Towards Behavioral Programming in Distributed Architectures[1]

David Harel[a], Amir Kantor[b], Guy Katz[a], Assaf Marron[a], Gera Weiss[c], Guy Wiener[d]

[a] *Weizmann Institute of Science, Rehovot, Israel*
[b] *IBM Research, Haifa Lab, Mt. Carmel, Israel*
[c] *Ben-Gurion University, Beer-Sheva, Israel*
[d] *HP Labs, Haifa, Israel*

Abstract

As part of expanding the implementation and use of the *behavioral programming* (BP) approach in a variety of languages and configurations, we tackle some of the challenges associated with applying the approach in a truly distributed, decentralized manner, where different modules run on separate machines. BP supports the development of reactive applications from modules that are aligned with the desired and undesired scenarios of system behaviors as described, say, in a requirements document, in an enhancement request, or a field problem report. A key advantage of this approach is that it facilitates incremental development where loosely coupled modules are added as requirements are introduced, and meaningful prototype execution can be carried out from early stages of development. In BP, each behavioral module (called a behavior thread) takes care of a separate facet of the requirements, thus control is conceptually decentralized. However, as the underlying principles of BP call for constant synchronization of, and "consultation" with, all behavior threads, efficient implementation in a physically distributed environment is a significant challenge on the road to broader acceptance of BP as a viable new way to develop systems. We begin by describing an implementation of BP in Erlang, where the coordination protocol is implemented via message passing. We demonstrate through examples how developing distributed systems in Erlang can benefit from BP advantages of incremental development and alignment of the code modules with the requirements. Next, we propose general BP design patterns (not limited to Erlang) for using BP in distributed applications, showing how to design applications using BP without forcing full synchronization among all threads at each step of the execution. This allows modules to run at different time scales and to wait for external input without stalling the entire system. Finally, we propose ways to alter the execution mechanism of BP so that execution can progress without necessarily waiting for the synchronization of all the threads. The enhanced execution algorithm has the potential of accelerating the distributed execution of behavioral programs.

[1] Preliminary versions of parts of the material in this paper appeared in D. Harel, A. Marron, G. Weiss, and G. Wiener, "Behavioral programming, decentralized control, and multiple time scales", *Proc. AGERE!*, 2011, G. Wiener, G. Weiss, and A. Marron, "Coordinating and Visualizing Independent Behaviors in Erlang", *Proc. ACM SIGPLAN Erlang Workshop*, 2010, D. Harel, A. Kantor, G. Katz, "Relaxing Synchronization Constraints in Behavioral Programs", *Proc. LPAR*, 2013.

1. Introduction

Behavioral Programming (BP), also termed *scenario-based programming*, is an approach for the incremental development of reactive systems based on decentralized control and interwoven modules of behavior. A basic tenet of BP is the constant cross-consultation among parallel processes, prior to each system action. In this paper we address the challenge of achieving the implied synchronization in physically distributed environments, avoiding performance degradations that could put into question the viability of BP as a system development approach. Specifically, we show that it is possible to achieve correct and efficient behavioral execution with parallel distributed scenarios that run with hard-to-predict network delays, or are clocked at very different time scales, or wait for external events that cannot be synchronized.

The principles of BP were first introduced in the visual language of *live sequence charts* (LSC) [14, 21], where parallel processes represent multi-modal scenarios of desired, mandatory or forbidden behavior of the system. These principles were later implemented in other programming languages and frameworks, such as Java [23], JavaScript with Blockly [38], SBT [35], and in the C language in the PiCos environment [42]. As part of this extension the term *behavior threads* or *b-threads*, defined formally below, was introduced. For an extensive review of the BP approach see [27]. In this paper, we begin the handling of distributed environments by describing, in Section 2, the BP development approach as a design pattern in Erlang [4]. Erlang was chosen because its entire architecture and development mindset is based on decomposing the application into relatively independent parallel processes, which indeed fits well with the basic principles of BP. We then explore how the two paradigms coexist. We demonstrate how enabling behavioral programming in Erlang enhances the capabilities for incremental development and how it allows alignment of the code with the scenarios described in the requirements documents. BP will thus join and complement other design patterns and packages in Erlang, such as `gen_server` or `gen_fsm`, aimed to assist in coordinating distributed processes (`gen_server` is an Erlang package for implementing the server of a client-server relation, and `gen_fsm` is a package for implementing a finite state machine). Specifically, we propose to implement b-threads as Erlang processes. The system is executed using an infrastructure module that uses standard Erlang message passing to constantly synchronize and coordinate the b-threads. We also introduce a proof-of-concept mechanism for visualizing the flow of individual scenarios coded in Erlang as self-explanatory transition systems, aimed at improving the comprehension of applications written in this manner. That section thus shows that the natural, incremental development facilitated by BP applies also in the context of distributed architectures based on message passing.

The second part of the answer comes in Section 3, where we present general BP design patterns that allow applications to deal with the challenges imposed by the requirement for constant synchronization. Basically, in that section we describe how to design b-threads to

Email addresses: `david.harel@weizmann.ac.il` (David Harel), `amirka@il.ibm.com` (Amir Kantor), `guy.katz@weizmann.ac.il` (Guy Katz), `assaf.marron@weizmann.ac.il` (Assaf Marron), `geraw@cs.bgu.ac.il` (Gera Weiss), `guy.wiener@hp.com` (Guy Wiener)

wait for external events without suspending the entire application. We propose to decompose the system into groups of b-threads that synchronize only internally within the groups and use asynchronous events to communicate with each other and with the external environment, relaxing the strict synchronization requirements. Through examples we show that these design patterns, although somewhat constraining, retain the naturalness and incrementality of development with BP.

