Relaxing Synchronization Constraints in Behavioral Programs
Supplementary Material

David Harel, Amir Kantor, and Guy Katz

Dept. of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel
{`dharel,amir.kantor,guy.katz`}`@weizmann.ac.il`

Appendix

I The Distributed Execution Mechanism

The technique discussed in Section 4 requires that each b-thread communicate with the global coordinator at every synchronization point. While this constraint is significantly weaker than stepwise synchronization with all other b-threads, it may limit the applicability of the approach for designing multi-component applications in distributed architectures, in which communication is costly and time-consuming. In this section, we show how a variant of eager execution, combined with Dead Reckoning techniques [1, 2], can be utilized to reduce these costs. This variant is referred to as *distributed execution.*

In order to have behavioral modules executed in a decentralized manner on different machines, we distribute the coordinator, so that each machine runs its own *coordinator agent.* These agents serve as the coordinators for their *local* threads, i.e., threads running on the local machine, but have no direct access to threads on other machines. Instead, they can communicate with other agents.

Before running the system, each agent is given the state graphs of all the threads in the system, including non-local threads (similarly to the technique in Section 3.2). Each coordinator agent then executes the program locally, using these state graphs to simulate non-local threads and predict their synchronization requests. Each agent is responsible for answering its local threads' synchronization requests, just as a central coordinator would. Observe that this requires that the event selection mechanism be a deterministic function — that is, a function from $2^{\Sigma} \setminus \{\emptyset\}$ to Σ, whose input is the set of enabled events — in order to ensure that the autonomous agents pick the same events.

In a program with deterministic threads, this form of distribution would suffice to make inter-component communication obsolete, as each coordinator agent could trigger precisely the same events as the others. In the case of systems with nondeterministic threads (such as reactive systems), some communication between the distributed components is mandatory. Intuitively, this communication is used to announce the outcome of nondeterministic choices made by a thread to the other components. Specifically, all coordinator agents are aware of each thread's nondeterministic forks, as they hold all the state graphs. Whenever such a nondeterministic fork is reached, the coordinator agent on which that thread is actually running is responsible for disseminating the outcome of the nondeterministic choice to the remaining agents. If other agents reach this point before the outcome has been broadcasted, they must wait for it. This guarantees that the execution is consistent across all program components, in the following sense:

Lemma 1. *Let P be a behavioral program executed using the distributed execution mechanism. Then, all coordinator agents trigger the same sequence of events, and this sequence is a valid run (under BP's semantics).*

In the distributed execution mechanism, a thread waits for threads in other components only to resolve nondeterminism in the behavior of the latter. For the correctness of code executed in accordance with the triggering of an event, the programmer should generally assume that in other components this event is triggered at a different time (before or after).

In the next section we describe an example of a distributed application; and in section I.2 we formally define the model and prove Lemma 1.

I.1 Example: A Distributed Application

Suppose that communication to and from the vehicles in the example from Section 4 is costly, and is to be minimized. In particular, it is desirable to avoid a central coordinator. This could be addressed using a distributed design, where each vehicle and agent pair runs on a dedicated machine. As before, we assume that the vehicles travel in some pre-determined cyclic route. As the threads of this example are deterministic, each vehicle can completely predict the whereabouts of the other vehicles at any point in the execution, and collisions can be averted without any inter-vehicle communication.

We now introduce a source of nondeterminism. Suppose that one of the vehicles is an antique, and often requires maintenance. Along that vehicle's route there is a garage; and whenever the vehicle passes that point of the route, it may go in for repairs. The decision of whether or not to stop for repairs is considered a nondeterministic input from the environment. This new setting prevents other vehicles from predicting the location of the malfunctioning vehicle — since each lapse it may or may not spend one time unit in repairs.

Whenever the malfunctioning vehicle passes by the garage, the coordinator agents reach a nondeterministic fork in its state graph, and suspend their execution. As soon as the malfunctioning vehicle synchronizes and reveals whether or not the vehicle stopped for repairs, its handling agent disseminates the information to the other agents, allowing them to resume their execution.

Even in the nondeterministic setting, using the distributed version of the system significantly reduces the number of messages being sent between the machines. In the central coordinator scenario of Section 4, each vehicle would have to communicate with the coordinator for every single move; but in the distributed setting, only one message per round is sent from the malfunctioning vehicle to the others.

