Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs

Article · January 2014

DOI: 10.1007/978-3-662-44871-7_1

4 authors:

David Harel
Weizmann Institute of Science
408 PUBLICATIONS **20,419** CITATIONS
SEE PROFILE

Guy Katz
Hebrew University of Jerusalem
30 PUBLICATIONS **343** CITATIONS
SEE PROFILE

Assaf Marron
Weizmann Institute of Science
44 PUBLICATIONS **361** CITATIONS
SEE PROFILE

Gera Weiss
Ben-Gurion University of the Negev
60 PUBLICATIONS **656** CITATIONS
SEE PROFILE

Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs

David Harel[1], Guy Katz[1], Assaf Marron[1], and Gera Weiss[2]

[1] Dept. of Computer Science and Applied Mathematics
Weizmann Institute of Science
Rehovot, Israel
firstname.lastname@weizmann.ac.il
[2] Dept. of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
geraw@cs.bgu.ac.il

Abstract. We show how, under certain conditions, programs written in the *behavioral programming* approach can be modified (e.g., as a result of new requirements or discovered bugs) using automatically-generated code modules. Given a trace of undesired behavior, one can generate a relatively small piece of code, whose execution is interwoven at run time with the rest of the system, and which brings about the desired changes without modifying existing code and without introducing new bugs. At the core of our approach is the ability of a thread of behavior to prevent the triggering of events from other threads. Our repair algorithms apply model checking of safety and liveness properties to the program and transform the counterexamples produced by the model-checker into corrective modules. The work is supported by a proof-of-concept tool, which creates understandable modules that can be further manually managed as part of a process of ongoing incremental system development.

Keywords: Program repair; verification; behavioral programming; model checking; patching.

1 Introduction

Software maintenance is a difficult and error prone task. As errors (bugs) are discovered and requirements are added or changed, developers work hard to modify existing code without introducing new errors. They are often constrained by limited knowledge of possible side-effects, since undocumented interdependencies might have been forgotten or might be known only to different people (usually, the original developers) who are unavailable. Research on automated program repair and, more generally, program synthesis from specifications, aims to address these and related challenges. Such automation may prove particularly valuable for handling failure/bug reports from users who simply press the *"Send* *to Software Vendor"* button. In such cases, the software engineer cannot discuss with the user the context of the problem, or possible generalizations thereof.

In this paper we focus on programs written in the *behavioral programming* approach, and our work is centered on the idea of repairing by carefully forbidding existing faulty execution paths. This technique is highly suitable for (a) non-intrusive incremental repair; i.e., large parts of the system are already developed and are not modified by the repair process; (b) methodological integration of the repair process with standard, ongoing development during and after the repair activity; and (c) practical techniques for dealing with the complexity of the use of model-checking when creating local patches in the repair process.

2 Background

Our work is carried out within the *behavioral programming* approach [11, 12] — an extension and generalization of scenario-based programming, which was introduced with the language of live sequence charts (*LSC*s) [6, 10], and is now implemented also in Java [11] and Erlang [13, 25].

A behavioral program consists of independent threads of behavior that are interwoven at run time. Each *behavior thread* (abbr. *b-thread*) specifies events and event sequences which, from its own point of view must, may, or must not occur. As shown in Fig. 1, the infrastructure synchronizes and interweaves all behaviors, selecting events that constitute integrated system behavior without requiring direct communication between b-threads. Specifically, all b-threads declare events that should be considered for triggering (called *requested events*) and events whose triggering they forbid (*block*), and then synchronize. An event selection mechanism then triggers one event that is requested and not blocked, and resumes all b-threads that requested the event. B-threads can also declare events that they simply "listen-out for", and they too are resumed when these waited-for events occur.

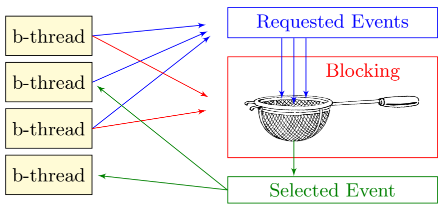


Fig. 1. Behavioral programming execution cycle: all b-threads synchronize, declaring requested and blocked events; a requested event that is not blocked is selected and b-threads waiting for it are resumed.

This facilitates incremental non-intrusive development as outlined in the example of Fig. 2.

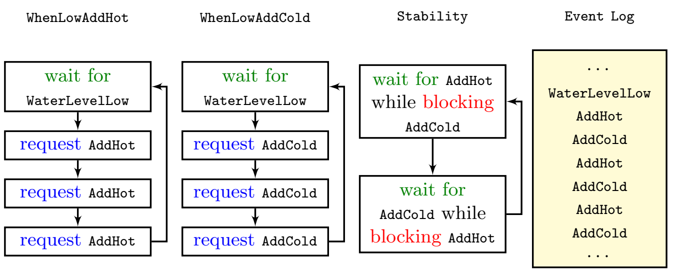


Fig. 2. Incremental development of a system for controlling water level in a tank with hot and cold water sources. The b-thread `WhenLowAddHot` repeatedly waits for `WaterLevelLow` events and requests three times the event `AddHot`. `WhenLowAddCold` performs a similar action with the event `AddCold`, reflecting a separate requirement, which was introduced when adding three water quantities for every sensor reading proved to be insufficient. When `WhenLowAddHot` and `WhenLowAddCold` run simultaneously, with the first at a higher priority, the runs will include three consecutive `AddHot` events followed by three `AddCold` events. A new requirement is then introduced, to the effect that water temperature should be kept stable. We add the b-thread `Stability`, to interleave `AddHot` and `AddCold` events. For details about how sensor and actuator b-threads interact with the physical environment (sensors, valves) without suspending the entire system see [13].

More detailed examples showing the power of incremental modularity in behavioral programming appear in [11, 13]. Briefly, in a program we wrote for playing Tic-Tac-Toe [11], each game-rule is implemented in a dedicated b-thread; e.g. *"block X moves when it is O's turn"* or *"block marking of already-marked squares"*. Similarly, player-strategy modules are oblivious of other strategies; e.g., *"wait for two X marks in the same line, and then request marking O in that line"*. A similar technique can be used to control a robot performing simultaneous missions, such as vehicle operation and route management. In stabilizing a quadrotor — an unmanned flying vehicle with four rotors — each of four b-threads in our program controls a particular orientation angle, or the quadrotor's altitude, solely by changing rotor speeds; see [13].

Each b-thread repeatedly requests and blocks events representing possible increases or decreases of rotor RPM, which could contribute to its own goal. The triggering of an event that is requested by one or more b-threads and blocked by none allows at least one b-thread to progress. Affected b-threads can then recalculate their declarations of requested and blocked events, and the process repeats.

In [8] and [15], model-checking and planning algorithms (respectively) are applied to play-out, the method for executing LSCs. These smart play-out techniques control the choice of the event to be triggered, such that, within the next superstep (i.e., prior to the next event driven by the environment), the specification is not violated by the program (if this is possible). In [9], a proof-of-concept model checker verifies behavioral Java programs "in vivo" - without first translating them into a model-checker-specific language. It is further shown in [9] how, when a problem is detected, the programmer can develop and add a b-thread that repairs the program by refining the behavior without modifying existing code.

3 Outline of the Repair Approach

In the present paper we utilize the model checker of [9] to automate elements of manual program-repair processes, using a principle that can be summarized as "taking the road not taken". For illustration, assume that a system was tested, or even model-checked, to satisfy its specification, and a new requirement was then introduced, or a bug reported, highlighting a required property not previously articulated, and thus neither tested nor model-checked. Our method calls for first adding the new property to the specification. We then model-check the program to find distinct violating runs. In the case of violated *safety* properties ("bad things never happen"), for each such run we add a special b-thread that waits for the sequence of all events in the run, up to the last one requested by the program (rather than by the environment). The repair b-thread then blocks this event. Some other pending requests might then be triggered. Violated *liveness* properties ("good things eventually happen") are handled similarly: when the system is traversing a loop in which "good things do not happen", the repair b-thread applies blocking to steer the run in another direction. In the liveness case blocking is only performed with some small probability, thus injecting bias towards certain desirable execution paths without forbidding other paths which are also permitted.

For example, consider a faulty game-strategy b-thread, whose event request leads to a loss. When this event is blocked, another b-thread, perhaps one that requests a set of default moves, comes into play (so to speak), offering an alternative. The elimination process continues until "the right" default move is the choice at that state. The new corrective wait-and-block behavior is non-intrusive, in that its implementation does not require changing the existing program code.

We refer to such a repair b-thread as *a patch*, and to the process as *patching*, or simply, *repairing*. We hope that combined with the behavioral-programming principles, our approach will help make the concept of patching seem less a "necessary evil" and more a useful, mainstream software maintenance practice.

As the full repair algorithm may not scale up to large programs due to the state explosion problem, we also discuss the case where patching can be limited to a bounded "neighborhood" of a specific operation scenario; for example, when we are provided with a bug report sent from a user.

We formally prove correctness and analyze the method, characterize the programs on which it can be used, and exemplify its usage with our proof-of-concept tool.

The rest of this paper is organized as follows. Basic definitions of behavioral programs and their model-checking are given in Sections 4 and 5, respectively. The repair of safety violations of loopless programs is discussed in Section 6, followed by a repair algorithm for safety violations in general programs in Section 7. Next, in Section 8 we extend the algorithm to also handle liveness violations. A method for handling large programs, called limited-depth patching, is described in Section 9. Related work is discussed in Section 10, and we conclude with Section 11.

4 Definitions

While behavioral programming is geared towards natural and intuitive development using almost any programming language, its underlying infrastructure can be conveniently described and analyzed in terms of transition systems.