Finally, the third element of the answer comes in Section 4, where we explore automatic ways for dealing with the above challenges. Specifically, we propose enhancements to the BP event-selection algorithm that take into account additional information about the internal structure of b-threads. This information can be used to perform early event selection, without necessarily waiting for all b-threads to synchronize. The non-synchronized threads are notified of the relevant events that they have missed via a queuing mechanism, which is provided by the BP infrastructure.

The application designs of Section 3 and the enhanced algorithms of Section 4 are language independent, and are not restricted to Erlang. Further, they can serve as alternatives to each other, or can complement each other in a single application, thus helping to accelerate the distributed execution of behavioral programs. In Section 5 we compare and contrast the solutions of these two sections.

In Section 6 we discuss how BP coexists with and complements actor-oriented, agent-oriented and aspect-oriented programming, as well as other decentralized control development approaches.

The code for the example applications as well as the infrastructure modules is available online at `http://www.b-prog.org`.

2. Behavioral Programming in Erlang

In this section we introduce BP directly, without referring to prior implementations, by describing in detail the proposed Erlang design pattern and the Erlang code module that implements it. The subsections below follow largely the section structure used by "The Gang of Four" for documenting design patterns[19] , including sections such as intent, motivation, applicability, structure, sample code, and related patterns. The description of BP as an Erlang design pattern is followed by the main formal definitions for BP as adopted from [23, 24], showing the same principles as composition of transition systems, in a language-independent manner. The section then concludes with a description of a visualization tool that translates individual scenarios coded in Erlang into drawings of transition systems, in support of the comprehension both of the individual scenarios themselves and of their composition in the integrated application.

2.1. Intent

We propose a design pattern called BP, and an associated module called `bp` (whose operation is described in subsequent subsections), for iteratively creating a sequence of events, where the next event is chosen with the help of the bidding protocol described

below. The bidders are Erlang processes registered as behavior threads (b-threads). In each iteration:

1. Each b-thread places a bid consisting of:

   Requested events: events that the b-thread proposes to be considered for triggering.

   Waited-for events: other events that the b-thread asks to be notified of if and when triggered.

   Blocked events: events that the b-thread forbids.

2. When all b-threads place their bids, an auction takes place: an evaluation mechanism chooses an event that is requested by some b-thread and not blocked by any b-thread.

3. B-threads are notified of the auction's outcome: the b-threads that requested or waited for the event are notified and resumed.

4. Resumed b-threads can execute arbitrary computations before placing their next bid in the subsequent iteration.

5. B-threads that were not resumed remain suspended, and their bid is considered again when all resumed b-threads complete their computations and place new bids.

In the auction step, if there are multiple requested events that are not blocked, the design pattern allows the designer to specify the event selection algorithm. Among the selection techniques we have implemented to-date are priority-based selection, random selection, probabilistic with reinforcement learning, and look-ahead subject to feedback regarding success or failure. In the remainder of the paper, we assume the priority-based selection, where b-threads are ordered subject to fixed priorities and all the requests of a given b-thread are ordered as well. The first event that is requested and not blocked is selected for triggering.

2.2. Motivation

The motivation for proposing this design pattern is to provide a simple mechanism, through which systems can be constructed from software components, each of which controls and coordinates a particular aspect of desired system behavior. Such construction complements the traditional approach to software development where program modularity revolves around objects and data-structures, and simplifies the programming of required behaviors whose intuitive description involves multiple objects and is not anchored on one "classical" object. Using events as markers of system behavior, and applying the proposed bidding mechanism for choosing events, the resulting integrated system behavior is an event sequence that reflects, at each step, every b-thread's view of how the system should proceed.

The BP design pattern helps programmers maintain b-thread independence by unifying the occurrence of events requested by several b-threads into a single occurrence, and by the fact that b-threads can block events regardless of whether other b-threads request them or not. As in standard publish-subscribe protocols, b-threads do not communicate with each other directly, and they are not notified of events that they do not request or wait for.

2.3. Applicability

BP is best used when the decomposition of system's behavior into independent scenarios is natural. For example, in automated transportation, it is natural to separate navigation, speed control, and stabilization into parallel scenarios, and in a game-playing application each game rule and each player strategy can be described in a separate scenario. Since interesting system behaviors can be observed in early stages of software design and specification, with only a few b-threads, BP is valuable in experimenting with for obtaining feedbacks from stakeholders and validating the understanding of requirements.

More generally, the BP design pattern is suitable for incremental development, as it allows adding and removing system behaviors with little or no change to existing code (if that code was developed with BP).

BP is also applicable in end-user customization, where end -users can add or remove behaviors in an existing application, say, to simplify activities or avoid mistakes, by adding b-threads that handle specific sequences of events.

Clearly, the apparent ease of incremental development comes with a price. For example, after a long period of development an application may appear as a loosely coupled collection of base scenarios, exceptions and exceptions to exceptions. It is then the role of the developers to determine whether to add a new requirement as a new scenario or to modify existing scenarios, and to decide when restructuring is desired. Another concern is the security controls over incremental development and end-user customization. With idioms such as priorities and event blocking at their disposal, incremental additions can have vast effects over the behavior of the application. Developers may wish to certify sources of such updates, or to introduce explicit permission mechanisms controlling what newly added b-threads are allowed to do.

