On Module-Based Abstraction and Repair of Behavioral Programs

Guy Katz

Dept. of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel
guy.katz@weizmann.ac.il

Abstract. The number of states a program has tends to grow exponentially in the size of the code. This phenomenon, known as *state explosion*, hinders the verification and repair of large programs. A key technique for coping with state explosion is using *abstractions*, where one substitutes a program's state graph with smaller over-approximations thereof. We show how module-based abstraction-refinement strategies can be applied to the verification of programs written in the recently proposed framework of *Behavioral Programming*. Further, we demonstrate how — by using a sought-after repair as a means of refining existing abstractions — these techniques can improve the scalability of existing program repair algorithms. Our findings are supported by a proof-of-concept tool.

Keywords: Abstraction-refinement, program repair, behavioral programming.

1 Introduction

Explicit model-checking algorithms operate by spanning a program's state graph and comparing it to a given specification. This method becomes infeasible for large systems, as the state graphs tend to grow exponentially in the size of the program (the *state explosion* problem). Abstraction techniques [11] are among the most important methods for coping with state explosion and increasing the scalability of model-checking algorithms.

The key idea underlying abstraction techniques is to replace the concrete system model (i.e., the program's state graph) with a smaller abstraction thereof. Typically, the abstraction constitutes an *over-approximation* — it includes the behaviors of the concrete system, and may also include other behaviors. In the case of model-checking, proving that a given property holds for the abstract model implies that it holds for the concrete model as well. Since the abstract model is more succinct, the state explosion problem is hopefully mitigated.

We study the application of abstraction techniques to the recently proposed programming framework of *Behavioral Programming* (*BP*) [17]. In BP, programs consist of *behavioral threads* — threads of code that run in parallel, each designed to affect a specific behavior of the system. In the first part of our work, we present a formulation of BP's semantics that supports the notion of *modules*, which

are logically related threads grouped together, and discuss abstracting these modules. We then demonstrate how the composition of module abstractions yields an over-approximation of the entire behavioral program.

In the second part of our work we discuss model-checking abstract behavioral programs, and propose a *counterexample guided abstraction refinement* (*CEGAR*) [10] scheme for BP. When model-checking over-approximations, counterexamples found by the model-checker may prove *spurious*, i.e. nonexistent in the concrete system. In CEGAR, one validates each counterexample against the concrete system and, if it is spurious, refines the abstract model in a way that eliminates it. The process is then repeated iteratively until the property is proven or a genuine counterexample is found. Based on our module-based abstraction of behavioral programs, we propose a two layer abstraction-refinement scheme, similar to that of [9], in which spurious counterexamples of the composed system are used to refine module abstractions. In our setting, module interdependencies make it impossible to resolve spurious counterexamples by examining modules individually; our algorithm compensates by considering these interdependencies and refining multiple modules simultaneously when needed.

In the third part of the paper, we combine our abstraction techniques with a program repair algorithm. In [15] we demonstrated how safety violations can be eliminated from behavioral programs by adding separate, non-intrusive behavioral threads to the program. Since that repair technique included spanning the program's concrete state graph, it was susceptible to the state explosion problem. Here, we modify the technique to work on abstract state graphs instead of concrete ones, without affecting the algorithm's correctness and soundness. We observe that a given abstraction might not allow finding a correct repair even if one exists, in which case we use the desired repair as a means for refining the abstraction further. We believe that similar repair-driven refinement techniques may also be applicable to other frameworks, besides BP.

The rest of this paper is organized as follows. We define behavioral programming and its semantics in Section 2, followed by a discussion on abstracting behavioral programs in Section 3. We then discuss applying CEGAR to BP in Section 4, and suggest an abstraction-based repair algorithm in Section 5. Our experimental results appear in Section 6. Discussion of related and future work appears in Section 7.

2 Behavioral Programming

2.1 Overview

Behavioral Programming (BP) is a programming approach that extends and generalizes scenario-based programming. It was introduced with the language of Live Sequence Charts (*LSC*s) [12, 16], and is now implemented also in a variety of programming languages, such as Java, C^{++}, Erlang and others; see [17] and references therein.

A behavioral program consists of independent threads of behavior that are interwoven at run time. Each *behavior thread* (abbr. *b-thread*) repeatedly per-

forms local computations, and then synchronizes with its counterparts. At every synchronization point, each b-thread declares sets of events to be considered for triggering (*requested events*) and events whose triggering it forbids (*blocked events*). The thread then pauses until the synchronization point is resolved.

Events that have been requested by at least one b-thread and blocked by none are termed *enabled*. In each synchronization point, an *event selection mechanism* triggers one of these events and notifies all b-threads, allowing them to resume. B-threads may react to triggered events that they did not request, in which case they are said to be *waiting for* these events. The model disallows inter b-thread communication except through the synchronization mechanism.

The motivation behind BP is that it facilitates incremental non-intrusive development, as demonstrated in the example of Fig. 1, borrowed from [15]. This trait also plays a role in our repair algorithm in Section 5.

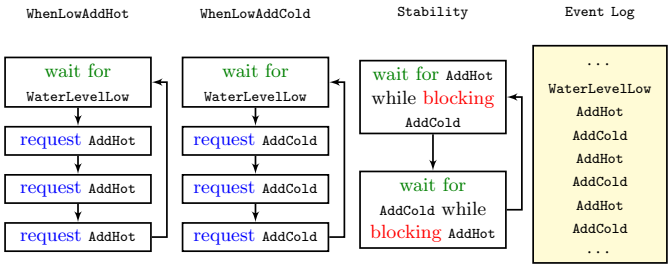

Fig. 1. (From [15]) An example of the incremental development of a system for controlling water level in a tank with hot and cold water sources. At first, b-thread `WhenLowAddHot` is created; it repeatedly waits for `WaterLevelLow` events and requests three times the event `AddHot`. It is then discovered that adding just three water quantities for every sensor reading is insufficient, and b-thread `WhenLowAddCold` is added. It performs a similar action to that of `WhenLowAddHot`, but with event `AddCold`. Then, when `WhenLowAddHot` and `WhenLowAddCold` are executed simultaneously, the run may include three consecutive `AddHot` events followed by three `AddCold` events. A new requirement is thus introduced, to the effect that water temperature should be kept stable. We add the b-thread `Stability` to enforce the interleaving of `AddHot` and `AddCold` events.