4.1 The Behavioral Programming Computational Model

The definitions below follow [9, 13] and were modified to include the notion of a b-thread tagging states of the system as having certain properties, commonly termed *atomic propositions (AP)* [3]. Recall that a *deterministic labeled transition system* is a 6-tuple $\langle S, E, \rightarrow, init, AP, L\rangle$, where S is a set of states, E is a set of events, $\rightarrow$ is a (possibly partial) function from $S \times E$ to S, $init \in S$ is the initial state, AP is a set of *atomic propositions*, and $L : S \rightarrow 2^{AP}$ is a labeling function. The *runs* of a transition system are sequences of the form $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_i} s_i \cdots$, where $s_0 = init$, and for all $i = 1, 2, \cdots$, $s_i \in S$, $e_i \in E$, and the function $\rightarrow$ maps the pair $\langle s_{i-1}, e_i\rangle$ to s_i, written $s_{i-1} \xrightarrow{e_i} s_i$. We say that $\langle S, E, \rightarrow, init\rangle$ is total if the function $\rightarrow$ is total.

Behavior threads are modeled as transition systems, with S, E, and AP finite, and the states being associated with event sets:

Definition 1. *A* behavior thread *(abbr.* b-thread*) is a tuple $\langle S, E, \rightarrow, init, AP, L, R, B\rangle$, where $\langle S, E, \rightarrow, init, AP, L\rangle$ forms a deterministic total labeled transition system, $R\colon S \rightarrow 2^E$ associates a state with the set of events* requested *by the b-thread when in it, and $B\colon S \rightarrow 2^E$ associates a state with the set of events* blocked *by the b-thread when in it.*

Definition 2. *The* runs of a set of b-threads $\{\langle S_i, E_i, \rightarrow_i, init_i, AP_i, L_i, R_i, B_i\rangle\}_{i=1}^n$ *are the runs of the labeled transition system $\langle S, E, \rightarrow, init, AP, L\rangle$, where $S = S_1 \times \cdots \times S_n$, $E = \bigcup_{i=1}^n E_i$, $init = \langle init_1, \ldots, init_n\rangle$, and $\rightarrow$ includes a transition $\langle s_1, \ldots, s_n\rangle \xrightarrow{e} \langle s'_1, \ldots, s'_n\rangle$ if and only if*

$$\underbrace{e \in \bigcup_{i=1}^{n} R_i(s_i)}_{e \text{ is requested}} \qquad \bigwedge \qquad \underbrace{e \notin \bigcup_{i=1}^{n} B_i(s_i)}_{e \text{ is not blocked}}$$

and

$$\bigwedge_{i=1}^{n} \Big(\underbrace{(e \in E_i \implies s_i \xrightarrow{e}_i s'_i)}_{\substack{\text{affected b-threads}\\ \text{move}}} \wedge \underbrace{(e \notin E_i \implies s_i = s'_i)}_{\substack{\text{unaffected b-threads}\\ \text{don't move}}} \Big).$$

We set $AP = \bigcup_{i=1}^n AP_i$ and, for $(s_1, \ldots, s_n) \in S_1 \times \ldots \times S_n$, we define:

$$L(s_1, \ldots, s_n) = L_1(s_1) \cup \ldots \cup L_n(s_n).$$

Note that when implemented in a standard programming language, we assume that b-threads do not share data, and rely solely on events for input and output. This results in the abstraction that a behavior thread is "in a state" only when synchronized with others, and that the state transition caused by executing program instructions between synchronization points is atomic.

Observe that while each b-thread is deterministic in its reaction to events, Definition 2 does not specify how events are selected, and thus there may be more than one run for a given set of b-threads. There could be multiple ways to select events and runs, including ones that are random, planned, or priority-based. The default behavioral execution infrastructure of LSC (in the *Play-Engine* and *PlayGo* tools), the Java package (*BPJ*) and the Erlang module (*bp*) executes a set of b-threads based on priorities. That is, in each state of the composite system, the first event that is requested and is not blocked is selected for triggering.

Definition 3. *For the transition system T, defined in Definition 2, a (deterministic)* event selection mechanism *is a function $f\colon S \to E$, such that for each $s \in S$ there exists a transition $s \xrightarrow{f(s)} s'$ of T.*

Behavioral programming is designed particularly for the development of *reactive systems* [14], and in this context it is critical to distinguish between environment behavior and program behavior.

Definition 4. *A* reactive behavioral program *is a set of b-threads, an event selection mechanism, and a partition of the events of the b-threads into external events representing uncontrollable occurrences coming from the environment, and internal events completely controlled by the program.*

We denote the set of external events by E_{env}, and the set of internal events by E_{prog}. By convention, the patches we present in this work may block only the triggering of events in E_{prog} and may not block events in E_{env}.

4.2 Specifications

We now introduce definitions that assist in the discussion of desired and undesired runs of behavioral programs.

Definition 5. *For a set of b-threads P and a run $\rho = (e_1, e_2, \ldots,)$, such that the execution corresponding to ρ is $s_{init} \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \ldots$, we define $APtrace(\rho) = L(s_{init})L(s_1)L(s_2)\ldots$ and define the set of all traces of P to be $APtraces(P) = \{APtrace(\rho) \mid \rho \in runs(P)\}$.*

Definition 6. *A* specification *for a behavioral program P is a linear time (LT) property Φ (i.e. a subset of $\left(2^{AP}\right)^{\omega}$). We say that P satisfies Φ, denoted $P \vDash \Phi$, iff $APtraces(P) \subseteq \Phi$.*

Since this definition assumes infinite runs, when dealing with systems of finite runs we pad any finite run with the trace $\emptyset^\omega$.

It is important to note, that the same set of b-threads can satisfy Φ with one event selection mechanism, and not with another. We adopt a wider perspective here, and ensure that the patched set of b-threads satisfies Φ with *all* event selection mechanisms. Such patching immediately detects and fixes any bugs that could have remained hidden with a certain mechanism, but which may emerge later. An approach that takes a specific event selection mechanism into account may also be useful for some applications.

In this work we focus on two major types of LT properties: safety properties and liveness properties. We define safety properties first, and give also the related definitions of invariants and deadlocks.

Definition 7. *An LT property Φ over AP is called a* safety *property if for all $\sigma \in (2^{AP})^\omega - \Phi$ there exists a finite prefix $\bar{\sigma}$ of σ such that*

$$\Phi \cap \left\{ \sigma' \in \left(2^{AP}\right)^\omega \;\mid \bar{\sigma} \text{ is a finite prefix of } \sigma' \right\} = \phi.$$

Intuitively, a safety property states that no “bad” sequences of events may happen. Any run that causes such a sequence has a *bad prefix*; after it the run does not satisfy the property no matter how it continues.

The notion of *invariants* plays a key role in the model-checking of safety properties:

Definition 8. *An LT Φ property over AP is an* invariant *if there is a propositional logic formula φ over AP such that $\Phi = \left\{A_0 A_1 A_2 \ldots \in \left(2^{AP}\right)^\omega \;\mid \forall j \geq 0, A_j \vDash \varphi\right\}$.*

Intuitively, invariants are properties of the current state of the system, and do not reflect the history of events leading to it.

Through invariant checking one can handle *regular safety properties*: those safety properties for which the associated bad prefixes are recognizable by some finite automaton [3], or, in our case, there is a b-thread that marks its state as bad when the bad prefix is recognized. By applying the invariant model-checker to a program with these threads added, we can effectively handle general regular safety properties.

Definition 9. *We say that a (finite) run $\rho = (e_1, e_2, \ldots, e_n)$* causes a deadlock *if it leads to a state s that has no enabled events (all requested events are also blocked).*

Much like invariants, deadlocks too are properties of states in the system, and not of runs.

When patching against safety violations, we will receive as input a program P and an invariant Φ. We will implicitly check that the system has no deadlocks; if it does, the patching algorithm will try to remove them. In particular, we will make sure that no new deadlocks are created while patching; otherwise we could “patch” a system by simply blocking all enabled events at its initial state.

The other type of properties we consider is liveness properties. The following is adopted from [3]:

Definition 10. *An LT property Φ over AP is called a* liveness *property if any finite word can be extended such that the resulting infinite trace satisfies Φ. Formally, let pref(σ) $= \{\bar{\sigma} \in (2^{AP})^* \mid \bar{\sigma}$ is a finite prefix of $\sigma\}$ and pref(Φ) $= \bigcup_{\sigma\in\Phi}$ pref(σ). Then Φ is a liveness property if and only if pref(Φ) $= (2^{AP})^*$.*

In the case of regular safety properties, invariant checking plays a key role. When it comes to liveness properties, a similar role is played by *persistence* checking:

Definition 11. *An LT property Φ over AP is called a* persistence *property if it states that a certain condition holds forever, from some point in Φ. Formally, Φ is a persistence property if there exists a propositional logic formula φ such that $\Phi = \{A_0A_1A_2\ldots \in (2^{AP})^\omega \mid \exists_i$ such that $\forall_{j\geq i},\ A_j \vDash \varphi\}$. Formula φ is called the persistence (or state) condition of Φ.*

As discussed in, e.g., [3], the model-checking of regular liveness properties is reducible to persistence checking. The latter is performed by portioning the states of the system into two sets: states in which φ holds, termed “cold” states, and states in which it does not hold, termed “hot” states. Then, the property holds if and only if there are no reachable cycles consisting strictly of hot states (which we refer to as reachable “hot cycles”). This can be checked, for instance, using a nested DFS algorithm.

When patching against liveness violations, we will receive as input a program P and a persistence property Φ. In practice, this property is given by an indicator thread that marks the system's states as either hot or cold.

