מכון ויצמן למדע
WEIZMANN INSTITUTE OF SCIENCE

Thesis for the degree
Master of Science

עבודת גמר (תזה) לתואר
מוסמך למדעים

Submitted to the Scientific Council of the
Weizmann Institute of Science
Rehovot, Israel

מוגשת למועצה המדעית של
מכון ויצמן למדע
רחובות, ישראל

By
Guy Katz

מאת
גיא כץ

תיקון לא-פולשני של תוכניות התנהגותיות

Non-Intrusive Repair of Behavioral Programs

Advisor:
Prof. David Harel

מנחה:
פרופסור דוד הראל

March 2012

אדר תשע"ב

Non-Intrusive Repair of Behavioral Programs

A thesis submitted for the degree of Master of Science to the Scientific Council of the

Weizmann Institute of Science

Advisor: Prof. David Harel

Guy Katz

March 13, 2012

Abstract

We show how, under certain conditions, programs written in the *behavioral programming* approach can be modified (e.g., as 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 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 to the program and transform the counterexamples produced by the model-checker into corrective modules. Our work is supported by a proof-of-concept tool, which creates understandable modules that can be further manually managed as part of ongoing incremental system development.

Acknowledgments

I would like to express my earnest gratitude to Prof. David Harel for his continuing guidance and support throughout this work. I warmly thank Dr. Assaf Marron and Dr. Gera Weiss for their contribution to this work, and their willingness to share with me their knowledge and experience.

Contents

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 be known only to a different person (usually, the original developer) who is unavailable, or have been simply forgotten. 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 press the *"Send to Software Vendor"* button. In such cases, the software engineer cannot discuss with the user the broader context of the problem, or possible generalizations thereof.

In this thesis we focus on (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 [10, 12] — an extension and generalization of scenario-based programming, which was introduced with the language of live sequence charts (*LSC*s) [4, 8], and is now implemented also in Java [12, 11] and Erlang [24, 13].

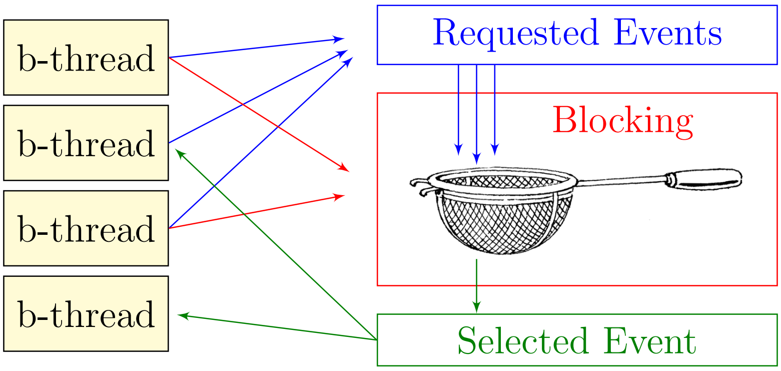

Figure 1: Behavioral programming execution cycle: all b-threads synchronize, declaring requested and blocked events; a requested event that is not blocked is selected and b-threads waiting for it are resumed.

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 Figure 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.

This facilitates incremental non-intrusive development as outlined in the example of Figure 2, adapted from [9].