2.2 Semantics

Since b-threads communicate strictly through the synchronization mechanism, a thread is considered "at state" only when at a synchronization point. Thus, local actions performed between synchronization points can be modeled and verified locally for each thread, and are omitted from the BP model.

We formally define a b-thread BT over event set Σ and atomic proposition set AP by a tuple $BT = \langle Q, \delta, q_0, R, B, L \rangle$, where Q is a set of states (one for each synchronization point), q_0 is the initial state, $R : Q \to 2^{\Sigma}$ and $B : Q \to 2^{\Sigma}$ map states to events requested and blocked at these states (respectively), $L : Q \to 2^{AP}$ is a labeling function, and $\delta : Q \times \Sigma \to 2^{Q}$ is a transition function. We stipulate that for every $q \in Q$, $R(q) \cap B(q) = \emptyset$. Further, we require that for every state $q \in Q$, if $e \in \Sigma - B(q)$ then $\delta(q, e) \neq \emptyset$; i.e., there is a transition for every event that is not blocked in state q, though it may be a self loop. If $|\delta(q, e)| \leq 1$ for every q and e, we say that BT is *deterministic*.

The construction of a program from b-threads is performed using the *composition* and *finalization* operators. The parallel composition of threads $BT^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1, L^1 \rangle$ and $BT^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2, L^2 \rangle$, both over the same Σ and AP, yields the b-thread defined by

$$BT^1 \parallel BT^2 = \langle Q^1 \times Q^2, \delta, \langle q_0^1, q_0^2 \rangle, (R^1 \cup R^2) - (B^1 \cup B^2), B^1 \cup B^2, L^1 \cup L^2 \rangle$$

where $\langle \tilde{q}^1, \tilde{q}^2 \rangle \in \delta(\langle q^1, q^2 \rangle, e)$ if and only if $\tilde{q}^1 \in \delta^1(q^1, e)$ and $\tilde{q}^2 \in \delta^2(q^2, e)$. The union and subtraction of labeling functions are defined in the natural way, i.e. $e \in ((R^1 \cup R^2) - (B^1 \cup B^2))(\langle q^1, q^2 \rangle)$ if and only if $e \in R^1(q^1) \cup R^2(q^2)$ and $e \notin B^1(q^1) \cup B^2(q^2)$. Observe that if an event is blocked in one thread and requested in the other, it becomes blocked in the composed thread, in consistence with the fact that a blocked event cannot be triggered even if requested. It is straightforward to verify that the requested and blocked events in every state remain disjoint, and that in every state there exists a transition for every non-blocked event. Hence, $BT^1 \parallel BT^2$ is a valid b-thread.

A composition of b-threads is also termed a *module*, which is of course a b-thread in its own. Intuitively, a module is a set of threads that have yet to be plugged into a specific behavioral program, and so it still contains the relevant request and block data. Only once all the modules in a program are composed with each other can this data be discarded, through the *finalization* operator.

The finalization operator, denoted $[\cdot]$, transforms a b-thread into a labeled transition system (LTS) over Σ and AP. Formally, $[\langle Q, \delta, q_0, R, B, L \rangle] = \langle Q, \delta', q_0, L \rangle$ where Q, q_0 and L remain the same, and the transition function $\delta' : Q \times \Sigma \to 2^Q$ is given by

$$\tilde{q} \in \delta'(q, e) \iff \tilde{q} \in \delta(q, e) \bigwedge e \in R(q)$$

Observe that R and B are omitted, as they are already taken into consideration through the definition of δ'. The output of the finalization operator thus represents a general (as opposed to a behavioral) program.

Formally, we define the *behavioral program* P, comprised of b-threads $BT^1, BT^2, \ldots, BT^n$ to be the LTS defined by $P = [BT^1 \parallel \ldots \parallel BT^n]$. An execution of P is an execution of this LTS: it starts from q_0, and in each state $q \in Q$ an event is chosen for triggering if such an event exists (i.e., an event $e \in \Sigma$ for which $\delta(q, e) \neq \emptyset$). Then, the execution moves to state $\tilde{q} \in \delta(q, e)$, and so on. An execution can thus be formally recorded as a (possibly infinite) sequence of states and triggered events, $\epsilon = q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \cdots$. The matching set of events, without states, is called a *run*. The set of all runs of the program is denoted by $\mathfrak{L}(P)$. Each execution ϵ of the system defines a *trace* $\mathrm{Tr}(\epsilon) = L(q_0)L(q_1)\ldots$, which is the sequence of sets of atomic propositions associated with the states visited along the execution. The traces of the system are defined as the traces of its executions, i.e. $\mathrm{Tr}(P) = \{\mathrm{Tr}(\epsilon) \mid \epsilon \text{ is an execution of } P\}$.

The above semantics for BP differ from those used previously (e.g., in [15]), as they offer better support of the notion of modules. An equivalence between these two versions is established in Appendix I of the supplementary material [2].

The BP semantics can be extended to better describe open systems. One variant is obtained by marking some of the threads and events as controlled by the environment ("external"), as is done in [15]. Another is to use concurrent game structures and alternating-time temporal logic [3]. While our techniques can be adapted to these extensions, we leave the details for future work.

3 Abstractions for Behavioral Programming

Given behavioral programs P and $\overline{P}$, we say that $\overline{P}$ is an over-approximation of P if and only if $\mathrm{Tr}(P) \subseteq \mathrm{Tr}(\overline{P})$. Thus, for any LTL formula Φ over AP, $\mathrm{Tr}(\overline{P}) \vDash \Phi$ implies $\mathrm{Tr}(P) \vDash \Phi$, and so verifying that $\mathrm{Tr}(\overline{P}) \vDash \Phi$ shows that the original program is correct (for an introduction to LTL see, e.g., [6]). In this section we focus on constructing a suitable program $\overline{P}$ that is smaller than P, so that checking whether $\mathrm{Tr}(\overline{P}) \vDash \Phi$ is easier than checking whether $\mathrm{Tr}(P) \vDash \Phi$.