5 Extending the Model-Checking of Invariants and Deadlocks

In order to prepare the ground for the correction of various safety and liveness violations, we begin by describing how to check that a behavioral program satisfies an invariant and is deadlock-free. We follow the algorithm in [3], section 3.3.1, and the implementation in [9].

Any state that violates the invariant or is deadlocked is marked as “bad”. We construct the state graph of the program, traverse it using DFS (trimming when arriving at a previously visited state), and check that all states reachable from the initial state are not bad. From each state we explore all enabled events (which reflects our decision to cater for all possible event selection mechanisms).

The runtime complexity of this algorithm, implemented as in [9], is as follows. Let $G = (V_G, E_G)$ denote the state graph constructed, and let n be the number of threads and $e = |E|$ the number of events in the original program. $|V_G| + |E_G|$ operations have to be performed to traverse the graph. Further, for each $state \in$

V_G we have to perform $n \cdot e$ operations in order to find all its enabled events. This yields:

$$T_{mc} = O\left(|E_G| + |V_G| \cdot (n \cdot e)\right).$$

This complexity is the minimum price one has to pay for running a model-checker on a behavioral program. Since our technique is based on model-checking, it will necessarily be forever linked in complexity to that of model checking [3,22], and the progress made there, for better or for worse. T_{mc} thus serves a base point with which to compare the complexity of our patching algorithms, and we are interested in how much additional overhead they incur above it.

We actually use a slightly different algorithm. For our purposes, the usual model-checking that returns a single violating run does not suffice: we want to explore *all* runs that violate the invariant or cause a deadlock.

This is achieved as follows: we traverse the state graph using the same DFS, but whenever we reach a bad state we store that information in its predecessor states. Each state already visited in the graph will thus contain information on all its bad successors. If the state is reached again, through another route from the root, we need not traverse its subtree again: we simply update the relevant states using the data already stored (see Fig. 3).

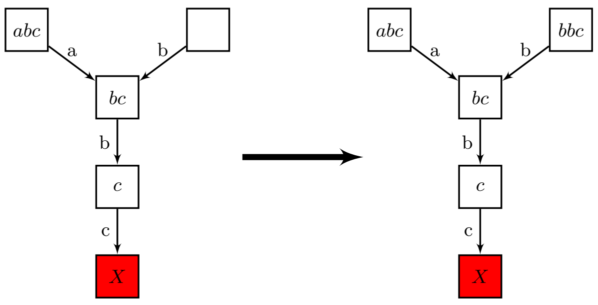


Fig. 3. When a “bad” state is reached, all its predecessors store the relative path from that point to the violation. When a node in this path is reached through a different path, the data is propagated. The DFS continues until the root stores all violating paths.

The added complexity of this algorithm is measured using the number of violating runs, $\varUpsilon$ (OOPSilon: pun intended), and the depth of the state graph D. For each violating run we propagate at most D events to the predecessors, causing an overhead of $1 + 2 + \ldots + D$ per violating run. The total runtime complexity is thus:

$$T = T_{mc} + \varUpsilon \cdot (1 + 2 + \ldots + D) = T_{mc} + O(\varUpsilon \cdot D^2).$$

Finally, if all direct successors of a state are bad, then the state itself can be considered bad; this is because the patching technique we discuss will cut off the violating children, rendering the state a deadlock. We thus add the following modification: if, during the DFS, all of a state's successors are violating or deadlocked, the state itself is marked as violating; thus its successors can be ignored. The runtime worst-case complexity remains unchanged.

6 Safety Patches for Loopless Programs

6.1 Generating Linear Safety Patches

Before discussing the safety violation patching of general programs, we begin with the simpler case of finite programs that are *loopless*: their state graph contains no cycles. In a loopless program, every run is finite.

Definition 12. *A* linear safety wait-block patch *for event sequence $(e_1, e_2, \ldots, e_n, e_{last})$, such that $e_{last} \in E_{prog}$, is a b-thread with the following properties:*

- *The patch waits for events $e_1, \ldots, e_n$, blocks e_{last} once and then terminates.*
- *If the run deviates from the sequence $e_1, \ldots, e_n$, the patch terminates.*
- *The patch never requests events and does not label states ($R(s) = L(s) = \emptyset$ for all s).*

Intuitively, the patch is designed to prevent one bad run from occurring. Events $e_1, \ldots, e_n$ will be chosen according to violating runs found by the model-checker. The patch will intervene before the last event, causing another event to be triggered, thus preventing the violation.

The patch only interferes with runs starting with events $e_1, \ldots, e_n$; other runs remain unchanged. Formally:

Lemma 1 (The Locality Lemma). *Let P be a collection of b-threads, let p be a linear safety wait-block patch for event sequence $(e_1, \ldots, e_n)$, and let $P' = P \cup \{p\}$ denote the patched program. Then for any run ρ of P that* does not *start with events $e_1, \ldots, e_n$, the events of ρ constitute a valid run ρ' of P', and $APtrace(\rho) = APtrace(\rho')$.*

Proof. To prove that ρ' is a valid run of P', we need to show that at each synchronization point during ρ', the triggered event is also enabled; namely, it is requested and not blocked. By definition, if a run does not start with events $(e_1, \ldots, e_n)$, then the patch never requests or blocks events. Further, the original b-threads will reach the same states during ρ' as they did during ρ, consequently requesting and blocking the same events. It follows that the program P' has the same requested and blocked events as P in each state during the run. Thus, the events triggered by ρ' are enabled, and the run is indeed valid.

Finally, since the original b-threads reach the same states during ρ' as during ρ, they will have the same atomic propositions associated with them. Since the patch has no atomic propositions associated with its states, we get that $APtrace(\rho) = APtrace(\rho')$. □

The Locality Lemma is our motivation for patching: it states (in this case, for linear patches) that when we add a patch to negate a single bad run, other runs remain unharmed, meaning that the patch is local. This is an advantage of our method as compared to traditional, manual, patching: our patches do not create new errors in unexpected parts of the code.

The distinct bad runs representing the bug or emanating from the new safety requirement are found by model-checking:

Linear Safety Patching(P, Φ):

```
1: Run the model checker on $(P, \Phi)$
2: if $P \models \Phi$ then
3:    return $P$
4: P' ← P
5: for each violating run $(e_1, \ldots, e_n)$ do
6:    if $\forall i, \quad e_i \in E_{env}$ then
7:       return Failure
8:    else
9:       Find the largest $k$ such that $e_k \in E_{prog}$
10:      Create a linear safety wait-block patch $p$ for $(e_1, \ldots, e_k)$
11:      $P' \leftarrow P' \cup \{p\}$
12: return $P'$
```

The idea is straightforward: the model-checker finds all runs violating Φ and we add a patch per run to prevent them. Because Φ is a safety property and P is loopless, there are only finitely many violating runs. The algorithm guarantees that the blocking performed by the patches creates no deadlocks, by first recursively marking as "bad" any state that has only "bad" children. Furthermore, because the model-checker works with respect to all possible event selection mechanisms, any bugs that emerged after the patching are fixed. The Locality Lemma guarantees that no good runs "far away" from the patch are harmed. If the algorithm returns a patched program, we thus know that it satisfies the specification Φ and causes no deadlocks.

There is also the case where the algorithm returns a failure notice, as a result of the model checker returning a violating run in which there were no program-requested events. This, of course, means that the program cannot be repaired through wait-block patching. Formally:

Lemma 2 (The Patchability Lemma). *Let P be a loopless program with state graph $G = (V_G, E_G)$ and let Φ be a safety property. Then the following three statements are equivalent:*

1. *The algorithm succeeds in returning a patched program P'.*
2. *There exist linear safety wait-block patches $p_1, \ldots, p_k$, such that $P \cup \{p_i\} \models \Phi$.*
3. *There exists a graph $G' = (V_G, E_{G'})$ with $E_{G'} \subseteq E_G$ and $E_G - E_{G'} \subseteq E_{prog}$, such that no states violating Φ or causing deadlocks are reachable from the initial state in G'.*

Proof. (1) $\Rightarrow$ (2) is trivial.

For (2) $\Rightarrow$ (3): Take the original state graph G, and for each p_i remove the edge corresponding to the event it blocks. Since the patched program satisfies Φ and does not deadlock, all reachable states in the graph obtained in this way satisfy Φ and do not cause deadlocks. Furthermore, by the definition of a wait-block patch, all edges removed are in E_{prog}, as needed.

For (3) $\Rightarrow$ (1): Without loss of generality, assume that P starts with an initialization event $e_{init} \in E_{prog}$. If this does not hold we can change to a new initial state s'_{init} and add a thread that forces event e_{init} to be chosen before proceeding to the original program.

Suppose that G' exists but that the algorithm returned a failure notice. We conclude that it deadlocked on the very first state, s'_{init}. This, in turn, means that state s_{init} was marked as bad, so that all paths starting in s_{init} lead to bad states. This contradicts the existence of G', thus proving the claim. $\square$

Condition (3) means that the original program was "not too far" from satisfying Φ: it contained some good runs and some bad runs, and through some blocking the bad runs could be averted. Observe that the equivalence of (1) and (2) is really the validity of the algorithm.

The worst case runtime complexity of the algorithm is just that of the modified model-checker, namely $T = T_{mc} + O(\Upsilon \cdot D^2)$. This shows the dependence of our algorithm on the number of violating runs in the original program. If their number and lengths are small enough our automatic patching is not much worse than regular model-checking. This also demonstrates why using this algorithm for synthesis could be costly. If the program is "far away" from satisfying Φ, as could be the case when trying to synthesize a program from scratch (say, from a general program that constantly requests all possible events), then Υ could be polynomial in the size of the state graph, greatly slowing the process.