There are, of course, cases where BP may not be the first choice. One such situation is when there are hard-to-achieve performance constraints for which the general solutions offered here and elsewhere are not sufficient, and application-specific and/or optimized algorithms are needed. Another case is where the requirements scenarios are already provided as a well documented algorithm, or can be readily combined into one easy to maintain scenario. Consider for example a robot navigating a maze using the well-known left-wall rule. In [23] it is shown how the robot's behavior can be decomposed into three scenarios (described in priority order): (1) repeatedly wait for a move forward and turn left, (2) repeatedly move forward (3) repeatedly turn right. While this decomposition is attractive in that it avoids any conditions in the scenarios, one may argue that it is excessive, and that the single scenario 'repeatedly do the following: turn left; if the robot is facing a wall turn right; otherwise, move forward' is preferable. In fact, there may be cases where replacing the parallel coordinated b-threads with a sequential procedure containing a few conditions may provide the necessary performance advantage. In development scenarios where incrementality is not needed, the arguments for using BP are further weakened. Nevertheless, we are interested in finding ways for optimizing performance of parallel execution of behavioral scenario bringing it as close as possible to that of more standard applications. Techniques include, among others, relaxing the synchronization requirements (as described in Sections 3 and 4), optimizing the data structures and algorithms for carrying out synchronization, de-

signing operating systems and hardware geared explicitly to efficient execution of behavioral programs, and, synthesizing single-threaded controllers from behavioral specifications with parallel scenarios.

As for BP implementations in Erlang, it should be noted that Erlang is able to handle numerous processes [4], and indeed, in our test of thousands of Erlang b-threads randomly exchanging events at a high rate, we did not observe slowdown due to the required coordination.

2.4. Structure

In the BP design pattern a system is constructed from independent b-threads that synchronize constantly and communicate indirectly using agreed-upon events, with the help of the coordination module.

2.5. Participants

In this design pattern, the participants are:

- The b-threads: The work-doing processes, each responsible for an aspect of the application/system behavior. Repeatedly, they
  - bid for the next event by sending a synchronization request to the BP controller with the following parameters: (a) requested events, (b) waited-for events, (c) blocked events;
  - wait until the BP controller sends them an event they are requesting or waiting for.
- The BP controller: A central server process. Repeatedly, it
  - receives synchronization messages from b-threads until all b-threads are synchronized,
  - decides on the next event,
  - sends the next event to the b-threads requesting it or waiting for it.

2.6. Collaboration

The collaboration between the BP controller and the b-threads is as follows (also depicted in Figure 1):

1. The BP controller is initialized.
2. B-thread processes are spawned and the method `bp:add` is called to register them at the controller.
3. The BP controller is started.
4. B-threads call `bp:bSync` to send their requests to the BP controller, and are suspended until they receive a response from the controller.
5. The BP controller waits until all the active registered b-threads send their requests.

6. The BP controller decides on the next event and sends responses to the b-threads requesting it or waiting for it.
7. The b-threads that receive the new event continue their computation until they call `bp:bSync` again.
8. When a b-thread exits, it is deregistered from the BP controller.

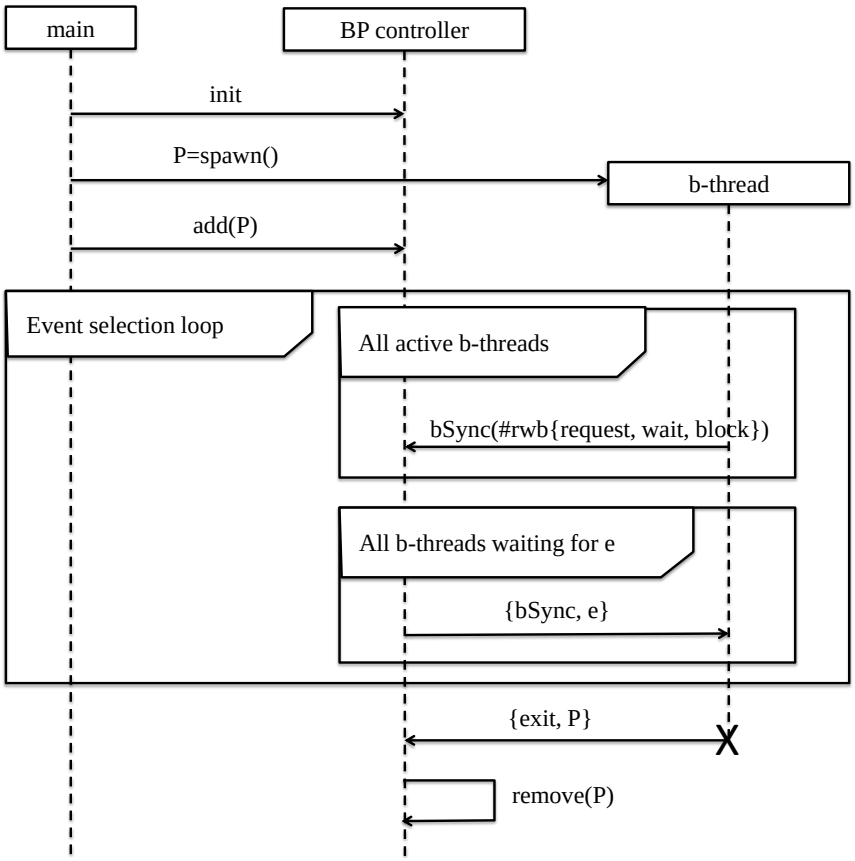


Figure 1: A sequence diagram depicting the collaboration between the BP controller and the b-threads. See detailed explanation of the steps in the text.

2.7. Code structure

Our implementation of the BP design pattern is based on a supporting module, `bp`. The `bp` module exports the following functions:

`init/0` : Initialize the BP controller.

`add/2` : Register a process with a given priority.