3.1 Abstracting a Behavioral Thread

We begin by defining abstractions of b-threads. Let $BT = \langle Q, \delta, q_0, R, B, L\rangle$ be a thread over events Σ and propositions AP, and let π be a AP-preserving partition of Q, i.e., $q_1 \equiv_\pi q_2 \implies L(q_1) = L(q_2)$. Let $\eta_\pi : Q \to Q/\pi$, termed the abstraction function induced by π, be a function that maps each state to its equivalence class under π. η_π gives rise to a b-thread $\overline{BT} = \langle \overline{Q}, \overline{\delta}, \overline{q_0}, \overline{R}, \overline{B}, \overline{L}\rangle$, called the abstraction thread of BT induced by π, defined in the following manner. The states of $\overline{BT}$ are the equivalence classes $\overline{Q} = Q/\pi$, and its initial state is $\overline{q_0} = \eta_\pi(q_0)$. For every state $\overline{q} \in \overline{Q}$, the mapping functions are given by $\overline{R}(\overline{q}) = \bigcup_{q\in\eta_\pi^{-1}(\overline{q})} R(q)$, $\overline{B}(\overline{q}) = \bigcap_{q\in\eta_\pi^{-1}(\overline{q})} B(q)$ and $\overline{L}(\overline{q}) = L(q)$ for (every) $q \in \eta_\pi^{-1}(\overline{q})$. The transitions relation $\overline{\delta}$ is derived from δ by:

$$\frac{q \xrightarrow{e} \tilde{q}}{\eta_\pi(q) \xrightarrow{e} \eta_\pi(\tilde{q})}$$

Note that for every $\overline{q}$, $\overline{R}(\overline{q}) \cap \overline{B}(\overline{q}) = \emptyset$, and that $\overline{q}$ has a transition for every $e \notin \overline{B}(\overline{q})$. Hence, $\overline{BT}$ is a valid b-thread. The definition is designed to make $\overline{BT}$ *more permissive* than BT — that is, to ensure that replacing BT with $\overline{BT}$ within a given program results in an over-approximation of that program. In particular, the abstraction preserves atomic proposition of states, and abstract states request at least as much and block no more than their matching concrete states. Formally, we present the following Lemma, proven in Appendix II of the supplementary material [2]:

Lemma 1. *Let $P = [BT^1 \parallel \ldots \parallel BT^n]$ be a behavioral program. Let π be an AP-preserving partition of the states of BT^1, and let $\overline{BT^1}$ be the abstraction of BT^1 induced by π. Finally, let $\overline{P} = [\overline{BT^1} \parallel BT^2 \parallel \ldots \parallel BT^n]$. Then* $\mathrm{Tr}(P) \subseteq \mathrm{Tr}(\overline{P})$.

By definition, a thread's abstraction is determined by the AP-preserving partition π in use. Clearly, an abstraction of a minimal number of states is

achieved when π is the AP-partition itself, i.e. $q_1 \equiv_\pi q_2 \iff L(q_1) = L(q_2)$. As our goal is to minimize the number of states of the composed program, this partition is of special interest. We refer to this abstraction as the coarsest abstraction of BT, and denote it by $\widehat{BT}$.

3.2 Abstracting a Behavioral Program

Due to BP's composite nature — where sets of composed threads are threads themselves — thread abstraction can be applied at various points throughout the composition process. In choosing when to apply it, our goal is to end up with an over-approximation that is neither too concrete (to mitigate state explosion), nor too abstract (so that it is meaningful). In our experiments, the best results were achieved by first grouping threads that are logically related and composing them into modules. Intuitively, this entails clustering threads that assign similar atomic propositions to their states into the same module. Each module is then abstracted individually, effectively ignoring threads that deal with other atomic propositions. Finally the abstractions are composed, generating the desired over-approximation. In this section we provide motivation for this approach, and propose an automated way for grouping together logically related threads.

To illustrate the benefits of using modules, we first discuss two of the more natural alternatives. One approach is to apply abstraction at the last step of the composition process: i.e., to compute $BT = BT^1 \parallel \ldots \parallel BT^n$ and then set $\overline{P} = [\widehat{BT}]$. While this method produces meaningful abstractions, it entails calculating the very large b-thread BT, which has at least as many states as P. Hence, this technique suffers from the state explosion problem that we have been trying to avoid. Another natural approach is to abstract each of the basic threads, i.e. calculate $\overline{P} = [\widehat{BT^1} \parallel \ldots \parallel \widehat{BT^n}]$. While this method does indeed circumvent the state explosion problem, our experiments show that the abstractions it tends to produce are too coarse to be of any practical use. Specifically, behavioral programming promotes writing threads that are small and specific, and tend to contain a single atomic proposition. Thus, early abstraction usually collapses the threads into a couple of states each, abstracting away most implementation details. Later, during verification tasks, multiple rounds of refinement are needed until a meaningful model is obtained.

The module based method can be seen as a middle ground between these two extreme alternatives. On one hand, as abstraction is applied during the early phases of the composition process, the state explosion problem is averted. On the other hand, as it is applied to threads that are sufficiently complex, the resulting over approximation is more likely to be meaningful.

The rationale behind grouping together logically related threads, as opposed to just using an arbitrary partitioning of the threads, is the desire to generate small modules: logically related threads tend to share atomic propositions, and request and block similar events. Consequently, the resulting abstractions tend to contain fewer states, and the approximation labeling functions $\overline{R}$ and $\overline{B}$ tend to be tighter, reducing the number of edges in the final over-approximation.

We conclude this section by discussing an automated method for grouping together logically related threads. As the above discussion suggests, such threads tend to share atomic propositions and requested/blocked events, and indeed this is how we attempt to group them. Let BT be a thread with states $q_1, \ldots, q_m$, and let $ap \in AP$. We define the correlation between BT and ap as:

$$cor(BT, ap) = \frac{|\{i \mid ap \in L(q_i)\}|}{m}$$

A thread's correlation to an atomic proposition is thus the fraction of states to which the labeling function assigns the proposition. Intuitively, threads that have high correlation to the same atomic proposition may be logically related. Setting a threshold M, say 0.5, induces a partitioning of the threads into modules, denoted $\equiv_M$. At first each thread is considered to reside in a separate module, and then pairs of modules are iteratively joined by the rule:

$$cor(BT^i, ap) \geq M \bigwedge cor(BT^j, ap) \geq M \implies BT^i \equiv_M BT^j$$

Analogous correlation can be defined between threads and events, by considering the fraction of states in which a thread requires or blocks the event. These correlations are easy to compute using static analysis of the threads, and are supported by the BPC framework.