6.2 Patching for a Specific Event Selection Mechanism

The above algorithm patches the program so that it satisfies Φ, regardless of the event selection mechanism used. However, it may be useful to patch the program for the specific mechanism M to be used, as it could speed up the patching process, reduce the number of generated patches, and most importantly, block less events, leaving open more options for further behavior refinements and repair, as explained in Fig. 4.

In this case, the model-checking algorithm is modified to return as output all violating runs of the original program, as well as all (and only) violating runs that would be created by blocking previously discovered bad transitions. Bad runs that will not be possible in the patched program, under the specific ESM, are ignored. This technique is readily applicable also to patches for programs with cycles and for liveness patches, discussed in the sequel.

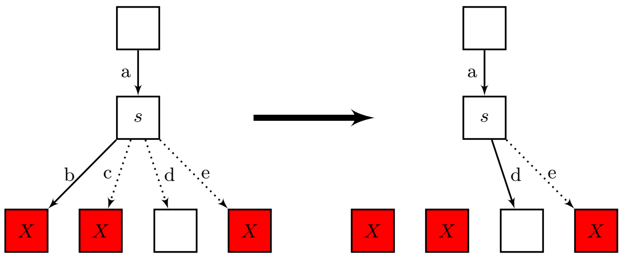


Fig. 4. In state s, a patch that considers all event selection mechanisms will block b,c, and e. A patch that considers only, say, an ESM that chooses events alphabetically, needs to block b and c, but can leave e unblocked, relying on the selection of d.

6.3 Example: Patching Tic-Tac-Toe

We demonstrate the use of the linear safety patching algorithm on the loopless Tic-Tac-Toe behavioral program from [9]. It is loopless since the fact that each step adds a new move to the board means that its state graph has no cycles.

Suppose that the original program is developed without a model-checker. At the time of development, the programmer is convinced that the program always achieves its goal, i.e., never loses (observe that this is a safety property — bad things do not happen). Various testers support this statement. The program is then deployed. Some months later, a customer defeats it and sends in the game's trace. However, the original software engineer has long quit the firm, and it would take a long time for a new engineer to repair the code. A suitable solution would be to apply an automatic patching algorithm to the malfunctioning software.

To simulate this, we took the complete program from [9], and omitted the more complex threads — those that handle situations where our opponent creates, simultaneously, two ways to win. If the human player does not try the complex strategy that create such double attacks, the program does indeed seem to work, but a skilled player can defeat it.

The automatic proof-of-concept tool is easy to use, requiring little modifications to the original program. The input is the behavioral program and the safety property Φ, given as b-threads marking bad states (e.g., victory of the opponent). The output is code files for new thread instances which are easy to read and to integrate into the original program (see Fig. 5).

Each such patch inherits from a parent class which implements its "main" function; see Fig. 6.

In our example, the patched Tic-Tac-Toe program contains 26 different patches, one of which is demonstrated in the figure. Subsequent verification by the model checker confirms that now the specification is indeed satisfied.

```
public patch1() {
   events.add(new X(2,2));
   events.add(new O(1,1));
   events.add(new X(0,0));
   events.add(new O(2,0));
}
```

Fig. 5. Example of a wait-block patch generated by the proof-of-concept tool. The patch's code contains a sequence of events that should be waited-for — events X(2,2), O(1,1) and X(0,0). The last event in the list, O(2,0), is the one that should be blocked by the patch. The automatically generated code is legible and comprehensible, as the more complicated details are hidden away in a parent class.

```
public void runBThread() {
   for (int i=0; i<events.size()-1; i++) {
      bp.bSync( none, all, none );
      if (!lastEventWas(events.get(i)))
         disablePatch();
   }
   bSync( none, all, events.getLast() );
   disablePatch();
}
```

Fig. 6. The patch thread's main function, `runBThread()` is part of the patching library, and is not added to the actual patched program. It waits for events defined by a particular patch instance (as in Fig. 5), blocking the last event and then terminating. If the events chosen deviate from those defined in the patch instance, it terminates.

7 Safety Patches for Programs with Cycles

7.1 Generating Safety Patches for Cycles

The correctness of the algorithms for linear safety patching relies on the program's state graph's having no cycles. As most reactive systems run indefinitely, periodically returning to some "idle" state, such systems cannot be patched by linear wait-block patches. For example, fixing a behavioral program that enters a bad state after a sequence of events of the form $(a)^*b$, will call for infinitely many linear patches.

Our solution is to extend the linear safety patch associated with a single sequence of events, into one that can keep track of an entire hierarchy of paths and cycles in the graph, blocking the violating event as needed.

Definition 13. *Given a state graph $G' = (V_{G'}, E_{G'})$, two special vertices marked v_{init} and v_{end} and an event $e \in E_{prog}$, a* cyclic safety wait-block patch *for G' is a b-thread with the following properties:*

- *It waits for all events chosen by the event selection mechanism and traverses the graph G' according to those events.*
- *Whenever state v_{end} is reached, it blocks event e once.*
- *If an event occurs such that there is no edge marked with that event, it terminates.*
- *It never requests events and does not label states.*

Intuitively, the patch is designed to prevent a family of bad runs that are similar to one another, in that they reach their bad state by transitioning from v_{end} via the event e. The graph G' will be chosen such that it contains *all* paths from v_{init} to v_{end}, thus rendering a single patch able to block that entire family of bad runs.

The Locality Lemma holds for the cyclic case as well: all runs of the original system, apart from those starting in v_{init} and ending in reaching the violating state through v_{end} and e, are valid runs of the patched system. The proof is based on the fact that in any such run, the generated patch does not request or block any events, and thus does not affect the events requested by the program.

Linear safety patches are a particular case of the cyclic ones, in which the graph G' is a path, meaning there is precisely one way to reach the violating state.

The cyclic safety patching algorithm is as follows (G denotes the full state graph traversed by the model-checker):

```
Cyclic Safety Patching($P, \Phi$):
 1: Run the model checker on $(P, \Phi)$
 2: if $P \models \Phi$ then
 3:    return $P$
 4: for each violating run $(e_1, \ldots, e_n)$ do
 5:    if $\forall i,\ \ e_i \in E_{env}$ then
 6:       return Failure
 7:    else
 8:       Find the largest $k$ such that $e_k \in E_{prog}$
 9:       Let $s_{end}$ denote the state reached after events $e_1, \ldots, e_{k-1}$
10:       Construct the minimal subgraph $G'$ containing all paths in $G$ from $s_{init}$ to
          $s_{end}$
11:       Create a cyclic wait-block patch $p$ for $G'$ with states $v_{init} = s_{init}$, $v_{end} = s_{end}$,
          and event $e_k$.
12:       $P' \leftarrow P' \cup \{p\}$
13: return $P'$
```

Constructing the minimal subgraph G' is done using a modified BFS algorithm, in the following manner. Given the full graph and the two vertices s_{init} and s_{end}, we run a modified BFS search from s_{init}. Unlike a regular BFS search, where each vertex stores a single predecessor (the first vertex from which it is found), here each vertex stores *all* the vertices from which it is found. When the search is over, we begin in s_{end} and backtrack through all possible predecessors of each vertex, until reaching s_{init}. The set of edges and vertices traversed this way forms the subgraph G' that we need.

To show that every path from s_{init} to s_{end} is in G', let $p = (s_{init}, s_1, \ldots, s_n, s_{end})$ be a path. If p is simple, i.e., no state repeats itself, then clearly after $n+1$ iterations of the BFS search each vertex in p has its preceeding state marked as a predecessor. Therefore, the entire path will be traversed during the backtrack phase, meaning that p is in G'.

Now, suppose that p is a complex path with one cycle (the proof for the general case is an easy extension). Then p can be expressed as follows:

$$p = (s_{init}, s_1, \ldots, s_k, \underbrace{s'_1, s'_2, \ldots, s'_j, s_k}_{\text{the cycle}}, s_{k+1}, \ldots, s_n, s_{end})$$

The states before and after the cycle are found as before. The cycle's states, $s'_1, \ldots, s'_j$, are found at the latest during the j'th iteration after the first arrival at s_k. When the cycle ends, s'_j is marked as a predecessor of s_k. Therefore, during the backtrack phase that passes through s_k, the entire cycle will be found. Consequently, the returned subgraph contains p.

To see why G' is minimal, observe that if a state is added to the subgraph it is part of at least one path from s_{init} to s_{end}, and therefore cannot be omitted from the graph.

Lemma 3. *If the algorithm returns a patched program P', then $P' \vDash \Phi$.*

Proof. Suppose that there exists a run ρ of P' violating Φ. Denote its states $s_1, \ldots, s_n$, and extract from them a violating run with no cycles. If $s_i = s_j$ for some $j > i$, delete states $s_{i+1}, \ldots, s_j$. Denote the remaining states as $s_{t_1}, \ldots, s_{t_k}$. The run corresponding to this state sequence was found by the model checker, and a patch for some subgraph G' which contains this run was created. Since G' contains all paths from s_1 to s_n, it also contains ρ. Therefore, the patch would have blocked the last program-requested event of ρ, causing a contradiction. $\square$

As with the linear case, it is possible for the algorithm to return a failure notice. The Patchability Lemma, which characterized programs that could be fixed in the linear case, holds for the cyclic case as well; its proof is analogous.

The complexity of the algorithm is as follows: The exploration of violating runs costs, as before, $O(T_{mc} + \Upsilon \cdot D^2)$. Constructing the relevant subgraph for each violating run costs another $|V_G| + |E_G|$ times Υ runs, yielding:

$$T = O\left(T_{mc} + \Upsilon \cdot D^2 + \Upsilon(|V_G| + |E_G|)\right).$$

Again, this shows our dependence on the number of violating runs, Υ. The smaller that number, the closer our complexity is to that of the model-checker; the higher it is, the closer we are to the notorious, worst-case complexity of the synthesis problem.

