On Module-Based Abstraction and Repair of Behavioral Programs

Supplementary Material

Guy Katz

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

Appendix

I Two Equivalent Semantics for BP

In this section we prove an equivalence between the BP semantics used in this work, and that previously used by, e.g., [1]. This proposition allows us to use both semantics interchangeably — and consequently, results proven for one apply for the other.

Proposition 1. *Let $BT^1, \ldots, BT^n$ be a set of threads. Let $P = [BT^1 \parallel \ldots \parallel BT^n]$, and let P' be the behavioral program consisting of $BT^1, \ldots, BT^n$ as defined by [1]; then $\mathfrak{L}(P) = \mathfrak{L}(P')$. Further, an execution ϵ is a valid execution of P if and only if it is also a valid execution of P', and it has the same trace under both semantics.*

Both semantics use the b-threads in order to construct an LTS, the runs of which constitute the runs of the behavioral program. Thus, it suffices to show that both semantics generate the same LTS. For completeness, we bring the alternative set of definitions from [1]:

Alternative definition: Behavioral Threads. A *behavior thread* (abbr. *b-thread*) is a tuple $\langle Q, \Sigma, \rightarrow, init, AP, L, R, B\rangle$, where $\langle Q, \Sigma, \rightarrow, init, AP, L\rangle$ forms a total labeled transition system, $R \colon Q \rightarrow 2^{\Sigma}$ associates a state with the set of events *requested* by the b-thread when in it, and $B \colon Q \rightarrow 2^{\Sigma}$ associates a state with the set of events *blocked* by the b-thread when in it.

Alternative definition: Behavioral Programs. The *runs of a behavioral program* $\{\langle Q_i, \Sigma_i, \rightarrow_i, init_i, AP_i, L_i, R_i, B_i\rangle\}_{i=1}^{n}$ are the runs of the labeled transition system $\langle Q, \Sigma, \rightarrow, init, AP, L\rangle$, where $Q = Q_1 \times \ldots \times Q_n$, $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$, $init = \langle init_1, \ldots, init_n\rangle$, 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

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

and

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

The atomic propositions are $AP = \bigcup_{i=1}^{n} AP_i$ and, for $(q_1, \ldots, q_n) \in Q_1 \times \ldots \times Q_n$, the labeling function is: $L(s_1, \ldots, s_n) = L_1(s_1) \cup \ldots \cup L_n(s_n)$.

Begin by observing the events and atomic propositions. In our version, Σ and AP are traits of the program, and all threads use these global definitions. In the alternative version, these are properties of threads — but eventually, the events and atomic propositions of the resulting program are the union of those of its threads. Thus, we can assume that in the alternative definition all threads have $\Sigma_i = \Sigma$ and $AP_i = AP$ without loss of generality.

Next, we observe the state sets of the two transition systems. In our definition, each composition entails a cartesian product between the states of two threads — whereas in the alternative definition, the state set is the cartesian product of all threads in the program. Clearly, as $((Q_1 \times Q_2) \times Q_3 \ldots) \times Q_n = Q_1 \times Q_2 \times \ldots \times Q_n$, we get that both transition systems have the exact same state set. Using similar arguments, we get that in both transition systems the labeling function assigns the same labels to each state.

Finally, we only need show that the edges are the same. Using the alternative definition, an edge $\langle q_1, q_2, \ldots, q_n\rangle \xrightarrow{e} \langle q_1', q_2', \ldots, q_n'\rangle$ will exist in the LTS if and only if $\forall i, q_i \xrightarrow{e} q_i'$, and e is enabled (i.e. requested and not blocked). Using our definitions, the parallel composition operator guarantees that in state $\langle q_1, q_2, \ldots, q_n\rangle$ of the composed (and not yet finalized) thread $BT_1 \parallel \ldots \parallel BT_n$, event e will be requested (recall that using our definitions, a requested event cannot be blocked). Further, it is straightforward to prove inductively that the edge $\langle q_1, q_2, \ldots, q_n\rangle \xrightarrow{e} \langle q_1', q_2', \ldots, q_n'\rangle$ exists in the thread. Hence, it will survive the finalization operator and appear in the finalized LTS.