I.2 Distributed Execution Formalized

In this section we provide a rigorous definition of the model, and prove that the runs that it produces abide by the semantics of BP.

Let $P = \{BT^1, \ldots, BT^n\}$ be a (possibly nondeterministic) behavioral program, where $n \in \mathbb{N}$ and each BT^i is a distinct b-thread, and let $f : 2^{\Sigma} \setminus \{\emptyset\} \to \Sigma$ be a deterministic event selection function. Suppose that the threads run on different *machines* $M_1, \ldots, M_k$. Each machine is defined as the set of thread that it runs, i.e. $\bigcup_{i=1}^{k} M_i = P$.

Each machine M_i has a coordinator agent, C_i; this agent acts as the coordinator for the threads of M_i, and answers their synchronization requests. Each coordinator agent is supplied with the state graphs of all threads in the system, and uses these graphs to locate nondeterministic transitions of the threads throughout the run.

The pseudocode for coordinator agent C_i in charge of managing threads M_i is given below. The agent uses variables $s_1, \ldots, s_n$ to keep track of the states of all threads in the system.

Coordinator Agent C_i:

```
 1: $\forall i, s_i \leftarrow q_0^i$
 2: LastEvent $\leftarrow \phi$
 3: while true do
 4:    Sync $\leftarrow \phi$
 5:    while |Sync| $< |M_i|$ do
 6:       Receive synchronization request from thread $BT^j$
 7:       Mark the new state of $BT^j$ as $s'_j$
 8:       if LastEvent $\neq \phi$ and $|\delta^j(s_j, \text{LastEvent})| > 1$ then
 9:          Broadcast $s'_j$
10:       $s_j \leftarrow s'_j$
11:       Sync $\leftarrow$ Sync $\cup BT^j$
12:    for $BT^\ell \notin M_i$ do
13:       if LastEvent $\neq \phi$ then
14:          if $|\delta^\ell(s_\ell, \text{LastEvent})| > 1$ then
15:             Update $s_\ell$ according to broadcasts from other agents
16:          else
17:             $s_\ell \leftarrow \delta^\ell(s_\ell, \text{LastEvent})$
18:    $E \leftarrow \bigcup_{j=1}^{n}(R^j(s_j)) - \bigcup_{j=1}^{n}(B^j(s_j))$
19:    LastEvent $\leftarrow f(E)$
20:    Inform threads in $M_i$ that LastEvent was triggered
```

Note the slight abuse of notation of line 17 — where $\delta^\ell(s_\ell, \text{LastEvent})$ is not the state of the thread, but rather a set containing that state. Also, we implicitly assume that all broadcasts between the coordinator agents contain the index of the synchronization point that they refer to, to prevent cases where information about synchronization point t_1 could be mistakenly used in synchronization point t_2.

Intuitively, the coordinator agent waits for the threads that it manages (loop on line 5), same as in the centralized case. Whenever a thread synchronizes, the agent checks if the thread's last transition was nondeterministic (line 8). If so, the new state is broadcasted to the other agents — as they have no other way of finding out which transition was taken.

Once all the agent's threads have synchronized, it turns to consider threads that run on other machines. The key fact is that if a non-local thread is at a nondeterministic transition (line 14), the agent has to wait to receive a broadcast message (line 15) in order to determine the new state of that thread. Otherwise, it can go ahead and determine the thread's state locally (line 17).

After the synchronization requests of all threads have been determined, the next event to be triggered is selected (line 19), and then broadcasted to the agent's threads. This part is the reason for stipulating that f be a deterministic function — in order to maintain cohesiveness, all agents much trigger the same event on line 19.

Observe that each coordinator agent uses information regarding the transition functions (lines 8 and 14) and synchronization requests (line 18) of all the threads in the system — both threads that run locally on that agent, and threads that run on other agents. This information is given prior to the run, in the form of the state graphs of all the threads in the system.