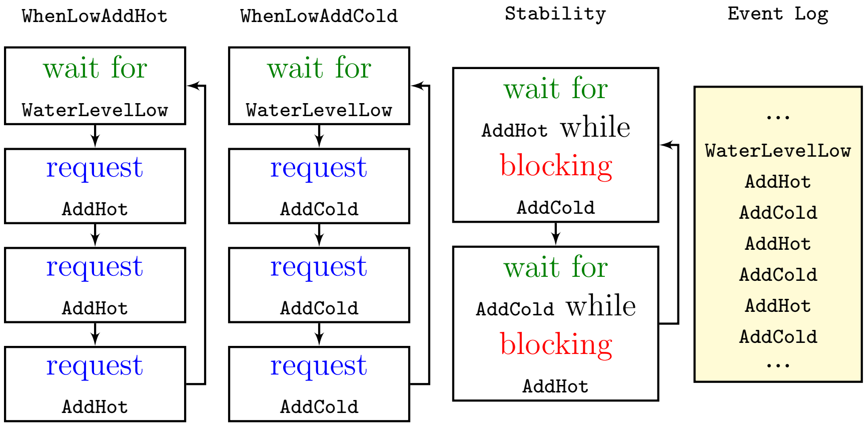

Figure 2: Incremental development of a system for controlling water level in a tank with hot and cold water sources. The b-thread `WhenLowAddHot` repeatedly waits for `WaterLevelLow` events and requests three times the event `AddHot`. `WhenLowAddCold` performs a similar action with the event `AddCold`, 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]. For details about model-checking such software with all possible environment behaviors, see [7].

More detailed examples showing the power of incremental modularity in behavioral programming appear in [12, 13]. Briefly, in a program for playing Tic-Tac-Toe [12], 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 the 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 [6] 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 [7], 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 [7] 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 this thesis we utilize the model checker of [7] 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. For each such run, we add a special b-thread, which 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. If this does not correct the problem, the process repeats.

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 full program repair may not always be possible, 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 thesis is organized as follows: basic definitions of behavioral programs and their model-checking are given in Sections 4 and in Section 5, respectively. The repair of loopless programs is discussed in Section 6, followed by a repair algorithm for general programs in Section 7. Finally, limited-depth patching is described in Section 8. Each of the three repair algorithms is followed by a concrete example.

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 [7, 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)* [2]. Recall that a *deterministic labeled transition system* is a 6-tuple $\langle S, E, \to, init, AP, L\rangle$, where S is a set of states, E is a set of events, $\to$ 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 $L : S \to 2^{AP}$ 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 transition function $\rightarrow$ is a total function.

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*

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

and

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

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 $f : S \to E$, 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 $(2^{AP})^\omega$). 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 $\varnothing^\omega$.

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.

We now narrow the definition of Φ, which can represent any LT property, to invariants and deadlocks.

Definition 7. *An LT Φ property over AP is an* invariant *if there is a propositional logic formula φ over AP such that $\Phi = \{A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \forall j \geq 0, A_j \vDash \varphi\}$.*

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:

Definition 8. *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.

Regular safety properties are those for which some finite automaton recognizes the bad prefixes [2], 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, 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.

5 Extending the Model-Checking of Invariants and Deadlocks

To check that a program P satisfies an invariant and does not cause a deadlock we follow the algorithm in [2], section 3.3.1, and the implementation in [7].

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 cycle), 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 all possible event selection mechanisms).

The runtime complexity of this algorithm, implemented as in [7], 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. For each $state \in V_G$, we have to perform $n \cdot e$ operations in order to find all its enabled events. In order to determine if a state was already visited earlier (also needed to detect cycles), assuming all states are stored in some hash table, we have to calculate a unique identifier for each state; this costs an additional $\log |V_G|$ operations, if we assume some optimal labeling of the states that only takes that many bits. Finally, another $|V_G| + |E_G|$ operations have to be performed to later traverse the graph. In total, we have:

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

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 [21, 2], 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 Figure 3).