Further information that can be taken into account when looking for related threads includes various string distance metrics applied to their respective names and locations in the directory structure — as programmers tend to group similar threads together and give them similar names. These measures are also straightforward to compute using automated methods. Finally, any or all of the above measures can be combined into a single metric, yielding the desired partition into logically related modules.

We summarize the resulting module-based abstraction algorithm:

Algorithm 1 Module-Based Abstraction

1: Partition the threads into modules $BT^{M_1}, \ldots, BT^{M_m}$
2: For each module BT^{M_i}, calculate $\widehat{BT^{M_i}}$
3: **return** $\overline{P} = [\widehat{BT^{M_1}} \parallel \ldots \parallel \widehat{BT^{M_m}}]$

By iteratively applying Lemma 1, we get the following corollary:

Corollary 1. *Let $BT^1, \ldots, BT^n$ be threads over event set Σ and atomic propositions AP. Let $P = [BT^1 \parallel \ldots \parallel BT^n]$, and let $\overline{P}$ be the program returned by algorithm 1. Then* $\mathrm{Tr}(P) \subseteq \mathrm{Tr}(\overline{P})$.

4 Counterexample Guided Abstraction-Refinement

Given a behavioral program P and an LTL property Φ, we attempt to prove that $P \vDash \Phi$ by calculating an over-approximation $\overline{P}$ and proving that $\overline{P} \vDash \Phi$. However, it may be the case that $P \vDash \Phi$ but $\overline{P} \nvDash \Phi$, because $\overline{P}$ is too abstract

(see an illustration in Fig. 2). Model checking $\overline{P}$ then results in a *spurious* counterexample, i.e. one that exists in $\overline{P}$ but not in P. A standard technique for handling this problem, known as *counterexample guided abstraction refinement* (*CEGAR*) [10], uses such spurious counterexamples in order to refine $\overline{P}$ in a way that eliminates them. The process is repeated until a genuine counterexample is found, or until the property is shown to hold.

In this section, we describe an implementation of CEGAR in the context of BP. The two main phases of the technique — determining whether a counterexample is spurious or genuine and refining the abstraction in order to eliminate spurious executions — are discussed in Sections 4.1 and 4.2, respectively.

For simplicity, we limit the discussion to safety properties, for which counterexamples are finite executions. The method can be extended to liveness properties and the associated loop counterexamples through *loop unwinding*; see [10].

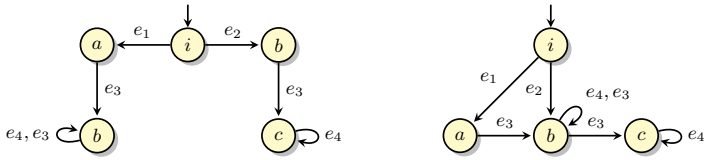

Fig. 2. A concrete state graph (on the left), and a matching abstraction (on the right). The atomic proposition labeling appears inside the states. The two states with identical labeling (b) are abstracted into a single state. The abstract state graph contains fewer states, but it also allows spurious executions. While some properties, such as $G(a \rightarrow X\neg a)$, hold for both graphs, the property $G(a \rightarrow G\neg c)$ holds in the concrete case but not in the abstract one, because of the spurious execution fragment $i \stackrel{e_1}{\rightarrow} a \stackrel{e_3}{\rightarrow} b \stackrel{e_3}{\rightarrow} c$.

4.1 Determining if an execution is spurious

Suppose that on checking whether $\overline{P} \models \Phi$, the model-checker replies in the negative, providing a finite counterexample $\overline{\epsilon}$. We wish to determine whether $\overline{\epsilon}$ is a valid execution of the original system. The idea, based on [10], is to simulate $\overline{\epsilon}$ on the concrete program in order to check if it constitutes a genuine execution. During this simulation, we must take into account the two layer structure of our abstraction scheme, as well as the role of *requested* and *blocked* events, in determining whether runs are valid.

Let $\overline{P} = [\widehat{BT^{M_1}} \parallel \ldots \parallel \widehat{BT^{M_m}}]$ be an abstract program, composed of m abstract modules, and let $\overline{\epsilon} = \overline{q_0} \stackrel{e_1}{\rightarrow} \overline{q_1} \stackrel{e_2}{\rightarrow} \ldots \stackrel{e_n}{\rightarrow} \overline{q_n}$ be a finite execution of $\overline{P}$. It is tempting to say that $\overline{\epsilon}$ is a valid execution of the concrete system if and only if its projections onto the modules form valid executions of the modules; indeed, a similar technique is used in [9]. However, in our context, this approach does not suffice. Consider, for instance, the case where the transition labeled e_1 in $\overline{\epsilon}$ exists in each of the concrete modules, but that none of them requests event e_1. In this case, looking at each module separately, we would have no way of knowing whether event e_1 is indeed enabled on the program level. Thus, our scheme must take into account the mutual effect modules have on each other.

We begin with some notation. For a set of states S, we denote by $R(S, e)$ the subset of states of S in which event e is requested. We use $\mathrm{Post}(S, e)$ to

denote the set of successors of states in S when event e is triggered. Finally, let $\overline{q} = \langle \overline{q^1}, \overline{q^2}, \ldots, \overline{q^m} \rangle$ denote an abstract state, and let η_j denote the abstraction function of module BT^{M_j}. We use η to denote the global abstraction function, i.e. $\eta(\langle q^1, \ldots, q^m \rangle) = \langle \eta_1(q^1), \ldots, \eta_m(q^m) \rangle$. This function and its inverse function are not stored explicitly, as doing so for every state in $\overline{P}$ would entail enumerating all states of P — negating the advantages offered by our two layered approach. Instead, η is only computed locally for specific states, on demand, by invoking the module abstraction functions.