`start/0` : Start the controller.

sync/1 : Send a synchronization request that includes requested, waited-for and blocked events to the controller and wait until one of the requested or waited-for events is selected. This function takes a record as an argument. The record definition is `-record(rwb, {request, wait, block})`, where `rwb` is the chosen record name and with record fields matching the respective event sets, each having an empty list as its default value. For example, the code below invokes `bp:bSync`, passing to it a `rwb` record (referenced as `#rwb`), where it requests the event `E1`, waits in addition also for `E2` and `E3`, and blocks `E4` until one of the events `E1`,`E2`, or `E3` occurs:

```
bp:bSync(#rwb{
   request=[E1],wait=[E2,E3],block=[E4]})
```

remove/1 : Deregister a process from the controller.

It is the programmer's responsibility to initialize the `bp` process and spawn the b-threads as processes. There are no constraints on the calculations that b-threads perform before or after the calls to `bp:bSync`. However, when b-threads perform heavy calculations or wait for external events, they affect synchronization of all participating b-threads. Design patterns for addressing this kind of issue are presented in Section 3.

2.8. A simple example

To illustrate this coding technique, consider a system for controlling the water level in a tank with hot and cold water sources (the example is borrowed from [26]). One of the b-threads in the system requests the event `addHot` five times. Another b-thread adds five quantities of cold water, by similarly requesting the event `addCold` five times. After observing a run in which the five `addHot` events occurred before the first `addCold` event, a new requirement is introduced by the developer, to the effect that water temperature should be kept relatively stable. The developer then adds a b-thread that interleaves `addHot` and `addCold` events, using the event-blocking idiom, by repeatedly performing the steps of first waiting for `addHot` while blocking `addCold` and then waiting for `addCold` while blocking `addHot`. In this example, the interleaving b-thread was added without changing existing b-threads and without specifying in the new b-thread direct interactions with the existing ones.

The code below illustrates how to implement this system in Erlang, using the `bp` module. Detailed explanation of the code then follows.

```
requestFiveAddHotEvents() ->
  [bp:bSync(#rwb{request=[addHot]}) || _ <- seq(1,5)].

requestFiveAddColdEvents() ->
  [bp:bSync(#rwb{request=[addCold]}) || _ <- seq(1,5)].

interleave() ->
  bp:bSync(#rwb{wait=[addHot], block=[addCold]}),
  bp:bSync(#rwb{wait=[addCold], block=[addHot]}),
  interleave().

display() ->
  Event = bp:bSync(#rwb{wait=?ALL}),
```

```
  io:format("Event: ~w~n", [Event]),
  display().

test() ->
  bp:init(),
  bp:add(spawn(fun requestFiveAddHotEvents/0), 1),
  bp:add(spawn(fun requestFiveAddColdEvents/0), 2),
  bp:add(spawn(fun interleave/0), 3),
  bp:add(spawn(fun display/0), 4),
  bp:start().
```

and the resulting event log is

```
Event: addHot
Event: addCold
Event: addHot
Event: addCold
Event: addHot
Event: addCold
Event: addHot
Event: addCold
Event: addHot
Event: addCold
```

The behavior threads are implemented as four functions that are spawned to run in separate processes. The b-thread `requestFiveAddHotEvents`, for example, calls `sync` five times. The loop is implemented using Erlang *list comprehensions* where the expression `[X || _ <- seq(1,N)]` is shorthand for "perform X N times". In each iteration of the loop the process passes `addHot` as the only requested event and blocks nothing. The b-thread `requestFiveAddColdEvents` is similar, and the b-thread `interleave` repeatedly waits for and blocks alternating events. The infinite loop of this b-thread is implemented by a recursive call to itself. The `wait` and `block` clauses may also use filter functions for events instead of an explicit list. For example, the macro `?ALL` brings a filter for all possible events.

The `display` behavior thread waits for all events and for each received event it generates textual output for illustration purposes. It then invokes itself recursively. In real applications this b-thread would be replaced by an actuator that translates the events into physical outputs, such as tap opening (see more about sensors and actuators in [38] or in [5] in this journal issue).

Note also the second parameter of the `bp:add` function, which represents the priority of the b-thread, as discussed in Section 2.1. To appreciate the role of priority, consider, for example, the water tap application without the `interleave` b-thread. Because the priority of `requestFiveAddHotEvents` is higher than that of `requestFiveAddColdEvents`, we would get the five `addHot` events before the five `addCold` Events.

2.9. Formal Definitions

For completeness of the exposition of BP we include here the formal definition of behavioral programming as transition systems.

2.9.1. B-Threads

In the following definitions we implicitly assume a given set Σ of *events*. A *behavior thread (b-thread)* BT is 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*, $\delta : Q \times \Sigma \to Q$ is a *transition function*, $R : Q \to \mathcal{P}(\Sigma)$ assigns to each state a set of *requested events*, and $B : Q \to \mathcal{P}(\Sigma)$ assigns to each state a set of *blocked events*.

Note that in these definitions, a b-thread's transition rules are given as a *deterministic*, single valued, function δ, assigning the next state given a state and an event triggered in that state. A natural variant in which the transitions are *nondeterministic*, is defined analogously; see Appendix C. Since behavioral programs may contain rich functionality beyond their state transitions, the nondeterministic transitions can be a useful abstraction for describing transitions that are based on auxiliary information. For instance, consider a b-thread currently in state s_1, waiting for event e_1 which represents a temperature measurement by a sensor. When e_1 is observed, the thread transitions into either state s_2 or s_3 based on the actual value of the measurement. The abstraction of the transition as nondeterministic, is thus a convenience, where the alternatives include distinguishing events with different temperatures as totally different events, or adding guard conditions to the transitions, as is common in other formalisms, such as statecharts [20]. Abstracting transitions as nondeterministic is also useful when a thread chooses the next state at random; say, to create a mix of certain actions.