The added complexity of this algorithm is measured using the number of violating runs, Υ (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 at most

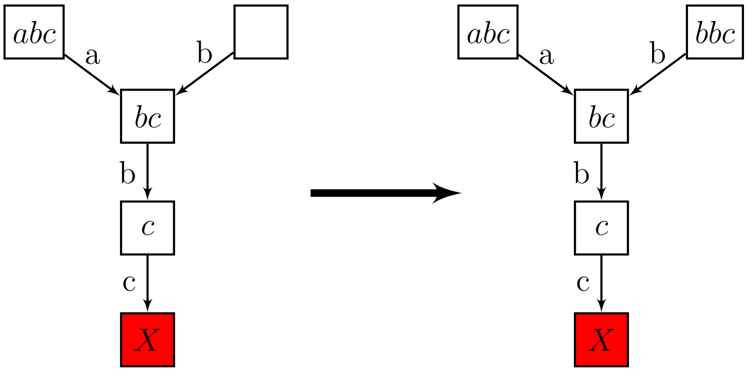

Figure 3: When a “bad” state is reached, all its predecessors store the relative path from that point to the violation. When a node in this path is reached through a different path, the data is propagated. The DFS continues until the root stores all violating paths.

$1 + 2 + \ldots + D = O(D^2)$ per violating run. The total runtime complexity is thus:

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

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 (see Figure 4). The runtime worst-case complexity remains unchanged.

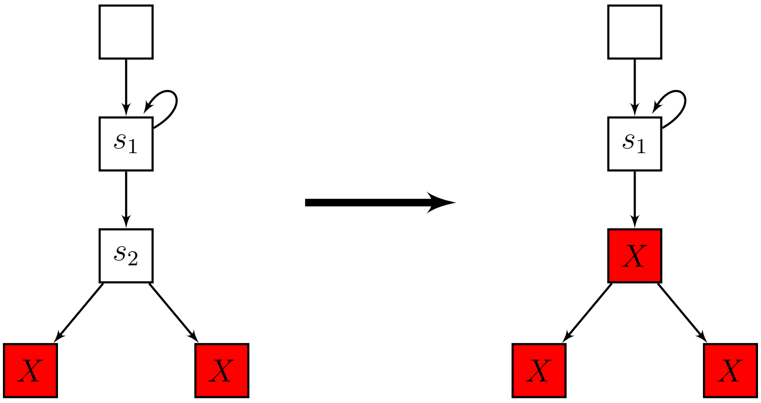

Figure 4: All of s_2's predecessors are marked as bad; as a result, the state itself is also marked as bad. If this were not the case, it would be turned into a deadlocked state. State s_1 has a self loop and is thus its own successor. Since not all its successors are bad, the propagation stops there.

6 Linear Patches

6.1 Generating Linear Patches

Before discussing 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, and as we mentioned earlier, in such cases we add a self-looped trap state and associate it with the accepting states of the transition system.

Definition 10. *A* linear 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 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. By definition, if a run does not start with events $(e_1, \ldots, e_n)$, then the patch never requests or blocks events. This means that the events chosen by P' are exactly those requested by P. If we simulate the run ρ on P' (i.e., have the environment choose the same events as it did with P), then the program will always request the same events as it did before. This means that events $\rho' = (e_1, \ldots, e_n)$ form a valid run of P'.

Finally, since the original b-threads will reach the same states as before, they will have the same atomic propositions. 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 requirement are found by model-checking, as follows:

Linear Patching(P, Φ):

```
  Run the model checker on $(P, \Phi)$
  **if** $P \models \Phi$ **then**
     **return** $P$
  P' ← P
  **for** each violating run $(e_1, \ldots, e_n)$ **do**
     **if** $\forall i, \;\; e_i \in E_{env}$ **then**
        **return** **Failure**
     **else**
        Find the largest $k$ such that $e_k \in E_{prog}$
        Create a linear wait-block patch $p$ for $(e_1, \ldots, e_k)$
        $P' \leftarrow P' \cup \{p\}$
  **return** $P'$
```

The idea is straightforward: the model-checker finds all runs violating Φ and we add a patch per run to prevent them. The algorithm guarantees that we create no deadlocks in doing so. 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 wait-block patches $p_1, \ldots, p_k$, such that $P \cup \{p_1, \ldots, p_k\} \vDash \Phi$.
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) ⇒ (2) is trivial.
- (2) ⇒ (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 reach a deadlock, all reachable states in the graph obtained in this way satisfy Φ and do not cause deadlocks. Furthermore, by definition of a wait-block patch, all blocked events were in E_{prog}, as needed.
- (3) ⇒ (1): Without loss of generality, we 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. In this case 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', proving the claim.