7.2 Subgraph Representation

The generated code for a linear safety patch contains only the list of events to be waited for, followed by the event to be blocked. This list can be readily understood and possibly manipulated by a human, say, for documentation or analysis. Further, the developer may simplify or generalize the patch; e.g., skip waiting for certain guaranteed events or consolidate patches into fewer "symbolic" one, using BPJ's event filters. However, when a patch traverses a complex subgraph, gaining such insights is harder. Thus, we propose to represent the subgraph as a collection of easily readable linear event scenarios, amenable to human manipulation. The operation of the cyclic safety patch will be as before.

Specifically, We use the term *line* for a finite sequence of events that occur along some contiguous path in the state graph, and along which no state is visited twice. We use the term *tail* for a line whose last event would lead to a bad state in the state graph. The program's state graph, or parts thereof, are stored as a collection of lines, each containing its sequence of events, and links to other lines that are reachable by a single event from the last event in the line. See Fig. 7.

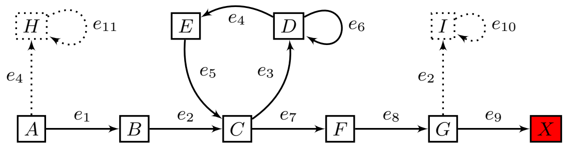


Fig. 7. A state graph of a buggy program. The model-checker returns the violating run with events e_1, e_2, e_7, e_8, e_9. The subgraph of all paths from state A to state G (see solid states and edges) is decomposed into: $line_1 = e_1, e_2$ (successors $tail, line_2$); $line_2 = e_3$ (successors $line_3$, $line_4$); The self-loop $line_3 = e_6$ (successors $line3, line_4$); $line_4 = e_4, e_5$ (successors $line_2$,$tail$) ; $tail = e_7, e_8$ (with event to be blocked, e_9). In addition to the run found by the model checker, the patch prevents other runs, e.g., $e_1, e_2, \underline{e_3, e_6, e_6, e_4, e_5, e_3, e_4, e_5}, e_7, e_8, e_9$, where the underlined events correspond to cycles.

Thus, each patch,

– begins by activating lines containing the initial state;
– waits for all events and traverses active lines;
– deactivates active lines when they are deviated from;
– deactivates a line and activates its successors when the line's last event occurs;
– in a tail, prior to the event leading to the bad state, blocks that event, waits for one more event, and deactivates the tail.

The line representation can be implemented in a data structure or in separate patch b-threads, each beginning with waiting for a unique activation event. This results in a number of small patches and is readily implementable in all implementations of behavioral programming.

7.3 Example: Patching a Coffee Machine

We demonstrate cyclic safety patching with a simple coffee vending machine example, which is expected to repeatedly wait for a coin, wait for a coffee request, and prepare the coffee. The main requirement is that coffee is never prepared unless a coin is first inserted. However, if immediately after power-up the user requests coffee, the machine incorrectly allows coffee to be requested and prepared

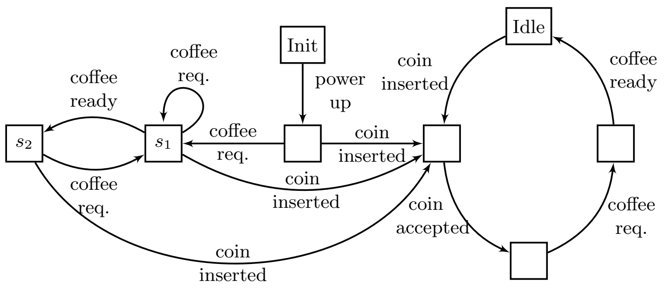


Fig. 8. The buggy coffee machine's state graph. After the `PowerUp` event, if a `CoffeeRequested` event occurs (before a coin is inserted), free coffee can be obtained infinitely many times, until a coin is inserted. The loop on the right-hand side of the graph represents the desired operation. The problematic state (marked s_1) has two enabled events: `CoffeeReady`, which is immediately requested (and selected), and the environment event `CoffeeRequested`. We expect the patch to block the `CoffeeReady` event.

infinitely many times without a coin. When the first coin is inserted, the machine enters normal operation. The machine's state graph is depicted in Fig. 8.

Observe that the bug is a safety bug — coffee is served without first inserting a coin. When it is discovered and automatic patching is attempted, the first step is to have a new b-thread identify and mark bad states (namely, s_2).

The automatic patching algorithm generates a single patch, corresponding to the subgraph depicted in Fig. 9.

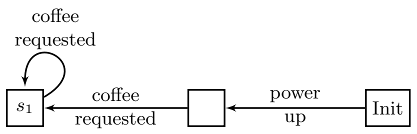


Fig. 9. The subgraph of the program's state graph for which a patch is created. It shows all paths from the graph's initial state to state s_1, in which event `CoffeeReady` must be blocked to prevent violations.

Finally, the graph of the patched program is depicted in Fig. 10, and the code generated by the proof-of-concept tool is shown in Fig. 11.

8 Dealing with Liveness

Up to this point, we dealt with safety properties — those that assert that "nothing bad happens". Another important class of properties is those involving liveness, asserting that "good things eventually happen". In this section we show how wait-block patches can be applied in order to fix liveness violations too.

In the case of safety properties, ensuring that a property holds is reducible to rendering all "bad" states unreachable, and so it was straightforward to use blocking in order to correct malfunctioning programs. Recall that in Section 4

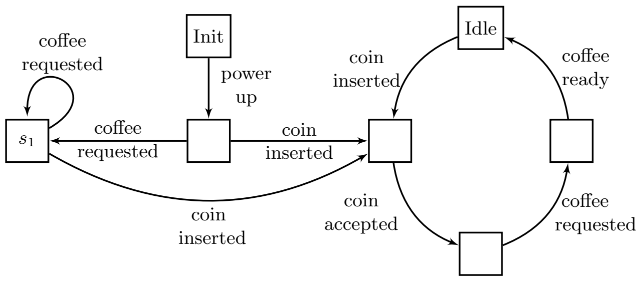


Fig. 10. The patched program's state graph (states of the patches themselves are omitted for clarity). The violating `CoffeeReady` event has been blocked, and the bad state no longer exists in the state graph.

```
public cyclicPatch1() {
   line1Events.add( new PowerUp() );
   line1Events.add( new CoffeeRequested() );
   line1 = new LineComponent( line1Events );

   line2Events.add( new CoffeeRequested() );
   line2 = new LineComponent( line2Events );

   tailEvents.add( new CoffeeReady() );
   tail = new TailComponent( tailEvents );

   line1.addSuccessor( tail );
   line1.addSuccessor( line2 );
   line2.addSuccessor( line2 );
   line2.addSuccessor( tail );

   this.addActiveComponent( line1 );
}
```

Fig. 11. The automatically-generated Java code for representation of the subgraph in Fig. 9. The first line contains events `PowerUp` and `CoffeeRequested`, and the second line contains `CoffeeRequested`. The tail contains only the event to be blocked, `CoffeeReady`. The code is readily understandable.

we mentioned that liveness violations correspond to reachable cycles of "hot" states (i.e., "hot cycles") in the program's state graph, and so it is less clear how to apply blocking. One natural approach might be to identify when the system is traversing a hot cycle, and then block one of the cycle's transitions (when an alternative exists), forcing the run to leave the cycle. This has several drawbacks:

1. Unlike in the safety case, where a bad state was never to be visited, in the liveness case it is legal to traverse the hot cycle any finite number of times. Consequently, safety-like patching would destroy good runs, which is highly undesirable.
2. Naïvely forcing the run to leave a hot cycle does not guarantee that it reaches a cold state; it could enter another hot cycle.
3. We would need to keep track of the hot cycles in the graph — the number of which could be very large.

To overcome these difficulties, we adopt a different perspective. Instead of considering runs and the hot cycles they traverse, we consider the hot states themselves. We show how, using wait-block patches, one can enforce a state-based policy that forces every run to visit cold states infinitely often, thus ensuring that the liveness property in question holds.

Our technique works by distinguishing between two types of hot states: *hot-trap* states and *hot-escapable* ones. Hot-trap states have the property that once they are visited, a liveness violation cannot be prevented; i.e., the system can never force the run into a cold state again. Consequently, hot-trap states are considered as "bad" states, and we use safety wait-block patches to render them unreachable. The hot-escapable states are those from which the system could force the run to visit a cold state, via some transitions; however, we cannot assume that these transitions may ever be traversed. In particular, it is possible for the system to continuously choose transitions that keep the run in hot states, although transitions to cold states are always enabled. We handle the hot-escapable states by enforcing *fairness*: we make sure that if a transition is enabled infinitely often, it will eventually be traversed. This type of fairness can be enforced using *probabilistic wait-block patches*, which we also call *liveness patches*. Through their use we can ensure that any liveness violations are effectively eliminated.

In the remainder of the section we discuss the liveness patching process more thoroughly.