Our technique follows the idea of [10], and defines a series of sets $\{S_i\}$, representing the concrete states the system can actually reach in each step of $\bar{\epsilon}$. These sets are computed by using the concrete module state graphs. The definition of S_i is given by $S_0 = \{\langle q_0^1, q_0^2, \ldots, q_0^m \rangle\}$ for the concrete initial states and $S_i = \text{Post}(R(S_{i-1}, e_i), e_i) \cap \eta^{-1}(\overline{q_i})$ for $1 \leq i \leq n$.

The idea behind this definition is to walk on the abstract graph according to the execution, and for each abstract state identify the concrete states that are truly reachable along this specific execution, using the S_i sets. As we later prove, a run is genuine if and only if it corresponds to a series of non-empty sets. Each set is derived from its predecessor by looking only at states in which the next event is requested, and calculating their successor states. Out of these successors we only keep those that are abstracted to the next state of the abstract execution, as expressed by intersecting with $\eta^{-1}(\overline{q_i})$.

The actual algorithm for checking whether an execution is spurious is thus:

Algorithm 2 Check If Spurious

```
1: for i := 0 to n do
2:     Calculate S_i; if it is empty, return True
3: return False
```

The algorithm's correctness is established via Lemma 2, proven in Appendix III of the supplementary material [2]:

Lemma 2. *Let $\bar{\epsilon}$ be an execution of $\overline{P}$. Then $\bar{\epsilon}$ is spurious, i.e. is not a valid execution of P, if and only if algorithm 2 returns* True.

Observe that computing the S_i sets is performed using the concrete state graphs of the modules, and does not entail constructing the explicit state graph of P. Every state $q \in S_i$ is stored as the set of module states to which it corresponds. The sets $R(q, e)$ and $Post(R(q, e), e)$ can be computed locally from these states. Further, there is no need to actually compute $\eta^{-1}(\overline{q_i})$, which is costly; instead, for every $q \in \text{Post}(R(S_{i-1}, e_i), e_i)$, we check whether $\eta(q) = \overline{q_i}$ by applying the module abstraction functions to its components, which is substantially cheaper.

4.2 Refining in order to eliminate a spurious execution

We now discuss refining $\overline{P}$ in order to eliminate a spurious counterexample, thus allowing another round of model-checking. The iteration on which algorithm 2 halted indicates where the refinement should occur. Indeed, this is where the

abstract and concrete graphs diverge, and so splitting the previous abstract state into multiple states could render the spurious execution invalid.

Suppose that the *Check If Spurious* algorithm stopped because $S_{i+1} = \emptyset$. This indicates a problem with transition $\overline{q_i} \overset{e_{i+1}}{\rightarrow} \overline{q_{i+1}}$ of the execution: either the concrete system can only reach states that are not mapped to abstract state $\overline{q_{i+1}}$, or event e_{i+1} is not even enabled in the concrete program — although it is enabled in the abstract one. Each case is characterized and handled differently:

Case 1. For all concrete states in S_i, transitions labeled e_{i+1} do not lead to abstract state $\overline{q_{i+1}}$, i.e. $\text{Post}(S_i, e_{i+1}) \cap \eta^{-1}(\overline{q_{i+1}}) = \emptyset$. In this case, we split $\overline{q_i}$ into 2 abstract states: state $\overline{q_i'}$ that corresponds to the concrete states S_i, and state $\overline{q_i''}$ that corresponds to the remaining states, $\eta^{-1}(\overline{q_i}) - S_i$. By definition, execution $\bar{\epsilon}$ would visit abstract state $\overline{q_i'}$ instead of $\overline{q_i}$, from which there would be no transitions to $\overline{q_{i+1}}$. Thus, $\bar{\epsilon}$ would no longer be a valid execution of the abstract program. This case corresponds to the technique used in [10].

Case 2. There exists a state $q \in S_i$ such that $\text{Post}(q, e_{i+1}) \in \eta^{-1}(\overline{q_{i+1}})$. However, $e_{i+1} \notin R(q)$; if that were not so, we would get $S_{i+1} \neq \emptyset$. In this case, state q is *waiting* for event e_{i+1} without requesting it. The request for e_{i+1} is made by a different state in $\eta^{-1}(\overline{q_i})$. As both states are mapped into the same abstract state, the outcome is the edge $\overline{q_i} \overset{e_{i+1}}{\rightarrow} \overline{q_{i+1}}$.

In this case, performing refinement as in Case 1 might not suffice, as the state requesting event e_{i+1} might also be in S_i. We thus resort to two rounds of refinement: first, we split state $\overline{q_i}$ into $\overline{q_i'}$ and $\overline{q_i''}$, as before. Then, we further refine state $\overline{q_i'}$, in order to separate states requesting event e_{i+1} from those that do not. Formally, we split $\overline{q_i'}$ into state $\overline{q_i^R}$ corresponding to concrete states $q \in S_i$ such that $e_{i+1} \in R(q)$, and state $\overline{q_i^{NR}}$ corresponding to concrete states $q \in S_i$ such that $e_{i+1} \notin R(q)$. By definition, execution $\bar{\epsilon}$ would visit abstract state $\overline{q_i^{NR}}$ instead of $\overline{q_i}$, from which there would be no transitions to $\overline{q_{i+1}}$, making it an invalid execution of the abstract program.

The following Lemma immediately follows from the above discussion:

Lemma 3. *Let $\bar{\epsilon}$ be a spurious execution of $\overline{P}$, and let $\overline{P'}$ be the refined program obtained by the above refinement step. Then $\bar{\epsilon}$ is not a valid execution of $\overline{P'}$.*

Observe that the iterative verification process entails explicitly computing $\eta^{-1}(\overline{q})$ once per each refinement step. While this step is expensive, hopefully the number of iterations is small. Reducing the number of iterations is part of our motivation for using logically related modules — see discussion in Section 3.2.