□

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 Figure 5.

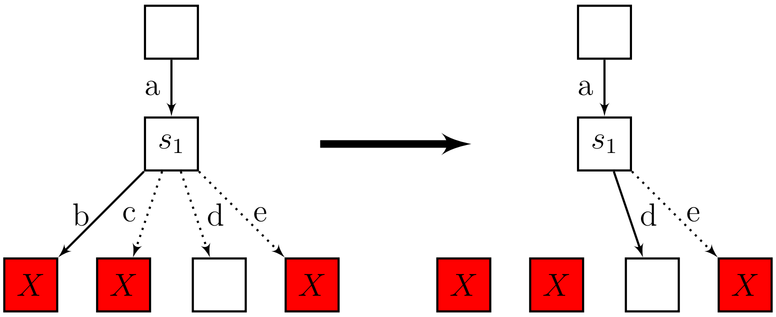

Figure 5: In state s_1, a patch that considers all event selection mechanisms will block b,c, and e. A patch that considers only, say, an ESM that chooses events alphabetically, needs to block b and c, but can leave e unblocked, relying on the selection of d.

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, discussed in the sequel.

6.3 Example: Patching Tic-Tac-Toe

We demonstrate the use of the linear patching algorithm on the loopless Tic-Tac-Toe behavioral program from [7]. 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 (never loses); 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 [7] and omitted the more complex threads — those that handle situations where an opponent creates, simultaneously, two ways to win, and thus cannot be blocked. If the human player does not try the complex strategy that creates 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 specification Φ in this case simply marks a victory of the human player as bad. The output is in the form of new files, each containing a new thread instance, which are easy to integrate into the original program. The patches themselves are easy to read and comprehend (see Figure 6).

```
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));
}
```

Figure 6: Example of a wait-block patch generated by the proof-of-concept tool. It waits for the moves X(2,2), O(1,1), X(0,0), and if they occur, it blocks a O(2,0) move. The code itself is fairly easy to comprehend; the more complicated details are hidden away in a parent class, making the auto-generated code legible and comprehensible.

Each such patch inherits from a parent class which implements its "main" function; see Figure 7.

```
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();
}
```

Figure 7: 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 Figure 6), blocking the last event and then terminating. If the events chosen deviate from those defined in the patch instance, the patch terminates.

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.

7 Patches for Programs with Cycles

7.1 Generating Patches for Cycles

The correctness of the algorithms for linear 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 a simple example, see Figure 8.

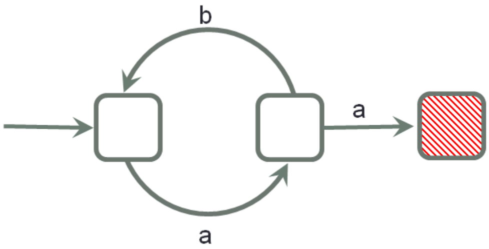

Figure 8: A simple state graph with a violating state and a cycle. Attempting to repair this program through linear patching would require infinitely many patches: a patch for aa, for $abaa$, for $ababaa$, and so on. In fact, any run of the form $(ab)^*aa$ would require its own patch. We must thus extend the definition of a linear patch into a patch more suitable for such programs.