Also note that in the formal definition of a b-thread, there is no need to distinguish between events that are waited-for by the thread, and those that are not. In any of the thread's states, an event that is not waited-for can be captured by a transition that forms a self-loop; i.e., a transition that does not leave the state.

2.9.2. Behavioral Programs

Observe a set $\{BT^1, \ldots, BT^n\}$ of b-threads, where $n \in \mathbb{N}$ and each $BT^i = \langle Q^i, q_0^i, \delta^i, R^i, B^i \rangle$ is a distinct b-thread. A behavioral program P comprised of these threads is a deterministic *labeled transition system (LTS)* [33], defined as follows. $P = \langle Q, q_0, \delta \rangle$, 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 deterministic transition function (i.e., one whose range includes only singletons and the empty set), defined for all $q = \langle q^1, \ldots, q^n \rangle \in Q$ and $a \in \Sigma$, by

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

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

An execution of the behavioral program P is an execution of the induced LTS. 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 an event exists (i.e., an event $a \in \Sigma$ for which $\delta(q, a) \neq \emptyset$). Then, the system moves to the next state $q' \in \delta(q, a)$, and the execution continues. Such an execution can be recorded as a possibly infinite sequence of triggered events, called a *run*. The set of all *complete* runs is denoted by $\mathfrak{L}(P)$. It contains either infinite runs, or finite ones that terminate in a state in which no event is enabled, called a *terminal state*.

As noted in subsection 2.1, when multiple events are requested and not blocked, the semantics does not specify which one will be chosen and a variety of techniques can be

implemented. Further, in Section 3.2 we discuss why the chosen semantics does not allow the simultaneous triggering of multiple events.

2.10. Visualization

The `bp` module can work with main programs and b-threads, regardless of the structure of their code, as long as they use the interface methods described above. Nevertheless, as the b-thread in fact implements a transition system, it is natural to try and create a mental image of this transition system. The resulting representation of the logic can then be useful to developers and other stakeholders in maintaining the software of individual b-threads or in thinking about the collective execution of a set of b-threads. To this end we have developed a proof-of-concept tool (a module named `bp_vis`) that automatically produces a visual representation of a b-thread's behavior as a transition system.

The code visualizer assumes that the code is structured as a state machine, following the method described in [2]. The code for each b-thread is contained in a separate module, and the states of the b-thread are represented as functions in the module. The function `start` is the first state of the b-thread. The body of each function associated with a b-thread state is written as follows:

```
case bp:bSync(#rwb{...}) of
  x -> state1();
  y -> state2();
  ...
end.
```

The function body consists of a `case` statement where the evaluated expression is a call to `bp:bSync`, and where each clause maps the returned event to another function call, which represents transitioning into the next state. The diagram is generated as a Graphviz file (see `http://graphviz.org`), with the following format:

- State functions appears as ellipses with multi-line labels. The first line is the function signature and the subsequent lines show the content of the request, wait and block arguments of the call to `bp:bSync` in the state function.
- An edge from ellipse A to ellipse B appears if one of the clauses in the case statement in the state function A is a function call to B.
- Each edge has a label "`Event (when Guard) / Call`". The `Event` is the head of the case clause whose body is the function call to B. The `Guard` is the guard part of the head of the clause, which is optional. The `call` is the exact function call to B, including arguments.
- If one of the state functions is called "`start`", it is emphasized by a small arrow, emanating from a black dot.

For example, suppose that some device can print, scan, send a fax and stop. Consider the following single b-thread:

```
-module(printjob).
-compile([{parse_transform, bp_vis}]).
-include("bp.hrl").
-define(LIMIT, 3).

pending(N) ->
  case bp:bSync(#rwb{request=[print],
             wait=[print, scan, fax],
             block=[stop]}) of
    print -> working();
    _ when N < ?LIMIT -> pending(N+1);
    _ when N >= ?LIMIT -> idle()
  end.

working() ->
  case bp:bSync(#rwb{wait=[finish],
                     block=[stop]}) of
    finish -> idle()
  end.

idle() ->
  case bp:bSync(#rwb{wait=[stop]}) of
    stop -> ok
  end.
```

This module has the following behavior. It tries to print three times. If the printing starts, it waits for it to end. If it fails more then three times, it gives up. When waiting for printing, it prevents the machine from stopping. The output of the visualization tool for this module is depicted in Figure 2. Since this b-thread does not have a designated `start` function, the diagram does not include a corresponding initial state arrow.