8.1 Classifying Hot States

The first step in our repair algorithm is partitioning the hot states in the state graph into the two types mentioned. These two sets are formally defined by the algorithm below, which takes as an input the state graph of the program $G = (V_G, E_G)$, and returns the sets of hot-escapable and hot-trap states. For each hot-escapable state the algorithm also outputs its *escape-distance*, denoted δ: this is the length of a path from the hot-escapable state that reaches a cold state, and which the system can enforce regardless of the environment's behavior. See Fig. 12 for an illustration.

```
Classify Hot States($V_G, E_G$):
1: $A \leftarrow ColdStates(V_G)$, $B \leftarrow HotStates(V_G)$, $iteration \leftarrow 1$
2: $continue \leftarrow true$
3: while $continue$ do
4:     $continue \leftarrow false$, $New \leftarrow \phi$
5:     for each state $s \in B$ do
6:         if at least one outgoing edge (internal or external) from $s$ leads to a state in
           $A$, and no outgoing external edge from $s$ leads to a state in $B$ then
7:             $continue \leftarrow true$
8:             $New \leftarrow New \cup \{s\}$
9:             $\delta(s) \leftarrow iteration$
10:        $iteration + +$
11:    $B \leftarrow B - New$, $A \leftarrow A \cup New$
12: $HotEscapable \leftarrow A \cap HotStates(V_G)$
13: $HotTrap \leftarrow B$
14: return $(HotEscapable, HotTrap)$
```

The algorithm performs a fixpoint computation of the set A of states that are either cold or from which the system can force the execution to reach a cold state. When the algorithm terminates, this set contains the hot-escapable states.

The key point in the algorithm is line 6, which contains the condition based on which a new hot state enters A. For a state to become hot-escapable, all its external edges must lead into A, which expresses the fact that these events are beyond our control, and are controlled by the environment. Since we cannot prevent (block) them, we require that they cause no problem in the first place — namely, that they lead to states that have already been classified as hot-escapable by their being in the set A. Another condition, which handles the case where a state only has internal events enabled, is that there be at least one edge going into a state of A. The key fact is that if either condition holds, the blocking idiom can be applied to block all edges that do not lead to A.

The *escape-distance* value, δ, of a hot-escapable state indicates the number of the iteration in which it joined A. It measures the shortest guaranteeable distance to a cold state — that is, the length of the shortest such path that can be enforced by blocking.

Observe that while this algorithm serves to define hot-escapable and hot-trap states, it is not efficient — primarily because of the loop in line 5. By considering, at every iteration, only nodes that have successors that were determined hot-escapable in the previous iteration, the run time complexity can be reduced to $O(|V_G| + |E_G|)$.

8.2 Handling Hot-Trap States

As discussed previously, once the run enters a hot-trap state the system cannot guarantee that it ever reaches a cold state. Consequently, we are forced to

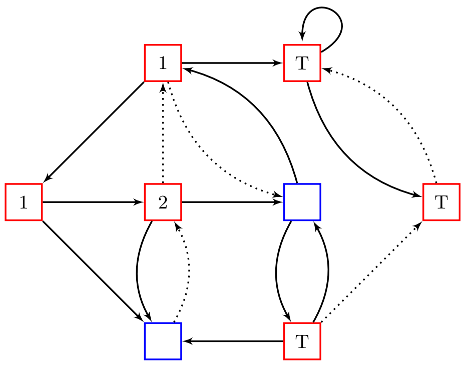


Fig. 12. Hot-trap and hot-escapable states. Hot states are marked red and contain either a number or the letter T; cold states are marked blue and are empty. Solid edges correspond to internal events, and dotted edges correspond to external events. A number inside a hot state designates the state as hot-escapable and indicates the escape-distance. The letter T designates the state as hot-trap.

block that entrance in the first place. This is done by applying safety wait-block patches, using the technique discussed in Section 7, which renders all hot-trap states unreachable.

Observe that this may remove potentially good runs too — namely, runs that go through a hot trap state yet still visit a cold state eventually. This can happen, for example, when an external event leading to another hot trap state is not triggered and, instead, an internal event that leads to a cold state is triggered. However, since we cannot depend on external events being triggered or not, our only way to ensure that no violations occur is to make hot-trap states unreachable.

8.3 Hot-Escapable States and Transition Fairness

The criterion used in determining the set of hot-escapable states ensures that careful use of the blocking idiom can force the run from a hot-escapable state into a cold state. The actual technique we propose is aimed at harming as few good runs as possible, and is based on *fairness*.

The notion of *fairness assumptions* [18] is used widely in formal verification, typically in order to rule out violating runs of the system because they are not realistic. Here, we discuss a special kind of fairness, called *transition fairness* [1]: if a transition is enabled infinitely often (i.e., its state of origin is visited infinitely often), then it is traversed infinitely often. We also allow a set of transitions originating from the same state to form a single constraint: if the state is visited infinitely often, then at least one of the transitions in the set is traversed infinitely often. Note that, unlike in the traditional setting where fairness is *assumed* for verification purposes, here we aim to *enforce* it within a malfunctioning system.

Intuitively, hot-escapable states have the property that if the event selection mechanism were to choose the triggered events uniformly at random, a run that visits them would eventually lead to cold states. It turns out that one can also settle for assumptions that are weaker than truly random event selection. We express these required assumptions as transition fairness constraints, and then discuss how to enforce them. Formally:

Definition 14. *Let P be a behavioral program with state graph G. A* transition fairness constraint *c on G is a set of one or more transitions (edges) in the graph, $\{e_1, \ldots, e_n\}$, all originating from the same node v. We say that P* satisfies *c, denoted $P \vDash c$, if it has the following property: if a run ρ of P visits v infinitely often, transitions from c are traversed infinitely often.*

Let $C = \{c_1, c_2, \ldots, c_k\}$ be a set of transition fairness constraints. We say that P satisfies *C, denoted $P \vDash C$, if $\forall_{1 \leq i \leq k} P \vDash c_i$.*

We now define a set of specific transition constraints for each of the hot-escapable states in the graph, and then show that they suffice for guaranteeing the liveness property in question.

Definition 15. *Let P be a behavioral program with state graph $G = (V_G, E_G)$, and let $V_{hot-escapable} \subseteq V_G$ be its set of hot-escapable states with respect to some liveness property Φ. For each $v \in V_{hot-escapable}$, the* transition fairness constraint *of v, $\tau(v)$, is defined as follows:*

- *if v has transitions corresponding to external events, $\tau(v)$ is the set of these transitions.*
- *otherwise, v has a neighbor, u, such that u is a cold state or $\delta(u) < \delta(v)$. In this case, we define $\tau(v)$ to be the edge leading from v to u.*

Observe that for every hot-escapable state v, $\tau(v)$ can be found during the hot state classification algorithm at no additional cost. We define the set of transition fairness constraints of the entire program to be the set of transitions fairness constraints on all its hot-escapable states, namely $\tau(P) = \bigcup_{v \in V_{hot-escapable}} \tau(v)$. The following proposition justifies our choice of constraints:

Lemma 4. *Let P be a behavioral program and let Φ be a liveness property. If P has no hot-trap states with respect to Φ and $P \vDash \tau(P)$, then $P \vDash \Phi$.*

Proof. Suppose, towards contradiction, that $P \nvDash \Phi$. Then there exists a run ρ of P and a hot state $v_0 \in V_{hot}$, such that v_0 appears infinitely often in ρ. Since P has no hot-trap states, v_0 is hot-escapable.

By our assumption that the constraints of $\tau(P)$ hold, there exists a neighbor of v_0, denoted v_1, that also appears infinitely often in ρ, and this v_1 is either a cold state or a hot-escapable state with $\delta(v_1) < \delta(v_0)$. If the former holds, then $\rho \vDash \Phi$ and we are done. If the latter holds, we reapply the same logic iteratively. Clearly, this produces a chain of hot-escapable states $v_0, v_1, \ldots, v_n$, all appearing infinitely often in ρ, with $\delta(v_0) > \delta(v_1) > \ldots > \delta(v_n)$. Since $\delta(v_0)$ is finite, this process ends in visiting a cold state infinitely often, again implying that $\rho \vDash \Phi$.

Note that the lemma assumes that P has no hot-trap states. However, this is not a real limitation, since, as previously explained, we can first apply safety patching to make such states unreachable.

8.4 Liveness Patches

We have characterized fairness constraints that are sufficient for correcting the liveness violation. Unfortunately, behavioral programs are not guaranteed to be fair. This is an intrinsic property of the event selection mechanisms commonly used in behavioral prgramming. For example, in arbitrary or priority-based selection certain transitions might be enabled infinitely often but never triggered. Consequently, we introduce a new type of patch, termed a *liveness wait-block patch*, aimed at enforcing a transition fairness constraint on the program.

Definition 16. *Given a state graph $G = (V_G, E_G)$, a probability η, a hot-escapable state $v \in V$ and its transition fairness constraint $\tau(v)$, a* liveness wait-block patch *for v is a b-thread with the following properties:*

- *It waits for all events chosen by the event selection mechanism and traverses the graph G according to them.*
- *It keeps track of the present state and notes when the execution reaches v.*
- *Whenever in v, with probability $1-\eta$ the patch does nothing. With probability η, it blocks all transitions except those in $\tau(v)$.*
- *It does not request events and does not label states.*

Intuitively, liveness wait-block patches are a way of incrementally injecting fairness into specific states of an already existing program, without modifying existing code. When the patch is applied to a hot-escapable state, it enforces the fairness constraint of that state; in runs in which the state is visited infinitely often, at least one of the transitions specified by the constraint will be triggered infinitely often. Indeed, the probability that edges in $\tau(v)$ are not traversed after after m visits to v approaches 0 as m tends to infinity, and this is true even for very small values of η. Note that, despite their probabilistic nature, these are essentially wait-block patches: they wait for a sequence of events, and then apply blocking to steer the run in the right direction.

Observe that it is indeed always possible to block all the transitions except those in $\tau(v)$. The only events that cannot be blocked are the external ones; and if there are external transitions in v, they are all in $\tau(v)$ by definition. Further, observe that by their definition liveness patches cannot cause deadlocks in states that were deadlock-free before the patching — as the patch always leaves unblocked at least one event that was already enabled.

Our motivation for using probability-based blocking is the desire to leave good runs unaffected. Choosing η to be small still guarantees that the fairness constraint holds, but makes it likely that runs that scarcely visit the state remain unaffected.