Our solution is to extend the linear 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 11. *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 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 a 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, with the exception of 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 similar to that of the linear case, and 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 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 patching algorithm is as follows (G denotes the full state graph traversed by the model-checker):

```
Cyclic Patching($P, \Phi$):
  Run the model checker on $(P, \Phi)$
  if $P \vDash \Phi$ then
    return $P$
  for each violating run $(e_1, \dots, e_n)$ do
    if $\forall i, \;\; e_i \in E_{env}$ then
      return Failure
    else
      Find the largest $k$ such that $e_k \in E_{prog}$
      Let $s_{end}$ denote the state reached after events $e_1, \dots, e_{k-1}$
      Construct the minimal subgraph $G'$ containing all paths in $G$ from $s_{init}$ to $s_{end}$
      Create a cyclic wb-patch $p$ for $G'$ with states $v_{init} = s_{init}, v_{end} = s_{end}$, and event $e_k$
      $P' \leftarrow P' \cup \{p\}$
  return $P'$
```

Constructing the minimal subgraph G' is performed 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 BFS search from s_{init}. Instead of the usual BFS, where each vertex stores a single predecessor (the first vertex from which it is found), we have each vertex store *all* vertices from which it is found. When the search is done, 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 form the subgraph G' that we need.

Claim 1. *The graph G' is the minimal subgraph containing all paths from s_{init} to s_{end} in G.*

Proof. Let $p = (s_{init}, s_1, \dots, s_n, s_{end})$ be a path in G. 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, meaning that the graph 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. □

The algorithm's correctness is proved by the following lemma:

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. □

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 $O(|V_G| + |E_G|)$ times Υ runs, yielding a total of:

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

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 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" ones.

However, in the general case, a patch is associated with an entire sub-graph of the state graph, and not with just one list of events. This makes gaining such insights considerably 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 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 Figure 9.

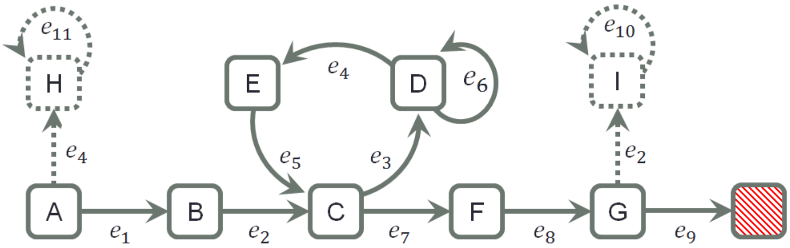

Figure 9: A state graph of a buggy program. The model-checker returns the violating run with events e_1, e_2, e_7, e_8, e_9. The subgraph of all paths from state A to state G (see solid states and edges) is decomposed into: $line_1 = e_1, e_2$ (successors $tail, line_2$); $line_2 = e_3$ (successors $line_3$, $line_4$); 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., $e_1, e_2, \underbrace{e_3, e_6, e_6, e_4, e_5, e_3, e_4, e_5}_{cycles}, e_7, e_8, e_9$.

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 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 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 Figure 10.

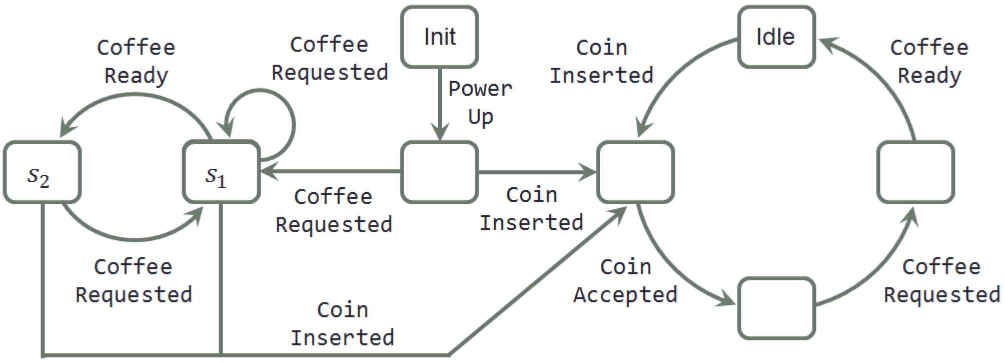

Figure 10: The buggy coffee machine's state graph. After the `PowerUp` event, if a `CoffeeRequested` event occurs (before a coin is inserted), free coffee can be obtained infinitely many times, until a coin is inserted. The loop on the right-hand side of the graph represents the desired operation. The 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.