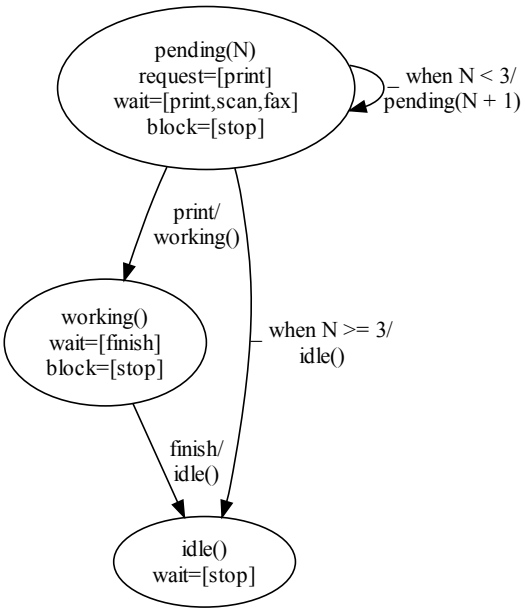
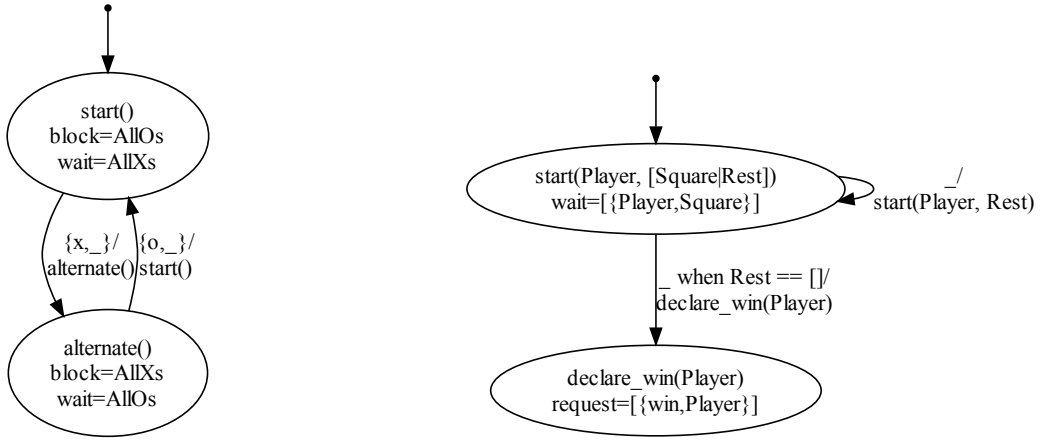


Figure 2: Visualization tool output. The transition diagram of a single b-thread

As another example, we developed a behavioral application in Erlang for playing Tic-

Tac-Toe. Figure 3 illustrates two b-threads of this application.



(a) Enforce Turns (b) Detect Win

Figure 3: Visualization of sample b-threads in an Erlang application for playing Tic-Tac-Toe. Two players, X and O, try to complete a line with marks of Xs and Os on a 3 by 3 grid. In the b-thread `enforce_turns`, `AllO` and `AllX` are lists of all the O events and all the X events, respectively. Turn enforcement is achieved by alternately blocking all events in one set while waiting for the events in the other. An instance of the `detect_win` b-thread is spawned for each of the six permutation of the three events that comprise one of the eight winning lines (3 vertical, 3 horizontal, and 2 diagonal) for each of the two players (yielding a total of 96 instances).

2.11. Consequences

In this subsection we discuss some (possibly negative) consequences and implications of the usage of BP, and ways to mitigate them.

Behavioral programming is based on consensus: It requires all participating processes to agree on the next step. It thus requires a synchronization point for all b-threads that are not waiting. To exit the blocking call to `bp:bSync`, each active b-thread must wait until all other active b-threads call `bp:bSync`. Although this is a reasonable requirement for an agreement protocol, a specialized protocol for a specific problem can be more efficient.

Reaching an agreement also requires sharing information between processes. In our implementation, we use a central process (the BP controller) to collect and handle the requests from all participating b-threads. This process is thus critical to the operation of the system. In mission critical real-world systems, special attention should be given to such potential single-point-of-failure.

Several properties of the BP architecture and its semantics can be exploited in designing the desired recovery capabilities, especially when combined with Erlang/OTP built-in recovery facilities. First, we observe that the state of the BP controller that must be saved or made redundant to enable recovery is quite minimal. It consists of the most recent declarations of all the b-threads that are presently at a synchronization point. In fact, in the

absence of redundancy and state saving, a simple protocol can be established, by which a recovered BP controller inquires of all synchronized b-threads as to their most recent declarations of requested, waited for and blocked events. As to recovering a failed b-thread, all standard ways of saving a process's state during execution and using system restart and recovery capabilities, are obviously available. In addition, one way to restore the state of a restarted b-thread that is application agnostic can be as follows: During the run, make sure that the log, or trace, of triggered events is saved; start a failed b-thread at its ordinary initial state; run it in a closed "sandbox", such that it does not interfere with other b-threads; execute it in "monitoring mode", i.e., interpret all its requested events only as waited for; select at each synchronization point the event that was selected in the real run, and feed it to the recovered b-thread, until the trace is exhausted. The restarted b-thread is now in synchronization with the rest. Another recovery option is to have b-threads designed specifically for this purpose. That is, when such a b-thread starts, in an analogy to a player joining a soccer game in the middle, it explores the present configuration, and/or some limited recent history of events, and determines its own desired state.

2.12. Example: Coordinated sequential processing

To illustrate one of the ways in which the BP design pattern can be used, we discuss in Appendix A an example involving applications that require bulk processing of a large number of records to perform business operations, and where multiple, independent sequential processes need to be interwoven.

2.13. Known uses and related design patterns

The event-based and state-like nature of BP makes b-threads similar to the generic finite state machine (`gen_fsm`) module (`http://www.erlang.org/doc/man/gen_fsm.html`) from the Erlang standard library. Both the `gen_fsm` and `bp` modules deal with a state-based reaction to events. However, there are several differences between the two:

- `gen_fsm` does not provide the option to block events.
- `gen_fsm` does not deal with coordinating several instances.
- `gen_fsm` does not distinguish between events that are waited for and other events. It will handle any call to `gen_fsm:send_event`.

One can view the `bp` module as an extension of `gen_fsm`, designed for coordinating several processes.