Having formally defined the operation of each agent, we can now prove the following proposition, which is a technical formulation of Lemma 1:

Proposition 3. *Let $P = \{BT^1, \dots, BT^n\}$ be a behavioral program, divided into machines $M_1, \dots, M_k$ with coordinator agents $C_1, \dots, C_k$. Let $f : 2^{\Sigma} \setminus \{\emptyset\} \to \Sigma$ be a deterministic event selection function. Then agents $C_1, \dots, C_k$ produce a* cohesive *run; that is, there exists a unique run $e_1 e_2 \dots$ such that at synchronization point i,* every *coordinator C_ℓ triggers e_i. Further, the sequence $e_1 e_2 \dots$ is a valid run (under BP's semantics).*

For simplicity, we prove the lemma for the case of two machines, i.e. $n = 2$; the proof can easily be extended to any $n \in \mathbb{N}$. The proof follows directly from the next proposition, which is in turn proven by induction over the index of the synchronization points of the run.

Proposition 4. *For $i \in \mathbb{N}$ and $m \in \{1, 2\}$, let $s^i_m(BT^\ell)$ denote the state of thread BT^ℓ at synchronization point i, from the point of view of coordinator agent m. Let S^i_m denote the system-wide state at synchronization point i from the point of view of coordinator agent m; that is, $S^i_m = \langle s^i_m(BT^1), \dots, s^i_m(BT^n)\rangle$. Then for all $i \in \mathbb{N}$, it holds that $S^i_1 = S^i_2$.*

Proof. Let $i = 1$, which is the first synchronization point in the program. At this point, by the initialization in line 1 in the coordinator agent's code, $s^1_1(BT^\ell) = q^\ell_0$ and $s^1_2(BT^\ell) = q^\ell_0$ for all ℓ. Consequently, $S^1_1 = S^1_2$.

Now, suppose that $S^i_1 = S^i_2$ for some i. At synchronization point i, both coordinator agents triggered the same event e_i. This is so because the event selection function is deterministic, and thus both agents triggered event $e_i = f(E(S^i_1)) = f(E(S^i_2))$. This event was passed to all threads of the system by their respective coordinator agents.

Observe synchronization point $i+1$ from the point of view of C_1. As soon as all threads in M_1 have synchronized, C_1 knows their states. In order to determine the states of the remaining threads (those running on machine M_2), C_1 uses their pre-supplied state graphs. For any $BT^\ell \in M_2$, agent C_1 checks whether $|\delta^\ell(s_1^i(BT^\ell))| = 1$, and if so it deduces that $s_1^{i+1}(BT^\ell) = \delta^\ell(s_1^i(BT^\ell))$. In this case, C_2 will learn the state of BT^ℓ when that thread synchronizes, and it will hold that $s_1^{i+1}(BT^\ell) = s_2^{i+1}(BT^\ell)$.

The other option is that thread BT^ℓ is performing a nondeterministic transition, i.e. $|\delta^\ell(s_1^i(BT^\ell))| > 1$. In this case, C_1 has to wait for thread BT^ℓ to synchronize and reveal its state to C_2, after which C_2 will broadcast this state to C_1. In this case, it will also hold that $s_1^{i+1}(BT^\ell) = s_2^{i+1}(BT^\ell)$.

Further, upon receiving the synchronization request from a local thread BT^t, agent C_1 uses its stored state graphs to check whether $|\delta^t(s_1^i(BT^t))| > 1$. If so, C_1 transmits the thread's new state as learned from the synchronization request, $s_1^{i+1}(BT^t)$, to C_2 — to inform C_2 of how that nondeterministic transition was resolved.

As agent C_2 behaves symmetrically, we conclude that for all t it holds that $s_1^{i+1}(BT^t) = s_2^{i+1}(BT^t)$, and consequently that $S_1^{i+1} = S_2^{i+1}$. $\square$

Proposition 3 immediately follows from Proposition 4, and from the fact that f is a deterministic function. Indeed, $S_1^{i+1} = S_2^{i+1}$ implies identical calculation of the set E (line 18 in both agents, and thus the same output for $f(E)$. Finally, the fact that the resulting run is a legal BP follows from the definition of the set E to be the set of enabled events at the synchronization point.

I.3 Further Relaxing the Distributed Execution Mechanism

The distributed execution mechanism described above utilizes eager execution in the sense that each machine may be able to continue its execution without waiting for slower machines — except in nondeterministic transitions. We point out that further relaxation can be achieved by applying static or dynamic analysis to threads within the scope of each coordinator agent. As in the non-distributed case, this would allow faster threads within the same machine to continue their execution without waiting for their slower counterparts.