When the bug 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 state graph of the coffee machine with that b-thread is depicted in Figure 11.

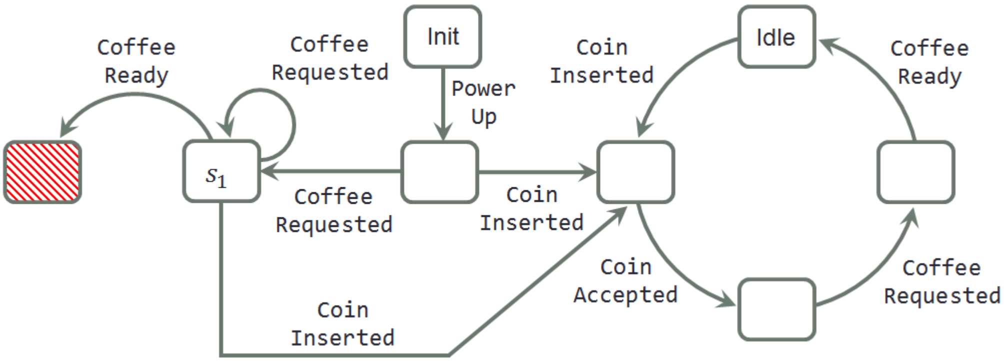

Figure 11: The coffee machine's state graph after the addition of a thread marking state s_2 as bad. The new thread's own states have been omitted for clarity.

The automatic patching algorithm generates a single patch, corresponding to the subgraph depicted in Figure 12.

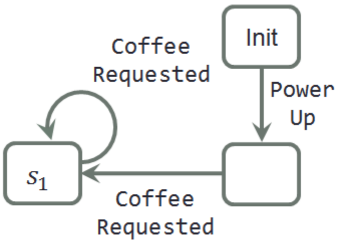

Figure 12: The subgraph of the program's state graph for which a patch is created. It shows all paths from the graph's initial state to state s_1, in which event `CoffeeReady` must be blocked to prevent violations.

Finally, the graph of the patched program is depicted in Figure 13, and the code generated by the proof-of-concept tool is shown in Figure 14.

8 Limited-Depth Repair

Due to our algorithms' heavy reliance on model-checking to come up with violating runs, they can only be applied to programs small enough to be model-checked. One solution might be using more effective model-checking techniques; we discuss this direction in Section 10. Another practical direction is to find another source that could provide us with information about the violating runs. A natural option, already implemented by large software vendors, is to allow users to send reports about buggy runs (for a known

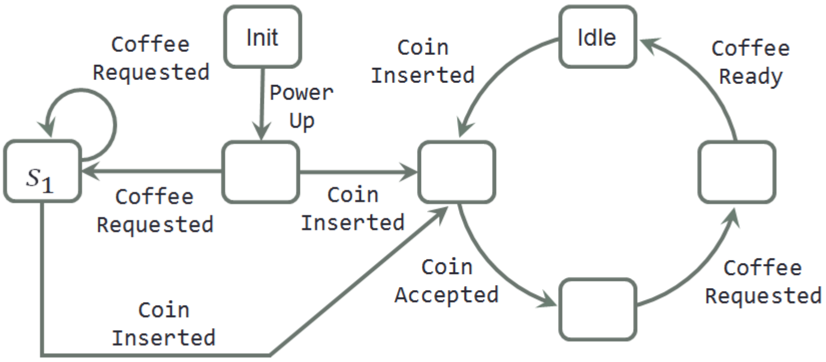

Figure 13: The patched program's state graph (states of the patches themselves are omitted for clarity). The violating `CoffeeReady` event has been blocked, and the bad state no longer exists in the state graph.

example, see Figure 15). For behavioral programs, these reports can partially replace the model-checker's role in the repair process.