Having shown that the same LTS is produced using either semantics, Proposition 1 immediately follows. □

II Abstract Threads Yield Over-Approximations

This section is dedicated to proving Lemma 1, which reads:

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

Observe that, without loss of generality, we may assume that $n = 2$; otherwise, we would first calculate the composition $BT' = BT_2 \parallel \ldots \parallel BT^n$, and then deal with $P = [BT^1 \parallel BT']$.

In order to prove the lemma, we look at an execution ϵ of P, and prove that there exists an execution $\overline{\epsilon}$ of $\overline{P}$ such that $\text{Tr}(\epsilon) = \text{Tr}(\overline{\epsilon})$ — and hence, $\text{Tr}(P) \subseteq \text{Tr}(\overline{P})$.

Let $\epsilon = q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \ldots$ be an execution of P. Each state q_i is comprised of two components, q_i^1 and q_i^2, denoted $q_i = \langle q_i^1, q_i^2 \rangle$, such that q_i^j is a state of thread BT^j. We look at an execution $\overline{\epsilon} = \overline{q_0} \xrightarrow{e_1} \overline{q_1} \xrightarrow{e_2} \ldots$, with same events as ϵ. The states are set to $\overline{q_i} = \langle \eta_\pi(q_i^1), q_i^2 \rangle$, where η_π is the abstraction function mapping each state to its equivalence class under partition π. We next show that this $\overline{\epsilon}$ is a valid execution of $\overline{P}$, and that it has the same trace as ϵ.

By definition, every state $\overline{q_i}$ is indeed a state of $\overline{P}$. Further, $L(\overline{q_i}) = L(\eta_\pi(q_i^1)) \cup L(q_i^2) = L(q_i^1) \cup L(q_i^2) = L(q_i)$, and consequently $\mathrm{Tr}(\overline{\epsilon}) = \mathrm{Tr}(\epsilon)$. It only remains to prove that for each $\overline{q_i}$, the transition to $\overline{q_{i+1}}$ is legal — namely, that event e_{i+1} is enabled at state $\overline{q_i}$ and that the transition $\overline{q_i} \xrightarrow{e_{i+1}} \overline{q_{i+1}}$ exists in $\overline{P}$.

To see why event e_{i+1} is enabled, recall that by definition $\overline{R}(\eta_\pi(q_i^1)) \subseteq R(q_i^1)$. Hence:

$$e_{i+1} \in R(q_i^1) \cup R(q_i^2) \implies e_{i+1} \in \overline{R}(\eta_\pi(q_i^1)) \cup R(q_i^2)$$

And so, if e_{i+1} is enabled in state $\langle q_i^1, q_i^2 \rangle$ then it is also enabled in state $\langle \eta_\pi(q_i^1), q_i^2 \rangle$. Finally, by the abstraction's definition, the transition $q_i^1 \xrightarrow{e_{i+1}} q_{i+1}^1$ in BT^1 implies the transition $\eta_\pi(q_i^1) \xrightarrow{e_{i+1}} \eta_\pi(q_{i+1}^1)$ in $\overline{BT^1}$; and, in turn, the transition $\langle \eta_\pi(q_i^1), q_i^2 \rangle \xrightarrow{e_{i+1}} \langle \eta_\pi(q_{i+1}^1), q_{i+1}^2 \rangle$ in $\overline{P}$. Thus, $\overline{\epsilon}$ is a valid execution of $\overline{P}$; the claim follows. □

III Correctness of the *Check If Spurious* Algorithm

We prove Lemma 2, stating that the *Check If Spurious* is correct:

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

We show that the algorithm answers *False* if and only if the run is genuine.

First Direction: A Genuine Run. Suppose that $\overline{\epsilon} = \overline{q_0} \xrightarrow{e_1} \overline{q_1} \xrightarrow{e_2} \ldots \xrightarrow{e_n} \overline{q_n}$ is a genuine execution of $\overline{P}$; i.e., there exists an execution $\epsilon = q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} q_n$ of P, such that for every $q_i = \langle q_i^1, q_i^2, \ldots, q_i^m \rangle$ and $\overline{q_i} = \langle \overline{q_i}^1, \overline{q_i}^2, \ldots, \overline{q_i}^m \rangle$ it holds that $\eta_j(q_i^j) = \overline{q_i}^j$ for every j. Further, in every concrete state q_i (for $0 \leq i < n$), event e_{i+1} is enabled.