Another possible enhancement for the distributed model above is to use approximations for nondeterministic threads on other machines that slow down execution. Suppose that controller agent C_1 of machine M_1 is waiting for thread $BT \in M_2$ to finish its nondeterministic transition in order to trigger an event. As was the case in the centralized version, if C_1 can deduce, using the state graph of BT, that its next state will be either s_1 or s_2, it can approximate its requested and blocked events with $\mathcal{R} = R(s_1) \cap R(s_2)$ and $\mathcal{B} = B(s_1) \cup B(s_2)$. This further reduces the dependency between the different machines, hopefully achieving better optimization.

II Nondeterministic Threads

In several points in the paper, we mention and demonstrate the use of *nondeterministic* threads. In this section we formally define such threads and discuss applying the eager synchronization mechanism to them.

Nondeterministic threads can be intuitively thought of as threads that do not depend solely on the (behavioral) events triggered, but also on other sources of input — such as randomness or user actions. For example, consider a thread currently at state s. In this state, the thread waits for event e. When that event is triggered, the thread flips a coin; "heads" sends the thread to state s_h, and "tails" sends it to state s_t. Thus, the thread's transition does not depend solely on the triggered event, e. We call such threads nondeterministic.

Formally, nondeterministic threads are defined as follows: A *nondeterministic behavior thread (nondeterministic b-thread) BT* is abstractly defined to be a tuple $BT = \langle Q, q_0, \delta, R, B\rangle$, where

- Q is a set of *states*,
- $q_0 \in Q$ is an *initial state*,
- $\delta : Q \times \Sigma \to 2^Q \setminus \{\emptyset\}$ is a *transition function*,
- $R : Q \to \mathcal{P}(\Sigma)$ assigns for each state a set of *requested events*,
- $B : Q \to \mathcal{P}(\Sigma)$ assigns for each state a set of *blocked events*.

The difference between this definition and that of a (deterministic) b-thread is in the definition of δ; here it may map each state and event pair into more than one possible successor. For instance, in the example given above we would have $\delta(s, e) = \{s_h, s_t\}$.

The semantics of behavioral programs with nondeterministic threads are naturally defined as follows. Let $P = \{BT^1, \ldots, BT^n\}$ be a behavioral program, possibly with nondeterministic threads. We construct a labeled transition system $\mathrm{LTS}(P) = \langle Q, q_0, \delta\rangle$, where

- $Q := Q^1 \times \cdots \times Q^n$ is the set of states,
- $q_0 := \langle q_0^1, \ldots, q_0^n\rangle \in Q$ is the initial state,
- $\delta : Q \times \Sigma \to 2^Q$ is a (nondeterministic) transition function, defined for all $q = \langle q^1, \ldots, q^n\rangle \in Q$ and $a \in \Sigma$, by

$$\delta(q, a) := \begin{cases} \{\, \langle r^1, \ldots, r^n\rangle \mid r^i \in \delta^i(q^i, a) \,\} & ;\text{if } a \in E(q) \\ \emptyset & ;\text{otherwise}\,. \end{cases}$$

  where $E(q) = \bigcup_{i=1}^n R^i(q^i) \setminus \bigcup_{i=1}^n B^i(q^i)$ is the set of enabled events at state q.

As in the deterministic case, an execution of P is an execution of the induced $\mathrm{LTS}(P)$. The latter is executed starting from the initial state q_0. In each state $q \in Q$, an enabled event $a \in \Sigma$ is selected for triggering if such exists (i.e., an event $a \in \Sigma$ for which $\delta(q, a) \neq \{\emptyset\}$). Then, the system nondeterministically (that is, depending on coin tosses, user input, etc) moves to one of the next states $q' \in \delta(q, a)$, and the execution continues. Such an execution can be formally recorded as a possibly infinite sequence of triggered events, called a *run*. The set of all *complete* runs is denoted by $\mathfrak{L}(P) \triangleq \mathfrak{L}(\text{LTS}(P))$, which contains either infinite runs or finite ones that terminate in a state in which no event is enabled.

II.1 Eager Execution of Programs with Nondeterministic Threads

As we briefly mentioned in the paper, eager execution can be adapted to programs with nondeterministic threads. We now discuss this adaptation more thoroughly.

Static Analysis The first method for eager execution mentioned in the paper is that of static analysis. In this approach, the coordinator is given, prior to running the program, an over approximation of the events that the thread might block during its run. Clearly, this method can be applied to non-deterministic threads as-is: the threads' nondeterministic nature does not affect the validity of the over approximations.