8.1 Automatic Repair from Field Error Reports

We propose the following 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 patch is generated as before.

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. One extreme case is where it is set to 0; in which case, only the violating run received as input would be prevented, but the repair

```
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 );
}
```

Figure 14: The automatically-generated Java code for representation of the subgraph in Figure 12. 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.

Please tell Microsoft about this problem.

We have created an error report that you can send to help us improve Microsoft Internet Explorer. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Figure 15: Event logs from bug reports are used in patch construction.

process would be quick. The opposite extremity is setting the depth parameter to a large number, greater than the maximal depth of the state graph. In this case, the limited depth algorithm would yield the same results as the general algorithm, but would also have the same running time. Any interim value of the search depth yields a different trade-off between the number of violating runs prevented and the running time of the repair process.

It is up to the user to use knowledge of the specific program's state graph, or perhaps run tests, in order to come up with the largest depth for which it is still feasible to run the repair algorithm.

8.2 Example: Dining Philosophers

Consider the dining philosophers problem [5]. 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 nondeterministically, pick up the other, put down one fork nondeterministically, and then the other. Each fork's b-thread waits for events that change the state of the fork, and blocks illegal events (e.g., a second picking up, or, a putting down by the "wrong" philosopher). In [7] the authors model-check this problem and variations thereof for safety and liveness properties.

The behavioral implementation described above can easily reach a deadlock - say, by having each philosopher picking up the left fork. However, the large amount of nondeterminism makes it infeasible to apply our automatic repair algorithm for a large number of philosophers.

In order to simulate a user report about a faulty run, we encoded a specific violating run arbitrarily chosen into our proof-of-concept tool, and ran experiments with different search depth parameters and different number of philosophers. The table in Figure 16 shows the results of these experiments. The large increases in the number of loops found when the depth parameter increases illustrates the state-explosion problem as the state graph we work on grows closer to the full state graph.

Search Depth	3 Philosophers	6 Philosophers	9 Philosophers
3	3 patches 3 loops 0.5 seconds	1 patches 2 loops 4.2 seconds	1 patches 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

Figure 16: 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 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 eating 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.

9 Related Work

The research in [20, 19, 17] 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 [23]. The repair relies on changes to existing code in order to correct problems that were assumed to be local in nature. In [1], 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 [18, 3].

10 Conclusion and Next Steps

The contribution of this thesis 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 and liveness properties and 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 [22] 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.

References

[1] 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.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[3] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Saar. Synthesis of Reactive(1) Designs. *Journal of Computer and System Sciences.* In press.

[4] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[5] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Inf.*, 1:115–138, 1971.

[6] 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.

[7] 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.

[8] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer, 2003.

[9] D. Harel, A. Marron, A. Nissim, and G. Weiss. A Software Engineering Framework for Switched Fuzzy Systems. *IEEE Int. Conf. on Fuzzy Systems (FUZZ-IEEE)*, 2012. To Appear.

[10] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM.* To appear.

[11] D. Harel, A. Marron, and G. Weiss. The BPJ Library. www.cs.bgu.ac.il/ geraw.

[12] 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.

[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] 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.

[19] S. Staber, B. Jobstmann, and R. Bloem. Diagnosis is Repair. In *Proc. 16th Int. Workshop on Principles of Diagnosis*, pages 169–174, 2005.

[20] S. Staber, B. Jobstmann, and R. Bloem. Finding and Fixing Faults. *Correct Hardware Design and Verification Methods*, 3275:35–49, 2005.

[21] 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.

[22] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10:203–232, 2003.

[23] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic Program Repair with Evolutionary Computation. *Communications of the ACM*, 53:109–116, 2010.

[24] G. Wiener, G. Weiss, and A. Marron. Coordinating and Visualizing Independent Behaviors in Erlang. In *Proc. 9th ACM SIGPLAN Erlang Workshop*, 2010.