A straightforward inductive argument on $i = 1 \ldots, n$ shows that for the i'th step of ϵ, set S_i contains the concrete state $\langle q_i^1, q_i^2, \ldots, q_i^m \rangle$, and is thus non-empty. As this state requests and does not block the next event of the execution, it follows that $q_{i+1} \in S_{i+1}^j$. Hence, for all i we get $S_i \neq \emptyset$, which in turn implies that the algorithm returns *False*, as needed. □

Second Direction: Algorithm returns *False*. Suppose that on execution $\overline{\epsilon} = \overline{q_0} \xrightarrow{e_1} \overline{q_1} \xrightarrow{e_2} \ldots \xrightarrow{e_n} \overline{q_n}$, the algorithm answers *False*. We show that this implies the existence of a genuine run ϵ that corresponds to $\overline{\epsilon}$.

By the algorithm's answer, we know that the computed sets S_i are not empty for all $0 \leq i \leq n$ and. We use these sets, backtracking from $i = n$ to $i = 0$, reconstructing the genuine run as we go.

For $i = n$, we pick an arbitrary $q_n = \langle q_n^1, \ldots, q_n^m \rangle \in S_n$. Then, for state q_{n-1}, we pick a state $q \in S_{n-1}$ such that $e_n \in R(q)$ and $q_n \in \text{Post}(q, e_n)$; such a state exists by the way the S_i sets are defined. This process continues iteratively, until $\epsilon = q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} q_n$ is constructed. It is straightforward to see that it constitutes a valid run of P. The claim follows. $\square$

IV Correctness and Soundness of the Repair Algorithm

This section is dedicated to proving Theorem 1, which reads:

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

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

We begin with a side note about the meaning of a patch eliminating executions. As the patch is intended to be integrated into the program as a thread, it will change the program's underlying state graph. Hence, it is not immediate that executions of the original system have any meaning in the context of the patched program.

We resolve this issue by making the following observation. Due to the special structure of the patch — namely, that it follows the program's state graph and only blocks events, without requesting events or assigning atomic propositions — the program graph of the patched program is *isomorphic* to that of the original program, except for the edges being removed. Hence, any execution of the original program corresponds to a unique execution of the patched program, and it makes sense to discuss such executions being eliminated. For simplicity, for the rest of the proof we ignore this issue, regarding patches as eliminating transitions in the original state graph without modifying its states.

The theorem's proof relies mainly of the following invariant of the algorithm, which we prove as a separate proposition:

Proposition 2. *Let $\overline{q}$ denote an abstract state that the algorithm puts in set BAD. Then for any concrete state abstracted into $\overline{q}$, i.e. for every $q \in \eta^{-1}(\overline{q})$, any execution ϵ of P that visits q must violate Φ.*

Proof. We prove the proposition using induction on the algorithm's iteration index. Observe iteration i, the first iteration in which some state $\overline{q}$ is about to enter set BAD. At the beginning of this iteration, set BAD contains only the abstract state $\overline{q_b}$. Since $\overline{q}$ is about to enter set BAD, it must be that $\overline{q} \in PRE$. Further, the *NeedToRefine* subroutine returned *False* on $\overline{q}$ — meaning that any event that is not blocked in $\overline{q}$ leads to $\overline{q_b}$. If there existed a concrete state $q \in \eta_{i_1}^{-1}(\overline{q})$ and an event $e \in R(q)$ such that $Post(q, e) \neq \{q_b\}$, a matching transition would also appear in the abstract graph, and $\overline{q}$ would not be put in BAD. Hence, $Post(q) = \{q_b\}$. In other words, any concrete execution passing through any concrete state associated with $\overline{q}$ is bound to visit q_b and cause a violation.

Now, suppose that the claim holds for the first i iterations, and observe iteration $i+1$. Suppose a new state $\overline{q}$ joins BAD in this iteration. The reasoning is the same as before: $\overline{q}$ is put in BAD only if for every $q \in \eta^{-1}(\overline{q})$ and every event $e \in R(q)$, $q \xrightarrow{e} q'$ implies that $\eta(q') \in BAD$. By the inductive hypothesis, an execution that visits q' is thus bound to cause a violation. Since this applies to every successor of every concrete state $q \in \eta^{-1}(\overline{q})$, the claim follows. □

A second observation that we prove separately is that the algorithm always halts:

Proposition 3. *The* Abstract Safety Patching *algorithm always halts.*

Proof. Observe the algorithm's main loop. If the algorithm does not stop, it must make infinitely many iterations of this loop. Each iteration that does not lead to termination is devoted to either performing a single refinement of the abstract program, or to moving an abstract state into the growing set BAD. We show that both types of iterations can only be performed a finite number of times, proving the proposition.

