(\beta) \in [\![\mathcal{V}]\!]_{\text{trace}}$. By Lemma 20, $\alpha(\beta) \not\models \phi^\alpha$. By Lemma 13, as α is a precise abstraction, we have $\beta \not\models \phi$.*

6 Implementation

To check the applicability of our framework we performed a number of verification experiments with μCRL specifications [14]. For constraint solving, we used Eclipse Prolog [1].

We took a mutant of the *Positive Acknowledgment Retransmission Protocol* (*PAR*) [25] as our case study. The usual scenario for *PAR* includes a sender, a receiver, a message channel and an acknowledgment channel. The sender receives a frame from the upper layer, i.e. from its environment, sends it to the receiver via the message channel, the receiver delivers the frame to the upper layer and sends a positive acknowledgment via the acknowledgment channel to the sender. *PAR* depends on timers, which we have chosen too low for our experiments.

We tried to verify that for any setting of the sender timer exceeding some value k, all messages sent by the upper layer to the sender are eventually received by the upper layer from the receiver. To prove that the property holds for any setting of the sender timer exceeding k, we applied the timer abstraction described in Section 1 to the sender timer. The property was not satisfied on the abstract system (since the k we took was less than the sum of the channel delays) and we obtained a counterexample.

The abstract counterexample was not reproducible on the concrete system, since the number of *tick* steps from a setting of the sender timer till its expiration varied along the trace due to the use of the abstraction. We transformed the counterexample into a violation pattern by relaxing the actions on the sender timer as influenced by the abstraction. The specification of the system was transformed from μCRL into a set of Prolog constraint rules, while the violation pattern was immediately formulated as a set of Prolog rules according to our theory (Def. 18, 19 and Fig. 9). The constraint solver was then able to find a concrete counterexample for our property.

7 Conclusion

We proposed a novel framework for interpreting negative verification results obtained with the help of data abstractions. Existing approaches to handling abstract counterexamples try to find an exact counterpart of the counterexample (e.g. [21]). When no concrete counterpart can be found, data abstraction is considered to be not fine enough and abstraction refinement is applied (e.g. [4]).

In our framework we look for useful information in false negatives, combining the two formal methods model checking and constraint solving. Given a specification of a system and a property (formulated as an *eALTL* formula), we first choose and apply data abstraction to both of them and then verify the abstract property on the abstract system. If the verification results in a violation of the abstract property and the obtained counterexample has no counterpart in the concrete system, we transform the counterexample into a violation pattern, which is further used to guide the search for concrete counterexamples.

The framework allows to handle counterexamples obtained when verifying safety properties, but also counterexamples for liveness properties. Moreover, the framework can be applied for searching concrete counterexamples in parameterized and infinite state systems. Success is not always guaranteed – the violation pattern can be too strict, concrete counterexamples can have a spiral form (i.e. a loop in the specification, that does not lead back to a state fully identical to its starting state), or there could be no counterexample at all since the property just holds on the concrete system. Still, our approach can help in finding counterexamples in those cases when a data abstraction influences the order and the number of some actions, e.g. as timer and counter abstractions do. Even though, we defined the framework for homomorphistic abstractions in this paper, it seems to be possible to generalize abstraction and refinement on the basis of Galois-connections and so define a framework for bughunting with false negatives based on abstract interpretation.

The approach to the generation of a violation pattern leaves a certain freedom in the sense that the set of actions to relax can be more/less restrictive. Tuning the violation pattern or using the expertise of system developers to pick an appropriate set of actions to relax can be potentially less costly than repeating the abstraction/refinement cycle immediately. More case studies comparing both approaches and trying their combinations are still needed.

References

1. P. Brisset et al. *ECLIPSe Constraint Library Manual*, version 5.9 edition, May 2006. `http://eclipse.crosscoreop.com/eclipse/doc/libman.pdf`.
2. J. R. Calamé, N. Ioustinova, and J. C. v. d. Pol. Towards Automatic Generation of Parameterized Test Cases from Abstractions. Technical Report SEN-E0602, Centrum voor Wiskunde en Informatica, March 2006. To appear in ENTCS.
3. S. Chaki, E. Clarke, O. Grumberg, J. Ouaknine, N. Sharygina, T. Touili, and H. Veith. State/Event Software Verification for Branching-Time Specifications. In *Proc. of the 5th Intl. Conf. on Integrated Formal Methods (IFM'05)*, pages 53–69. Springer, 2005.
4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journ. of the ACM*, 50(5):752–794, 2003.
5. E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994. A preliminary version appeared in the Proc. of the POPL'92.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of programming languages (POPL'77)*, pages 238–252, New York, NY, USA, 1977. ACM Press.
7. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD dissertation, Eindhoven University of Technology, July 1996.
8. D. Dams and R. Gerth. The Bounded Retransmission Protocol Revisited. *Electronic Notes in Theoretical Computer Science*, 9:26, 1999.
9. D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):253–291, 1997.

10. S. Das and D. L. Dill. Counter-Example Based Predicate Discovery in Predicate Abstraction. In *FMCAD*, pages 19–32, 2002.
11. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proc. of the 21st Intl. Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press, 1999.
12. D. Giannakopoulou. *Model Checking for Concurrent Software Architectures.* PhD thesis, Imperial College of Science Techn. and Med., Univ. of London, March 1999.
13. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proc. of the 9th Intl. Conf. on Computer-Aided Verification*, pages 72–83, 1997.
14. J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In A. Ponse, C. Verhoef, and S. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62, Berlin, 1994. Springer.
15. O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided Underapproximation-Widening for Multi-Process Systems. In *Proc. of the Ann. Symp. on Principles of Programming Languages*, pages 122–131, 2005.
16. Y. Kesten and A. Pnueli. Control and Data Abstraction: The Cornerstones of Practical Formal Verification. *Intl. Journ. on Software Tools for Technology Transfer*, 2(4):328–342, 2000.
17. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental Verification by Abstraction. In *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 98–112, 2001.
18. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
19. K. Marriott and P. J. Stuckey. *Programming with Constraints – An Introduction.* MIT Press, Cambridge, 1998.
20. G. Pace, N. Halbwachs, and P. Raymond. Counter-example Generation in Symbolic Abstract Model-Checking. *Intl. Journ. on Software Tools for Technology Transfer*, 5(2):158–164, 2004.
21. C. S. Pasareanu, M. B. Dwyer, and W. Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 284–298, 2001.
22. C. S. Pasareanu, R. Pelánek, and W. Visser. Concrete Model Checking with Abstract Matching and Refinement. In *Proc. of the Intl. Conf. on Computer-Aided Verification*, pages 52–66, 2005.
23. J. C. v. d. Pol and M. A. Valero Espada. Modal Abstractions in μCRL. In *Proc. of the Algebraic Methodology and Software Technology (AMAST'04)*, 2004.
24. V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In *Proc. of the 2nd Intl. Conf. on Integrated Formal Methods (IFM'00)*, pages 338–357. Springer, 2000.
25. A. S. Tanenbaum. *Computer Networks.* Prentice Hall International, Inc., 1981.