We note that the resulting refinement is defined in terms of a global abstract state that should be split into smaller states. However, as η is not stored explicitly, this refinement cannot be applied directly. Constrained by our two layered setting, we may only perform refinements on the module abstraction functions $\eta_1, \ldots, \eta_m$, indirectly refining η. Thus, a set of refinements for the $\eta_1, \ldots, \eta_m$ functions needs to be derived from the desired η refinement. This can be performed by separating (within the modules) any pair of concrete states that do

not always appear simultaneously in the new global abstract states. However, as not every refinement of η can be expressed as refinements of $\eta_1, \ldots, \eta_m$, the resulting global refinement may be finer (i.e., produce more states) than the desired one.

5 Repair using Abstractions

In this section, we propose a way of dealing with violated safety properties, using a program repair algorithm. For completeness, we begin with a brief review of the work in [15], which the present section extends.

Software maintenance is a difficult and error prone task. As bugs are discovered and requirements are added or changed, developers must modify existing code. This is tedious work; and as programmers are often constrained by limited knowledge of module interdependencies, they may wind up introducing new errors. Research on automated program repair aims to address these challenges.

Our scope includes fixing safety violations in existing programs. Finding these violations can be reduced to invariant checking [6]. Thus, without loss of generality, a program is correct if its state graph has no reachable "bad" states. This, along with the event blocking idiom of BP, enables an elegant method of repair by trimming: correcting the program by removing edges from its state graph using the blocking idiom, so that bad states become unreachable. This technique resembles the Supervisory Control model [23], where one seeks a supervisor that controls a plant by disabling transitions in the plant's state graph.

The repair is non-intrusive, i.e. performed strictly by adding new threads to the program (termed "wait-block patches"), and without modifying existing code. The patch threads are passive, in the sense that they never request any events or assign any atomic propositions to states, thus keeping the repaired program as close to the original as possible. Only when the execution gets dangerously close to a bad state does the patch block events that would cause a violation, forcing the system to choose a different execution path. In [15] it is shown that, for programs with deterministic threads, this method does not eliminate correct executions, as events are blocked only when they are guaranteed to lead to a violation. Further, no deadlocks are created as a result of such patching.

This repair technique is adequate for systems that are capable of generating the desired ("good") behavior but may, in some scenarios, produce erroneous output. For instance, patching may be applied to a variety of bugs resulting from race conditions between parallel components — fixing them by temporarily blocking one of the components, forcing it to yield to its counterpart. However, not all systems can be repaired in this way, and the repair algorithm fails gracefully in this case. A soundness result shows that if a correct patch exists, it will indeed be found by the repair algorithm.

The algorithm operates by analyzing a program's state graph and looking for the smallest fixpoint set of states that can be removed from the graph in order to render q_b, the single bad state, unreachable. Specifically, the algorithm backtracks from q_b, attempting to isolate it by trimming edges without creating

deadlocks. Whenever all the successors of a state are bad, it is marked as bad itself; see Fig. 3. Below is the repair algorithm's pseudo-code; Pre denotes the predecessor states of a given set of states.

Algorithm 3 Concrete Safety Patching

1: $BAD \leftarrow \{q_b\}$, $PRE \leftarrow Pre(BAD)$
2: **while** $\exists q \in PRE$ such that $\forall e, \text{Post}(q, e) \in BAD$ **do**
3: Move q from PRE to BAD
4: **if** q is the initial state **then return** $Failure$
5: $PRE \leftarrow Pre(BAD)$
6: **return** a patch that blocks edges from PRE to BAD

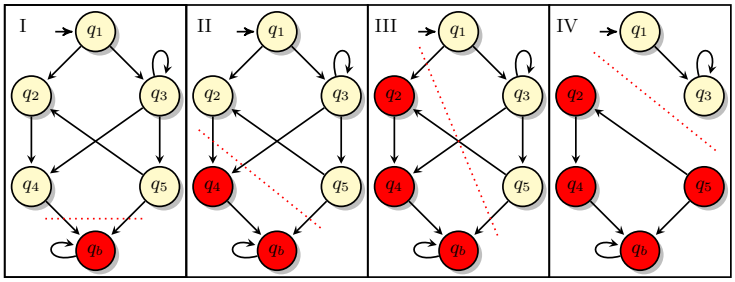

Fig. 3. The algorithm for trimming the concrete state graph of a program in order to correct a safety violation. Graph I depicts the initial configuration, with the only bad state, q_b, marked in red. The edges from states in PRE to states in BAD cross the dotted red line, and are candidates for blocking. In the first iteration, blocking these edges would cause a deadlocked in state q_4. Thus, in graph II state q_4 is also marked as bad, and q_2 joins PRE. Unfortunately, now a deadlock would be caused in state q_2, and the algorithm iterates again, putting q_2 in BAD. The next iteration puts q_5 in BAD. Only then, in graph IV, can edges crossing the dotted line be safely removed without causing deadlocks. The states in BAD are thus rendered unreachable, fixing the safety violation.

As this algorithm uses the program's concrete state graph, it does not scale to large programs. We thus seek to adjust it so it can use an over-approximation instead. Unfortunately, directly applying the concrete patching algorithm to an abstract graph yields erroneous results. In particular, the algorithm might fail when a correct answer exists, or the resulting patches might also eliminate good executions — traits that did not exist in the concrete version. See Fig. 4.

Intuitively, the reason for these failures is the fact that *patch-incompatible* concrete states are abstracted into the same abstract states. By patch-incompatible, we mean that the concrete algorithm would block a different set of events in each of the concrete states. In the abstract graph, however, such blocking becomes impossible, resulting in the algorithm's undesired behavior. In order to overcome this difficulty, we incorporate a refinement phase into the repair algorithm; however, instead of using counterexamples as means of guiding the refinement, the driving force is the need to create abstract states that correspond only to patch-compatible concrete states.