Further, recall that in Section 5 of the paper we discussed leveraging the eager execution mechanism in designing modular behavioral programs. As mentioned therein, the results given rely on the use of static analysis of the threads. Consequently, as static analysis is invariant to nondeterministic threads, the results regarding modular design equally hold.

Dynamic Analysis The second method for eager execution that we mentioned relies on dynamic analysis. In this variant, the global coordinator uses the threads' state graphs to determine their future synchronization requests, while they are busy performing lengthy actions. Naturally, nondeterministic transitions in a thread's state graph pose a problem to this technique, as the coordinator cannot determine the state of the thread without knowing which transition was finally chosen. This information only becomes available when the thread synchronizes, but at that time it is no longer helpful.

We propose a slightly different variant, that is slightly weaker than dynamic analysis of deterministic threads but still superior to static analysis. Consider the example given earlier, where a thread determines its next state by tossing a coin; i.e., $\delta(s, e) = \{s_h, s_t\}$. Further, suppose that coin tossing takes a long time. The coordinator has no way of knowing if the thread is in state s_h or s_t until it synchronizes, but it can approximate its requested and blocked events by $R = R(s_h) \cap R(s_t)$ and $B = B(s_h) \cup B(s_t)$. More generally, if the thread is known to arrive in one of the states $Q = \{q_1, \ldots, q_n\}$ for its next synchronization point, the coordinator can approximate its event sets by $R = \bigcup_{i=1}^{n} R(q_i)$ and $B = \bigcap_{i=1}^{n} B(q_i)$. In many cases, these approximation may prove sufficiently tight to allow the triggering of the next event, without actually waiting for the thread to finish its lengthy operations and synchronize.

Observe that this method can also be applied iteratively — i.e., many more events can be triggered before the nondeterministic thread synchronizes. All that is required is that the coordinator properly maintains the set Q of states the thread can reach at its next synchronization point.

II.2 Eager Execution Formalized for Nondeterministic Threads

In this section we extend the formal definitions of eager execution in the natural way, to include proper handling of nondeterministic threads. While many of the particulars remain the same as in the deterministic case, we repeat them here for completeness.

Let $P = \{BT^1, \ldots, BT^n\}$ be a behavioral program, where $n \in \mathbb{N}$ and each BT^i is a distinct, possibly nondeterministic b-thread. In order to define the eager execution mechanism, we construct a labeled transition system (LTS) denoted by $\widehat{\text{LTS}}(P) = \langle \widehat{Q}, \widehat{q_0}, \widehat{\delta} \rangle$, which is defined as follows.

- $\widehat{Q} := (Q^1 \times \Sigma^*) \times \cdots \times (Q^n \times \Sigma^*)$ is the set of states, in which each state is a tuple consisting of the state of each thread and the contents of the corresponding event queue. Let $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$ be a state. We use the notation $\delta^i(q^i, u^i)$ to denote the set of states in Q^i that can be reached after applying the (nondeterministic) transition function δ^i of thread BT^i starting from state q^i for each event in the queue u^i. Given q, we denote the set of tuples comprised of possible combinations of these states by $ind(q) := \{\langle r^1, \ldots, r^n \rangle \mid r^i \in \delta^i(q^i, u^i)\}$ we refer to it as the *indication* of q. Note that each $\overline{q} \in ind(q)$ naturally corresponds to a state in Q, which is the set of states of $\text{LTS}(P) = \langle Q, q_0, \delta \rangle$ defined above. We slightly abuse notation and write that $\overline{q} \in Q$.
- $\widehat{q_0} := \langle (q_0^1, \varepsilon), \ldots, (q_0^n, \varepsilon) \rangle \in \widehat{Q}$ is the initial state.
- In each state $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$, eager execution *approximates* the requested and blocked events of each thread. This is indicated by the following sets of events: $\mathcal{R}^i(q) \subseteq \Sigma$, for the requested events of thread BT^i, and $\mathcal{B}^i(q) \subseteq \Sigma$, for the its blocked events. The requirements imposed on them are the following. We require that $\mathcal{R}^i(q)$ is a subset of the events that are requested by thread BT^i at any of the states in $\delta^i(q^i, u^i)$, and that $\mathcal{B}^i(q)$ is a superset of the blocked events at these states. That is,

$$\begin{aligned} \mathcal{R}^i(q) &\subseteq \bigcap_{v \in \delta^i(q^i,u^i)} R^i(v) \\ \bigcup_{v \in \delta^i(q^i,u^i)} B^i(v) &\subseteq \mathcal{B}^i(q)\,. \end{aligned} \tag{1}$$