As in the case of cyclic safety patches (Section 7.2), liveness patches can be represented as a collection of *lines* and *tails* to make them more comprehensible.

We point out that so far we have dealt strictly with deterministic behavioral programs. Our probabilistic liveness patches, however, introduce nondeterminism into the system. This nondeterminism is not "against the grain" of behavioral programming, and indeed, extending behavioral programming definitions to support nondeterminism is straightforward, and is omitted.

8.5 The Liveness Patching Algorithm

Based on the discussion in the previous sections, we now present the patching algorithm itself:

```
Liveness Patching($P, \Phi$):
 1: $P' \leftarrow P$
 2: Run the model checker on $(P, \Phi)$
 3: if $P \models \Phi$ then
 4:    return $P$
 5: Run algorithm Classify Hot States on the state graph
 6: for each hot state $v_h \in V$ do
 7:    if $v_h$ is a hot-trap then
 8:       if creating a safety patch to prevent runs from reaching $v_h$ is impossible then
 9:          return Failure
10:       Create a safety patch $p^s_{v_h}$ that prevents runs from reaching $v_h$
11:       $P' \leftarrow P' \cup \{p^s_{v_h}\}$
12:    else
13:       Create a liveness patch $p^\ell_{v_h}$ for $v_h$
14:       $P' \leftarrow P' \cup \{p^\ell_{v_h}\}$
15: return $P'$
```

Observe that, as in the safety patching algorithm of Section 7, it may be impossible to create safety patches for hot-trap states in certain cases. One extreme example is when the entire state graph consists of hot-trap states only, so that attempting to render these states unreachable produces a trivial program that deadlocks in its initial state. In such cases, the algorithm returns a failure notice.

The correctness of the algorithm is established by the following lemma:

Lemma 5. *Let P' be a patched program returned by the algorithm, and let ρ be a run of P'. Then with probability* 1, *$\rho \models \Phi$.*

Proof. By the previously proved correctness of safety patching (Lemma 3), the algorithm ensures that there are no reachable hot-trap states in P'. By Lemma 4, it suffices to show that P' satisfies the constraints in $\tau(P)$ with probability 1. This claim is immediately derived from the definition of a liveness wait-block patch (Definition 16) and the discussion following it.

Part of our motivation for using wait-block patches in repairing violated safety properties was the Locality Lemma, which stated that any good runs remain unchanged. Unfortunately, that lemma cannot be proved for the liveness case; in fact, any liveness wait-block patch, by definition, might affect good runs as well as bad ones. We settle for the following:

Lemma 6 (The Weak Locality Lemma (Liveness)). *Let P be a collection of b-threads, let p be a liveness wait-block patch for hot-escapable state s_h, and let P' denote the patched program $P \cup \{p\}$. Any run ρ of P that* does not *reach s_h constitutes a valid run ρ' of P', and $APtrace(\rho) = APtrace(\rho')$.*

The proof is similar to that of the safety case and is omitted. The result is weaker, in the sense that if s_h is hot-escapable then good runs that pass through it might, with some probability, become invalid in the patched program. That probability increases the more times they pass through s_h. However, the effect on good runs can be reduced by decreasing the patches' blocking probability η.

The complexity of the liveness patching algorithm is as follows. The model checking phase costs $O(T_{mc})$. Classifying the hot states is linear in the size of the state graph. Each hot-trap state is then handled as a safety violation, adding $O(|V_{hot-trap}| \cdot (D^2 + |V_G| + |E_G|))$. Finally, for every hot-escapable state, we must construct the sub-graph needed to check when it is visited, yielding another $O(|V_{hot-escapable}| \cdot (|V_G| + |E_G|))$. Combining these, the total worst-case complexity becomes:

$$T = O\left(T_{mc} + (|V_{hot}| + 1) \cdot (|V_G| + |E_G|) + |V_{hot-trap}| \cdot D^2\right).$$

The runtime complexity shows the algorithm's dependence on the number of hot states: the smaller it is, the closer our complexity is to that of regular model-checking. In the next section we discuss a heuristic-based approach for reducing this in practice.

8.6 Minimal Fairness Enforcement

In the algorithm just discussed, we first rendered all the hot-trap states in the state graph unreachable and then enforced a set of transition fairness constraints in the hot-escapable states. By Lemma 5, we are guaranteed that this repairs any liveness violations in the program. However, the number of enforced fairness constraints is rather large — it is approximately the number of hot-escapable states in the program. Despite this, in some cases one can settle for fairness constraints that are far less extensive. For instance, consider the graph in Fig. 13.

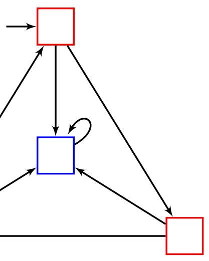


Fig. 13. An instance where the liveness repair algorithm would enforce more fairness than is required. The graph above has three hot-escapable states, and the algorithm would enforce transition fairness on the three edges leading from them into the cold state. Clearly, it suffices to settle for just one of these three constraints in order to guarantee that the cold state is eventually reached. Similar, larger constructions show that our algorithm might enforce any number of fairness constraints where just one would suffice.

Decreasing the number of fairness constraints being enforced is highly desirable, for two reasons. First, as we mentioned earlier, we wish to perform as few modifications to the original program as possible, and enforcing fewer constraints clearly serves this goal. Second, the size of the automatically generated code module is in correlation with the number of constraints that it enforces. Hence, fewer constraints means shorter modules, which are easier to maintain.

A natural question thus arises: can one identify a minimal-size set of fairness constraints that need be enforced on a given behavioral program in order to ensure that a given liveness property holds? Formally, we define the minimal fairness problem MF_{opt}, as follows: Given a behavioral program P with state graph $G = (V_G, E_G)$ and a liveness property Φ, such that G has no hot-trap states with respect to Φ, find a minimal-size set C of transition fairness constraints such that $P \vDash C \Rightarrow P \vDash \Phi$.

Unfortunately, it turns out that this problem is NP-complete. In [5], the authors study the problem of synthesis in the face of incomplete knowledge about the system's environment. In particular, they show that the problem of finding a minimal fairness assumption on the environment in order to make a given specification realizable is NP-complete. It is straightforward to show that this problem is reducible to MF_{opt} and that MF_{opt} is in NP, rendering it NP-complete.

Given this fact, we propose a greedy algorithm for approximating MF_{opt} in practice. The algorithm starts with an empty constraint set, and adds new constraints iteratively, in a manner similar to the way algorithm *Classify Hot States* finds the set of hot-escapable states.

Throughout its iterations, the algorithm maintains a growing set of already "handled" hot-escapable states. This is the set of states for which enforcing the current set of fairness constraints guarantees that they partake in no liveness violations. In other words, a run that visits any of these states infinitely often will reach a cold state infinitely often too.

The set of handled states is increased in each iteration. There are two ways for a state v to become handled:

1. By direct fairness enforcement: this happens when the algorithm chooses to enforce a fairness constraint leading from v into the set of already handled states.
2. By indirect domination: if, due to previous fairness constraints, all of v's successors are already handled, then v itself can be immediately marked as handled.

At each iteration, the algorithm imposes one fairness constraint, meaning that precisely one vertex becomes handled through method 1. Our criteria in choosing this particular vertex is trying to maximize the number of states that will become handled through method 2. The actual choice is performed by looking at all the candidates, namely nodes that can become handled through the enforcement of a single constraint. Each candidate is then assigned a value, which is its number of hot-escapable predecessors that are not yet handled (observe that these predecessors are precisely the vertices with potential to become dominated by choosing this vertex). Finally, the highest valued candidate is selected, and the corresponding fairness constraint is enforced. Here is pseudo-code outline of the algorithm:

```
Approximate $MF_{opt}(V, E)$:
1: $A \leftarrow HotStates(V)$, $handled \leftarrow \phi$, $constraints \leftarrow \phi$
2: while $A \neq \phi$ do
3:     $candidates \leftarrow FindAllCandidates()$
4:     $max \leftarrow MaxValuedCandidate()$
5:     Add a constraint that handles $max$ to $constraints$
6:     Move $max$ to from $A$ to $handled$
7:     while there are nodes in $A$ dominated by $handled$ do
8:         Move dominated nodes from $A$ to $handled$
9: return $constraints$
```

The two subroutines, $FindAllCandidates$ and $MaxValuedCandidate$, are omitted. As with algorithm *Classify Hot States*, an efficient implementation of the algorithm and its subroutines runs in time that is linear in the size of the program's state graph.

8.7 Example: Liveness Patching for the Dining Philosophers

We implemented our liveness patching algorithm (including the greedy approximation algorithm) within our proof-of-concept tool. For evaluation, we used the dining philosophers problem [7]. A behavioral implementation thereof includes the events of a philosopher picking up and putting down a given fork, a b-thread for the behavior of each philosopher and a b-thread for each fork. Each philosopher's b-thread is subject to a strict event sequence: pick up one fork, pick up the other, put down one fork, put down the other. Each fork's b-thread waits for events that change its state, and blocks illegal events (e.g., a second picking up, or, a putting down by the "wrong" philosopher). In [9] we model-checked this problem and variations thereof for safety and liveness properties.

For our experiment, we used a variant where the first $n-1$ philosophers are left-handed and the last one is right handed, which prevents deadlocks. All events in the program are internal, and so no hot-trap states exist. Finally, the liveness property used was this: "Philosopher #1 eats infinitely often". The results are shown in Table 1.

Each fairness constraint is translated into the actual code that enforces it, using the same mechanism as for safety patches. Although there may be many patches (as the example demonstrates), each of them is fairly comprehensible. The possibly high number of patches was part of our motivation for using the greedy algorithm; coming up with better algorithms to further reduce this number remains a topic for future work.