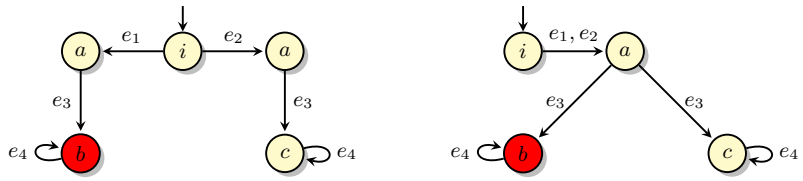

Fig. 4. A concrete state graph on the left, and its abstraction on the right. The atomic propositions appear inside the states. The safety property in question is the invariant $G\neg b$, which is violated when the states in red are reached. In the concrete graph, a simple patch can fix the problem: by blocking e_1 in the initial state, the red state is made unreachable, and no deadlocks are caused. On the abstract graph, however, no repair is possible without causing a deadlock somewhere in the program. As a result of the nondeterminism in state a, where two edges correspond to the same event, we are unable to block one edge while leaving the other enabled.

The algorithm uses an over-approximation of the state graph, in which $\overline{q_b}$ is the single abstract bad state, corresponding to q_b. As in the concrete case, we assume the concrete b-threads are deterministic. Here is the pseudo-code:

Algorithm 4 Abstract Safety Patching

$BAD \leftarrow \{\overline{q_b}\}$
while $True$ **do**
  $PRE \leftarrow Pre(BAD)$
  if $\exists \overline{q} \in PRE$ such that $NeedToRefine(\overline{q})$ **then**
    $Refine(\overline{q})$
  else if $\exists \overline{q} \in PRE$ such that $\forall e, \text{Post}(\overline{q}, e) \subseteq BAD$ **then**
    Move $\overline{q}$ from PRE to BAD
    if $\overline{q}$ is the initial state **then return** $Failure$
  else
    return a patch that blocks edges from PRE to BAD

The core of the algorithm remains the same as in the concrete case: we start at the bad state $\overline{q_b}$, backtracking and marking states that only lead to bad states as bad themselves. Once we reach a setting in which all states in PRE also have edges leading to good states (as to not create deadlocks), we return a patch trimming the edges from PRE to the bad states. The refinement phase prevents good executions from being likewise trimmed:

Algorithm 5 NeedToRefine($\overline{q}$)

$E \leftarrow \{e \in \Sigma \mid \text{Post}(\overline{q}, e) \cap BAD \neq \emptyset\}$
if exists $q \in \eta^{-1}(\overline{q}), e \in E$ such that $e \in R(q)$ and $\eta(\text{Post}(q, e)) \notin BAD$ **then**
  return $True$
if exists $q \in \eta^{-1}(\overline{q})$ such that $R(q) \subseteq E$ **then**
  return $True$
return $False$

In order to determine if an abstract state $\overline{q}$ needs to be refined, we look at the events that we would like to block in it (set E). If there exists a concrete state in $\eta^{-1}(\overline{q})$ for which $e \in E$ is requested and leads to a good state, refinement is needed to prevent good executions from being eliminated. Similarly if there exists

a state in $\eta^{-1}(\overline{q})$ that has no requested events that would remain unblocked, refinement is needed in order to avoid causing a deadlock. The actual refinement is performed as follows:

Algorithm 6 Refine($\overline{q}$)

1: For every $q \in \eta^{-1}(\overline{q})$ calculate $\mathfrak{B}(q) = \{e \in R(q) \mid \eta(Post(q,e)) \in BAD\}$
2: Form a partition $\eta^{-1}(\overline{q}) = C_1 \uplus C_2 \uplus \ldots \uplus C_k \uplus C_{deadlock}$ such that if $\mathfrak{B}(q) = R(q)$, then $q \in C_{deadlock}$; else, $q_1, q_2 \in C_i \iff \mathfrak{B}(q_1) = \mathfrak{B}(q_2)$.
3: Split abstract state $\overline{q}$ into $k+1$ new states $\overline{q_1}, \ldots, \overline{q_{k+1}}$ such that $\eta^{-1}(\overline{q_i}) = C_i$ for $1 \leq i \leq k$, and $\eta^{-1}(\overline{q_{k+1}}) = C_{deadlock}$.

Set $\mathfrak{B}(q)$ contains the events to be blocked in q. The refinement splits the problematic abstract state into multiple abstract states, each representing concrete states in which the same events need to be blocked. Observe that state $\overline{q_{k+1}}$, in which the necessary blocking will introduce a deadlock, will be put in BAD in one of the following iterations of the main algorithm.

For correctness and soundness, we present the following theorem, proven in Appendix IV of the supplementary material [2]. This result is analogous to the one for the concrete algorithm presented in [15]; hence, it demonstrates that the improved scalability does not come at the expense of the concrete version's desirable qualities.

Theorem 1. *For a behavioral program P and a violated safety property Φ,*

1. *A patch returned by algorithm 4 eliminates all bad executions of the program, does not eliminate good executions, and does not create deadlocks.*
2. *If there exists a wait-block patch that corrects P with respect to Φ, such a patch will be found by algorithm 4. Otherwise, the algorithm will issue a* Failure *notice.*

In this algorithm, the inverse global abstraction $\eta^{-1}(\overline{q})$ is computed multiple times; indeed, this is an expensive step. However, for programs that are "close to being correct", the repair algorithm may only need to perform a few refinements, hopefully terminating in reasonable time. As discussed in Section 4.2, not every refinement is obtainable in our two layered structure; see discussion therein.

6 Experimental Results

For our experiments we used the *BPC* framework for BP in C^{++}, available online [1]. We implemented the algorithms presented in the previous sections, namely thread abstraction, partitioning into modules, CEGAR verification and abstraction based patching, as a proof-of-concept tool on top of BPC. Since our goal was to show the improved scalability offered by the abstraction techniques, we also implemented concrete versions of the same algorithms in BPC. All implementations are explicit; symbolic implementation is left for future work.

We tested our algorithms on a BP based web-server application. The server, a work in progress, implements basic TCP and HTTP protocol stacks and is

compatible with the Firefox browser. Due to the server's size of several million states, BPC ran out of memory when attempting to verify it concretely.

In contrast, the abstraction based methods were able to produce an initial abstraction of the system within 22 seconds. The automated module partitioning algorithm successfully divided the threads into logically related modules along the lines of the TCP and HTTP layers, grouping the HTTP threads into a single module and dividing the TCP threads between a few modules. The resulting over-approximation contained 800 states and some 12500 transitions.