  Moreover, we require that in case a thread is synchronized, the two approximations are precise. More formally, if $u^i = \varepsilon$ for some $i \in [n]$ (consequently, $\delta^i(q^i, u^i) = \{q^i\}$), then

$$\begin{aligned} \mathcal{R}^i(q) &= R^i(q^i) \\ \mathcal{B}^i(q) &= B^i(q^i)\,. \end{aligned} \tag{2}$$

  From these, we obtain that the *approximated enabled events*, defined in the following, are contained in the enabled events at any of the states in $ind(q)$; i.e., $\forall \overline{q} \in ind(q)$

$$\mathcal{E}(q) := \bigcup_{i=1}^{n} \mathcal{R}^i(q) \setminus \bigcup_{i=1}^{n} \mathcal{B}^i(q) \subseteq E(\overline{q})\,.$$

In case all threads are synchronized, i.e., $u^i = \varepsilon$ for all $i \in [n]$, we obtain that $ind(q) = \{\langle q^1, \ldots, q^n \rangle\}$ and

$$\mathcal{E}(q) = E(\langle q^1, \ldots, q^n \rangle)\,. \tag{3}$$

– $\widehat{\delta} : \widehat{Q} \times (\Sigma \dot{\cup} \{\varepsilon\}) \to 2^{\widehat{Q}}$ is a nondeterministic transition function, which includes also silent ε-labeled transitions; these ε transitions are not considered part of the runs of the system. $\widehat{\delta}$ is defined for each state $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$, and $\sigma \in \Sigma \cup \{\varepsilon\}$, as follows:
  - If $\sigma = \varepsilon$, then $\widehat{\delta}(q, \varepsilon)$ is defined to be those states $\langle r^i, v^i \rangle_{i=1}^n \in \widehat{Q}$ for which there is $i_0 \in [n]$ and $a \in \Sigma$ such that $u^{i_0} = a\, v^{i_0}$ and $r^{i_0} \in \delta^{i_0}(q^{i_0}, a)$, and for all other $i \in [n] \setminus \{i_0\}$ it holds that $r^i = q^i$ and $v^i = u^i$. Each of these transitions corresponds to a thread with queued events when it finishes processing the head of the queue — it changes states, while the other threads don't move.
  - If $\sigma \in \Sigma$, and moreover $\sigma \in \mathcal{E}(q)$, then $\widehat{\delta}(q, \sigma)$ is defined to be the singleton $\widehat{\delta}(q, \sigma) = \big\{ \langle q^i, u^i\, \sigma \rangle_{i=1}^n \big\}$. These transitions correspond to new events being triggered.
  - If $\sigma \in \Sigma$ and $\sigma \notin \mathcal{E}(q)$, we define $\widehat{\delta}(q, \sigma) = \emptyset$. This reflects the fact that events that are not enabled cannot be triggered.

The definitions above capture the case of nondeterministic threads. They can be used to prove the result of the matching section in the paper for the nondeterministic case: that in the nondeterministic case it also holds that each complete run of $\widehat{\text{LTS}}(P)$ is a complete run of $\text{LTS}(P)$; i.e., $\mathfrak{L}(\widehat{\text{LTS}}(P)) \subseteq \mathfrak{L}(\text{LTS}(P))$. The actual proof is similar to that of the deterministic case.

III Modularity Formalized

In this section we use the formalization of eager execution described in Section 3.3 in order to rigorously formulate and prove Proposition 2.

Let $P = \{BT^1, \ldots, BT^n\}$ be a behavioral program (where $n \in \mathbb{N}$ and each BT^i is a distinct b-thread). Assume that P is composed of behavioral modules $M_1, \ldots, M_k$; i.e., $M_1, \ldots, M_k$ is a partition of the threads. For each thread BT^i, the set of events *controlled* by BT^i is denoted by $C^i := \Big(\bigcup_{s \in Q^i} B^i(s) \Big) \cup \Big(\bigcup_{s \in Q^i} R^i(s) \Big)$. For each module M_j, the set $E_j := \bigcup_{i \,:\, BT^i \in M_j} C^i$ is the set of events controlled in M_j. We assume that the modular design is *strict*; i.e., $E_1, \ldots, E_k$ are pairwise disjoint.