The synchronization and event selection driven by the `bp` module uses and hides the robust underlying Erlang message passing, thus creating the higher level of abstraction in the form of behavioral events. Such hiding of the underlying messaging is done also in many other Erlang generic modules.

In addition to the general capabilities and broad usage of Erlang in concurrent processing, particular attention to programming independent behaviors in Erlang can be observed in systems such as ERES rule-production [45] or eXat agent programming [44]. What distinguishes b-thread synchronization in our pattern from the classical programming of concurrent behaviors in Erlang is the ability of one process to prevent the occurrence of an

event requested by another process, without any party being explicitly aware of even the existence of the other.

3. Application Designs for Dealing with Asynchrony and Different Time Scales

In this section we propose design patterns for dealing with the synchronization challenges arising from the distributed orientation of Erlang, and which would exist in other languages too. The solutions are mostly adapted from established techniques and designs — some general, e.g., agent-based programming [30], and some specific to BP, notably including the Interplay method [6] for connecting multiple LSC Play-Engines via external events. Our goal is to benefit from the ability to carry out natural and incremental development in BP, while accomplishing a level of decentralization, and possibly physical distribution, where not all behaviors need to be fully synchronized.

3.1. How short is your zero time?

The collective-execution mechanism of BP calls for the synchronization of all b-threads prior to triggering an event. This, combined with event blocking, facilitates the incrementality and the ability to handle different aspects of behavior separately, as afforded by BP. However, such synchronization implies that system performance is constrained by its "weakest link"; i.e., by the time it takes, at each step, for the slowest behavior to reach the synchronization point.

Thus, In the BP design pattern there is a convention that b-threads are allowed to take only a small amount of time between synchronization points. This is a rather standard convention. It is used, e.g., for listeners in GUI frameworks and interrupt handlers, where callback routines are expected to complete quickly. Also, in the *logical execution time* (LET) approach proposed in [28], the design pattern relies on a "fast execution" assumption for interfacing with a physical environment in a deterministic manner.

The question of how small the processing time should be may sometimes be crucial. In way of dealing with this issue, and since smallness is relative, an application may contain behaviors that operate at different time scales. Examples include (1) handling external events whose source is not synchronized with the application; (2) interacting with (and waiting for) resources whose response time (including communications) is slower than the internal event-rate of other behaviors; and (3) a physically distributed multi-agent application, where constant, mandatory, full synchronization is counter-intuitive or impractical. Implementing solutions for these cases in the context of BP is exemplified in Sections 3.4, 3.5, and 3.6, respectively.

A related, and often-asked, question regarding BP is: "what happens if a b-thread *never* reaches its next synchronization point, thus stalling the entire application?".

This is indeed is a real concern. However, there are several approaches to addressing it. One kind of answer is closely related to the architecture proposed in Section 3.2 for handling external events: design the b-threads so that they do not wait for external, non-behavioral events, and do not perform heavy calculations, thus eliminating a major source

of potential delay, and, of course, run the system on an infrastructure that guarantees sufficient allocation of processor resources to all b-threads. Another approach (elaborated upon in Section 3.3) is based on dividing the application into groups (nodes) of synchronized b-threads that communicate with each other only asynchronously, thus limiting the inter-b-thread dependencies. Yet another partial solution to the problem of indefinite delay in b-thread synchronization comes from the relaxed synchronization design approach described in 4. Still, we should remember that a situation where a b-thread never reaches (or is significantly delayed in reaching) its next synchronization point is often a "normal" application bug like any other. The situation may be compared to that of a standard (non-BP) application, where an infinite loop in a high priority process causes it to monopolize all system resources, stalling other processes. Tools like the model checker presented in [24] and the trace visualizer presented in [16] can help in catching such bugs.

3.2. Handling external events

The external environment is an essential element in the design of reactive systems. It is the source of the events that the system needs to react to, and is the target of the actions taken by the system. The actual occurrence of events in the environment, e.g., temperature rising, cannot be synchronized with the processing done by the system, and the arrival of such events, e.g., when and how a sensor reading is processed, requires special handling. In our case, for coordinating the synchronized processing of BP with an external environment, we propose a solution based on the *super-step* approach, as in statecharts [22] (and which is used also in, e.g., LSC execution [21]) and on the logical execution time concept used, e.g., in GIOTTO [28].

This solution is implemented by the BP infrastructure, assuming only that the application complies with simple rules, without application-specific programming. Specifically, we propose that behavioral program execution be divided into cycles, called *super-steps*, and external events are introduced only at the beginning of a super-step. The philosophy is that all internal events in the body of a super-step are perceived as happening in the same physical time unit but they are nevertheless ordered; i.e., there is a sequence of events (not associated with meaningful time-stamps) between the beginning and end of each super-step (c.f. hybrid time-set [37]). Thus, they are viewed as all taking place in zero time.

According to this convention, it is sufficient to consider a real-world occurrence of an external event only in relation to when super-steps begin and end; i.e., only external events have meaningful time-stamps. When sampling times need to be equidistant, as is the case when implementing algorithms for certain kinds of real time systems and in control theory, one can program the system such that all super-steps take a constant amount of time, by adding delays (under the assumption that the super-step's internal events runs fast enough to always complete within the time frame).

One initial approach to the implementation of the above philosophy is to have a non-behavioral process handle the external event by dynamically creating a lowest-priority b-thread that will join the next synchronization point and request a corresponding behavioral event. While this approach is general and simple, it may — depending on the

implementation platform and language — present performance issues related to the creation of threads and processes.