Begin with iterations dedicated to refinement. Any refinement step splits an abstract state into at least two states; hence, each such step increases the number of states of the abstract program by at least one. Since this number is bound from above by the number of states of the original program, only a finite number of refinements can be performed. Once the abstract and concrete program coincide, the *NeedToRefine* subroutine will return *False* on every state, and the algorithm will cease attempting to refine the program.

We now turn to iterations in which states are moved to BAD. Observe the set of concrete states mapped to BAD in iteration i, denoted $\eta_i^{-1}(BAD)$. These sets start with $\eta_1^{-1}(BAD) = \{q_b\}$, and for each iteration i that puts a new state in BAD we have $|\eta_i^{-1}(BAD)| > |\eta_{i-1}^{-1}(BAD)|$. Since the size of $|\eta_i^{-1}(BAD)|$ is also upper bounded by the number of states in the concrete program, we get that the number of such iterations is also finite. We thus conclude that the algorithm always halts. □

We now use these propositions to prove part 1 of the theorem. Consider a patch BT_P produced by the repair algorithm. This patch eliminates transitions leading to all states in set BAD, effectively disconnecting them from the state graph. In particular, all executions leading to state $\overline{q_b}$ are eliminated. Since the existence of a concrete execution leading to q_b implies the existence of an abstract execution leading to $\overline{q_b}$, it follows that the patch indeed eliminates all bad executions in the concrete system.

Next, we show that no good executions are eliminated. All transitions that were removed from the state graph lead to states in BAD. By Proposition 2, any execution that visits these states is bound to cause a violation; hence, none of the affected executions are good.

Finally, we show that no deadlocks are created by the algorithm. A deadlock is created if and only if there exists a state in $q \in \eta^{-1}(PRE)$ for which the set of requested events, $R(q)$, coincides with the events to be blocked. Hence, when the state graph is finalized, state q would have no outgoing transitions.

Observe state $\overline{q} = \eta(q)$. This state is in PRE, and is not moved to BAD; hence, it has outgoing transitions that lead to good states. These transitions cannot originate in q; hence, there is another state, $q' \neq q$, such that $\eta(q') = \overline{q}$ and q' would not become deadlocked when the patch is applied. This contradicts the fact that $\overline{q} \in PRE$ at the time the algorithm halts, as subroutine *NeedToRefine* would return *True* for state $\overline{q} = \eta(q)$, leading to its being refined. This refinement would cause states q and q' to be mapped into separate abstract states; and in the algorithm's next iteration, the abstract state of q would be put in BAD. Hence, no deadlocks can occur as a result of patching, and the first part of the theorem is proven. □

We now turn to part 2 of the theorem. Here, we must show that the algorithm does not return a *Failure* when a correct patch exists. Suppose, then, that a correct patch BT_P exists. This patch corresponds to a set of transitions that are to be blocked, cutting off some of the concrete program's states. Also, this patch does not create deadlocks. We mark the set states to be cut off by S. Again we observe the series of sets $\eta_i^{-1}(BAD)$ that our algorithm grows through its iterations. By Proposition 2, for every i the set $\eta_i^{-1}(BAD)$ consists only of states that must lead to a violation of Φ. Since BT_P is correct, it follows that it, too, cannot allow executions to reach states in $\eta_i^{-1}(BAD)$. In other words, for every i we have $\eta_i^{-1}(BAD) \subseteq S$.

Our algorithm only issues a *Failure* notice if it reaches a state where the initial state of the concrete system, q_0, is in $\eta_i^{-1}(BAD)$. However, by the correctness of BT_P, set S cannot contain q_0, or else it would create deadlocks. Hence, our algorithm will not return a *Failure* notice. As Proposition 3 establishes that the algorithm must halt, we conclude that it will return some patch. Finally, by the first part of the theorem, this patch will be correct. We conclude that our algorithm will indeed output a correct patch if such a patch exists, as needed. □

References

1. D. Harel, G. Katz, A. Marron, and G. Weiss. Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 3–12, 2012.