We will assume that the program P is executed with the eager execution mechanism, which is formalized as the transition system $\widehat{\text{LTS}}(P)$ in Section 3.3. The strict modular design translates into a constraint on the approximations used — namely, that these approximations only include events controlled by the specific module. Formally, in each state $q \in \widehat{Q}$, and for each module M_j and thread $BT^i \in M_j$, the approximation of the blocked events satisfies

$$\mathcal{B}^i(q) \subseteq E_j\,. \tag{4}$$

This obviously holds in both static and dynamic analysis. Observe that the analogous constraint, $\mathcal{R}^i(q) \subseteq E_j$, follows directly from (1).

We now turn to prove the following technical proposition that, when applied iteratively, implies Proposition 2.

Proposition 5. *Let* $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$ *be a state of* $\widehat{\mathrm{LTS}}(P)$ *in which all threads of module* M_j *have already synchronized; i.e., if* $BT^i \in M_j$ *then* $u^i = \varepsilon$. *Let* $q' = \langle r^i, v^i \rangle_{i=1}^n \in \widehat{Q}$ *be a state such that* $q \xrightarrow{\varepsilon} q'$ *in* $\widehat{\mathrm{LTS}}(P)$. *Then for all* $i \in [n]$ *such that* $BT^i \in M_j$ *it holds that* $v^i = \varepsilon$, *and an event* $e \in E_j$ *is enabled in* q, *i.e.* $e \in \mathcal{E}(q)$, *if and only if it is also enabled in* q', *i.e.* $e \in \mathcal{E}(q')$.

Proof. By the definition of the transition function $\widehat{\delta}$ of $\widehat{\mathrm{LTS}}(P)$, for all $i \in [n]$ such that $BT^i \in M_j$ it holds that $v^i = u^i = \varepsilon$, as required, and also $r^i = q^i$.

We begin by showing that $e \in \mathcal{E}(q') \implies e \in \mathcal{E}(q)$. By (3), $e \in \mathcal{E}(q')$ implies $e \in \mathcal{R}^l(q')$ for some $l \in [n]$, and $e \notin \bigcup_{i=1}^n \mathcal{B}^i(q')$. By (1), $\mathcal{R}^l(q') \subseteq C^l$, where C^l is the set of events controlled by BT^l; as $e \in E_j$ and the design is strict, $BT^l \in M_j$. From the above, we get that $r^l = q^l$ and $v^l = u^l = \varepsilon$. Therefore, from (2) we obtain $\mathcal{R}^l(q) = R^l(q^l) = \mathcal{R}^l(q')$, so that $e \in \mathcal{R}^l(q)$. For the same reason, and due to (2), for all $i \in [n]$ such that $BT^i \in M_j$, it holds that $\mathcal{B}^i(q) = B^i(q^i) = \mathcal{B}^i(q')$; as we know that e is not in the latter approximation set, we get that $e \notin \mathcal{B}^i(q)$. For other $i \in [n]$, for which $BT^i \notin M_j$, we get from (4), and from the design being strict, that $\mathcal{B}^i(q) \subseteq \Sigma \setminus E_j$. Consequently, here also, $e \notin \mathcal{B}^i(q)$. Conclude that $e \in \bigcup_{i=1}^n \mathcal{R}^i(q) \setminus \bigcup_{i=1}^n \mathcal{B}^i(q) = \mathcal{E}(q)$, as needed.

The proof for the other direction, i.e. $e \in \mathcal{E}(q) \implies e \in \mathcal{E}(q')$, is similar and is omitted. The proposition follows. □

References

1. W. Cai, F. Lee, and L. Chen. An Auto-Adaptive Dead Reckoning Algorithm for Distributed Interactive Simulation. In *Proc. 13th IEEE. Workshop on Parallel and Distributed Simulation (PADS)*, pages 82–89, 1999.
2. R. Fujimoto. Parallel and Distributed Simulation. In *Proc. Winter Simulation Conference (WSC)*, pages 118–125, 1995.