We then used this over-approximation to identify and repair a bug where the TCP stack would, under certain conditions, acknowledge a FIN message for already closed connections. Identifying this bug using the CEGAR-based verification algorithm took 9.5 minutes, and included 3 refinement phases, at the end of which a genuine counterexample was produced. Producing a patch that fixes the bug using algorithm 4 then took 38 minutes.

Our experiments were run on a 2.66 GHz T500 laptop. The model and some of the properties used for our tests are available from [2].

7 Related Work and Conclusion

The main contribution of our work is in applying abstraction techniques to behavioral programming. In particular, we propose a technique for efficiently generating over-approximations of programs, which can later be used in analysis algorithms. We demonstrate two such algorithms: a CEGAR based method for model-checking behavioral programs, and an abstraction based algorithm for the repair of safety violations. We regard this research as a step in the direction of developing more scalable methodologies and tools for formal analysis of BP.

Another contribution of our work is in the field of program repair, where we show an abstraction based algorithm that uses repair-guided refinement. Program repair is closely related to the synthesis problem, where various abstraction-refinement schemes have been proposed (e.g., [13, 18]); thus, we feel that this is a useful concept that could potentially improve the scalability of existing repair methods, not necessarily restricted to BP.

The use of abstraction-refinement based techniques to expedite model-checking has been extensively studied (e.g., [4, 10, 11, 22]) and has been implemented in several frameworks, such as SLAM [7] and BLAST [19]. Among these, the work most closely related to ours is the MAGIC framework [8, 9]. There, the authors similarly propose a two layer CEGAR approach, in which modules are abstracted separately and their abstractions then composed. However, the setting of [8,9] allows spurious counterexamples to be checked against each module separately — whereas in the setting of BP, checking involves all modules simultaneously. Analogously, refinements may not be confined to a single module.

In the area of program repair, recent work has focused on locating faulty components and then using synthesis to alter or replace them. In [20, 24], the authors seek corrections in the form of strategies that may be implemented without introducing new states (memoryless strategies), in order to alter the original

program as little as possible. We address the same need by only adding code, leaving the original program unmodified. The work of [14] discusses repairing boolean programs by using abstractions of these programs. This approach is similar to ours, but does not include a refinement phase in case spurious executions in the abstract program prevent finding a repair. In [21], the authors tackle state explosion by maintaining an under-approximation of a repair candidate, at each iteration adding more constraints that it must fulfill. New constraints are produced by checking the candidate against the concrete faulty system. This technique appears orthogonal to our own, in which the program is abstracted and the repair candidate is calculated explicitly. Attempting to combine the two methods seems promising, and is left for future work.

A different repair approach includes using genetic and co-evolutionary programming [5, 25], where a set of candidate programs is iteratively evaluated against the specification. Programs with high *fitness* survive, and are *mutated* to produce the next iteration's candidates, until a correct program is obtained. This approach handles more general bugs than ours (as it is not limited to trimming), but may extensively alter the original program's code.

In the future, we plan to extend our abstraction-based repair algorithm to handle violated liveness properties, as well safety ones. Indeed, some preliminary work we have done shows promising results. Another direction we hope to pursue is improving the performance of BPC by enhancing it with symbolic capabilities. Finally, another interesting line of work is strengthening our module-partitioning algorithm: we feel the programmer-created b-threads contain currently untapped meta data about the structure of the system, which could be utilized in making "smarter" partitions. We hope that tapping this meta data will also prove useful in the context of automated compositional verification.

Acknowledgments. We thank D. Harel for his guidance and support, O. Kupferman for her insightful remarks on this work, and the anonymous reviewers for their valuable and thorough comments. This work was supported by an Advanced Research Grant to D. Harel from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013), and by an Israel Science Foundation grant.

References

1. BPC: Behavioral Programming in C^{++}. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/`.
2. Supplementary material. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/module_based_abstraction/`.
3. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-Time Temporal Logic. *Journal of the ACM*, 49(5):672–713, 2002.
4. N. Amla and K. L. McMillan. Combining Abstraction Refinement and SAT-Based Model Checking. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 405–419, 2007.
5. A. Arcuri and X. Yao. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *Proc. 10th IEEE Congress on Evolutionary Computation (CEC)*, pages 162–168, 2008.

6. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
7. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. 8th Int. Workshop on Model Checking of Software (SPIN)*, pages 103–122, 2001.
8. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering*, pages 385–395, 2004.
9. E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient Verification of Sequential and Concurrent C Programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.
10. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2000.
11. E. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. In *Proc. 19th. Symposium on Principles of Programming Languages (POPL)*, pages 343–354, 1992.
12. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
13. L. de Alfaro and P. Roy. Solving Games via Three-Valued Abstraction Refinement. In *Proc. 18th Int. Conf. on Concurrency Theory (CONCUR)*, pages 74–89, 2007.
14. A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for c programs. In *Proc. 18th Int. Conf. on Computer Aided Verification (CAV)*, pages 82–99, 2006.
15. D. Harel, G. Katz, A. Marron, and G. Weiss. Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 3–12, 2012.
16. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
17. D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90–100, 2012.
18. T. A. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided Control. In *Proc. 30th Int. Conf. on Automata, Languages and Programming (ICALP)*, pages 886–902, 2003.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Proc. 10th Int. Workshop on Model Checking of Software (SPIN)*, pages 235–339, 2003.
20. B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. In *Proc. 17th Int. Conf. on Computer Aided Verification (CAV)*, pages 226–238, 2005.
21. R. Könighofer and R. Bloem. Repair with On-The-Fly Program Analysis. In *Proc. 8th Haifa Verification Conference (HVC)*, pages 56–71, 2012.
22. K. L. McMillan and L. D. Zuck. Abstract Counterexamples for Non-disjunctive Abstractions. In *Proc. 3rd Int. Workshop on Reachability Problems (RP)*, pages 176–188, 2009.
23. P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. on Control and Optimization*, 25(1):206–230, 1987.
24. S. Staber, B. Jobstmann, and R. Bloem. Diagnosis is Repair. In *Proc. 16th Int. Workshop on Principles of Diagnosis (DX)*, pages 169–174, 2005.
25. W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic Program Repair with Evolutionary Computation. *Communications of the ACM*, 53:109–116, 2010.