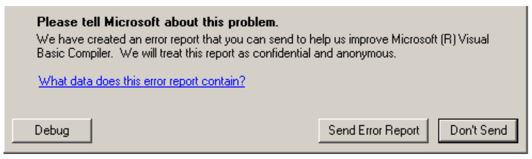
Table 1. Comparing the results of the naïve repair algorithm and the greedy approximation repair algorithm for the dining philosophers problem, with 9 - 12 philosophers. The *States* column shows the total number of states in the program. The *Patches (Naïve)* and *Patches (Greedy)* columns show the number of patches generated by the naïve and greedy algorithms, respectively. Observe that since the naïve algorithm generates one fairness constraint per hot-escapable state, the *Patches (Naïve)* column reflects the number of hot-escapable states as well. Finally, the *Reduction* column shows the percentage of patches saved by using the greedy version.

#Philosophers	**#States**	**#Patches (Naïve)**	**#Patches (Greedy)**	**Reduction**
9 Philosophers	19682	17495	9913	43%
10 Philosophers	59048	52487	30760	41%
11 Philosophers	177146	157463	93989	40%
12 Philosophers	531440	472391	287283	39%

9 Limited-Depth Repair

9.1 Automatic Repair from Field Error Reports

Many facilities exist for end-users to send reports of software failures to the software vendor (see, e.g., Fig. 14). Typically, these reports correspond to violated safety properties (e.g., "the system never crashes").


Please tell Microsoft about this problem.
We have created an error report that you can send to help us improve Microsoft (R) Visual Basic Compiler. We will treat this report as confidential and anonymous.

What data does this error report contain?

Fig. 14. Event logs from bug reports are used in patch construction.

For behavioral programs, we propose a methodology for using such failure reports in order to cope with the state-explosion problem inherent to model-checking, and to patch programs with many violating runs:

– The failure report contains an event log.
– Using the fact that the effect of a patch is local, we constrain the model checking depth to a neighborhood of the path of the failure (the bad run), followed by a limited fan-out of possible continuations, past the blocked transition.
– This is enforced by a dedicated b-thread, which monitors all events, and when an event occurs that is not along the reported bad path, it starts counting the distance from the bug report. When the distance is greater than a given parameter, the b-thread calls a model-checker API to prune the search.
– Finally, the safety patch is generated as above.

Such patching prevents the failure reported by the end-user, along with any other failures "not far" from it, and can help when full model-checking and patching consumes too much resources. The search-depth parameter is key, and needs to be adjusted per repaired program; higher depth means repairing more violations, but poorer performances. It is up to the user to use knowledge of the program's state graph, or run tests, in order to come up with the best choice.

9.2 Example: Limited-Depth repair of the Dining Philosophers

Again consider the dining philosophers problem [7], this time where all philosophers are left handed. The reported bug fixed is the classical deadlock where all philosophers pick up the fork on their left. Table 2 shows the results of patching for the single bad run that we gave the patcher.

Table 2. Patching the dining philosophers problem using bounded depth patching. Receiving a bug report (e.g., each philosopher picked up a single fork), the algorithm searches for event sequences that deviate from, or continue, the event trace in the bug report by no more events than the search depth parameter. The patches handle cycles discovered within the search depth (e.g., one of the philosophers completing a full cycle of picking up and putting down her two forks, while the others do not proceed). The tests were carried out on a PC with a Intel Quad Core Q6600 CPU @ 2.40GHz.

Search Depth	**3 Philosophers**	**6 Philosophers**	**9 Philosophers**
3	3 patches 3 loops 0.5 seconds	1 patch 2 loops 4.2 seconds	1 patch 2 loops 30 seconds
4	15 patches 30 loops 1.2 seconds	2 patches 4 loops 22 seconds	3 patches 6 loops 4.5 minutes
5	20 patches 380 loops 3.2 seconds	12 patches 1200 loops 2 minutes	12 patches 2580 loops 45 minutes

10 Related Work

The research in [17,20,21] presents fault localization and automatic repair of programs, where a set of software components that are suspected to cause a fault is replaced by a set of synthesized components, such that the resulting system is guaranteed to meet the full specification. Automatic repair of concurrency bugs (e.g., accessed to shared memory), is presented in [16]. The detection mechanism uses bad runs associated with bug reports, and the analysis involves actual execution. The repair is manifested in modification to existing code. Genetic-programming-based repair of legacy C programs is demonstrated in [24]. The repair relies on changes to existing code in order to correct problems that were assumed to be local in nature. In [2], genetic-programming is combined with co-evolution of the test cases against which the program is evaluated. Naturally, any work on automatic-repair would be considered a particular case of program synthesis [4,19].

As mentioned in Section 8.6, our work on repairing liveness violations relates to that of [5], where the authors show how to synthesize fairness assumptions the environment must uphold in order for a specification to be realizable. Our work tackles similar difficulties, but in a different setting: the environment is fixed, and fairness constraints on the system are synthesized. Our repair algorithm then imposes those constraints on the previously designed system.

As for other approaches for coordinating simultaneous behaviors, such as Esterel, BIP or Linda (see related work in [11, 12] for a comparison of behavioral programming with these approaches), we believe that comparable localized repair mechanisms would be possible. The key would be implementing the equivalent of blocking which, combined with ability to subscribe to all events, is central to our solution. This, of course, is possible, as it was in Java and Erlang, and could also benefit other aspects of incremental development in these environments.

11 Conclusion and Next Steps

The contribution of the present paper is in the proposed automated approach, in which faulty components are neither identified nor modified. Instead, the system is non-intrusively augmented with additional components, to yield desired overall system behaviors. The entire approach is made possible by the incrementality and modularity of behavioral programs. The new components are readily understandable by humans, and can be documented, enhanced, or generalized as part of standard development. The generated patches can then be distributed to users without re-distributing the original software. Finally, contributing to the on-going and up-hill battle with state explosion, we propose a methodology and a practical technique for constructing local patches using limited-depth model-checking.

This research is a step in the direction of developing methodologies and tools for the repair of behavioral programs. An important next step is to enrich the tool with interactive capabilities, allowing the developer to examine the state graph and enhance the proposed repairs: consolidating similar patches, generalizing or constraining patch functionality, or perhaps changing existing code after all.

Future research problems include repairing the program with regard to time-related properties, as well as integration with other formal methods tools and techniques, including other synthesis algorithms, symbolic model-checking, and compositional verification. Our tool could be combined with Java Pathfinder [23] or other tools to explore support of richer inter-process communication beyond solely behavioral events, and possibly solving concurrency problems among b-threads, as in [16].

We hope that with further developments in incremental, non-intrusive development, supported by powerful repair automation, the task of software maintenance may eventually shed its present (often lackluster) image, becoming a rewarding undertaking, allowing software engineers to quickly address customer needs in a productive, satisfying manner.

Acknowledgments

We thank A. Kantor, S. Maoz, Y. Sa'ar, S. Szekely and G. Wiener for their valuable suggestions on the manuscript. The research of D. Harel, G. Katz and A. Marron was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013), and by the Israel Science Foundation. The research of G. Weiss was supported by the Lynn and William Frankel Center for CS at Ben-Gurion University, by a reintegration (IRG) grant under the European Community's FP7 Programme, and by the Israel Science Foundation.

References

1. B. Aminof, T. Ball, and O. Kupferman. Reasoning about Systems with Transition Fairness. In *Proc. 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 194–208, 2004.
2. 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.
3. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
4. R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Saar. Synthesis of Reactive(1) Designs. *Journal of Computer and System Sciences*. In press.
5. K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Environment Assumptions for Synthesis. In *19th Int. Conf. on Concurrency Theory (CONCUR)*, pages 147–161, 2008.
6. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
7. E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Inf.*, 1:115–138, 1971.
8. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 378–398, 2002.
9. D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. In *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, pages 279–288, 2011.
10. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
11. D. Harel, A. Marron, and G. Weiss. Programming Coordinated Scenarios in Java. In *Proc. 24th European Conf. on Object-Oriented Programming (ECOOP)*, pages 250–274, 2010.
12. D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90–100, 2012.
13. D. Harel, A. Marron, G. Weiss, and G. Wiener. Behavioral Programming, Decentralized Control, and Multiple Time Scales. In *Proc. of the SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, pages 171–182, 2011.

14. D. Harel and A. Pnueli. *On the Development of Reactive Systems*, volume F-13 of *NATO ASI Series*. Springer-Verlag, New York, 1985.
15. D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 485–499, 2007.
16. G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated Atomicity-Violation Fixing. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011.
17. 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.
18. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: Specification.* Springer-Verlag, 1992.
19. A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. 16th ACM Symposium Principles of Programming Languages (POPL)*, pages 179–190, 1989.
20. S. Staber, B. Jobstmann, and R. Bloem. Diagnosis is Repair. In *Proc. 16th Int. Workshop on Principles of Diagnosis*, pages 169–174, 2005.
21. S. Staber, B. Jobstmann, and R. Bloem. Finding and Fixing Faults. *Correct Hardware Design and Verification Methods*, 3275:35–49, 2005.
22. A. Valmari. The State Explosion Problem. *Lectures on Petri Nets I: Basic Models*, Reisig, W. & Rozenberg, G. (eds.), Lecture Notes in Computer Science, 1491:429–528, Springer-Verlag, 1998.
23. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10:203–232, 2003.
24. W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic Program Repair with Evolutionary Computation. *Communications of the ACM*, 53:109–116, 2010.
25. G. Wiener, G. Weiss, and A. Marron. Coordinating and Visualizing Independent Behaviors in Erlang. In *Proc. 9th ACM SIGPLAN Erlang Workshop*, 2010.