We thus propose the following design:

- External events are first captured by non-BP processes and are placed in a common queue, using standard programming constructs.
- A lowest priority b-thread repeatedly requests a predesignated event, called, say `idle`, which by convention is never blocked by other b-threads (which can be enforced). Since this event can occur only when there are no other events that are requested and not blocked, its triggering marks the end of a super-step.
- A designated b-thread repeatedly waits for the event `idle`, and then "peeks" at the external event queue using standard programming constructs. This peeking may be slightly delayed, as the queue may be temporarily locked by other processes during an atomic operation. However, this kind of delay is acceptable, since no internal events other than `idle` can become enabled in the next synchronization point. If no new external event is found, the b-thread proceeds to its next iteration where it waits for another `idle` event. If an external event is found, the b-thread requests a corresponding behavioral event that represents the external one, and proceeds to its next iteration, waiting again for the `idle` event.

Actually, the first and third points above can be replaced by allowing a single b-thread to wait for an `idle` event, and then, wait for external events using standard (non behavioral) language constructs, such as `receive` in Erlang or `wait` in Java. This will in fact stall the process and cause the next synchronization point to be delayed until the waiting process also synchronizes, which will happen only after the next external event arrives. It does not stall the system's operation, however, since by convention no internal events can be triggered between the `idle` event and the next external event. The three-point design above allows a richer variety of actions in separate b-threads, such as super-step logging, deadlock handling, blocking of external events, and additional peeking at external event queues, while replacing the first two points as suggested may be simpler and more suitable for small applications.

This approach, based on logical execution time, also explains why the BP semantics does not allow for simultaneous triggering of multiple events. On the one hand, allowing only a single event to be triggered at each synchronization point makes it possible for every b-thread to react to the triggered event and change states as needed; e.g., for subsequent requesting, waiting, or blocking events. Debugging and formal analysis of this semantics is much simpler than when simultaneous triggering is allowed. On the other hand, event sequences that do not require such strict sequencing, possibly benefiting from simultaneous event triggering, do not generally suffer from the sequencing, as all the events indeed occur within the same time slot, marked by two time ticks or two other external events. The relaxed synchronization techniques described in Section 4 also assist in enabling the desired parallel execution, when applicable. We should also note that when, for some reason, simultaneous event triggering *is* desired, it can be partially implemented in application-specific ways by enriching the event

objects to represent sets of more primitive events. Finally, if desired, an implementation of BP can easily allow for simultaneous event triggering, for example by triggering all (or some) of the events that were requested and not blocked, notifying all b-threads that requested or waited for any of them, and returning to each of the notified b-threads either the entire set of triggered events or just those relevant to it.

3.3. Accommodating behaviors at different time scales

Many applications that include behaviors of different time-scales can be decomposed as follows. The application is divided into groups of behaviors, with all behaviors in a group being on the same time-scale. If desired, such groups can be further broken down into sub-groups; e.g., by physical components or specific relevant events. Note that the behaviors in each such sub-group are, by definition, of the same time-scale. In the context of BP we call such a group of behaviors a *behavior node*, or *b-node*, for short. The work shown here is a continuation and generalization of the InterPlay tool that was designed to allow coordinated execution of multiple LSC and statechart specifications [6].

All the b-threads in each b-node run in a synchronized manner, as described in Section 2. In actuality, the b-threads in a b-node may run on multiple cores or, when the time scale allows, on multiple computers, where the execution environment (e.g., native operating system, JVM, Erlang, etc.) facilitates the constant inter-b-thread synchronization implemented in the BP collective-execution mechanism. Here we focus on *inter*-b-node coordination and not on *intra*-b-node parallelism.

Each event generated by the environment is processed by a designated b-node, as described in Section 3.2. Communication between b-nodes is carried out only through external events, which can be sent and received by all behavioral programs in a consistent and standard way, as follows:

- **Send**: In each b-node, a designated b-thread waits for certain internal behavioral events, and then transmits a corresponding external event to the desired destination, or broadcasts it to all other b-nodes using some (non BP) protocol.

- **Receive**: Per the design in Section 3.2, for each b-node, a designated non-behavioral process listens to all external events directed at that b-node, and places them in a single queue associated with this receiving b-node. A designated b-thread in this b-node peeks at the queue at the end of each super-step and requests a corresponding behavioral event. B-threads that depend on external events are programmed (behaviorally) to wait for the corresponding behavioral event.

Note that the result of an incoming external event could be that some of the b-node's b-threads decide to block certain events internal to the node, until some specific other external event arrives. This allows one b-node to cause the blocking of events in other b-nodes. Since blocking is central to the incrementality afforded by BP, the ability to propagate event-blocking is an important feature of the proposed decentralized architecture.

This design for communication between behavioral programs reflects several choices that we believe are common and natural in development situations. First, consider a b-node busily

working autonomously. When an external event arrives, which is expected to change the behavior of the node, it is acceptable that one or more events of the b-thread's autonomous behavior be triggered before the new course of action is taken. This form of inertia is commonly observable not only in the physical world and in typical human handling of interrupts ("please just let me finish sending this email, and I'll be right with you"), but also in the delays tolerated when sensing events or handling interrupts in computer systems.

The second choice is that even a node that wishes to be extremely attentive to external events should not be synchronized with the source of those events, the way, e.g., that b-threads are synchronized within a b-node. Consider a corporation, and a manager-employee analogy for the relationship between two b-nodes. Employees who follow their managers everywhere in order to be ready to respond quickly to new requests may be inefficient and disruptive. Put differently, excessive synchronization between b-nodes can greatly increase the computational overhead in the system. We believe that the proposed design nicely balances the use of autonomous behavioral components with efficiency. In particular, we expect that the delay in reacting to messages will be tolerable in a computer application that is properly decomposed according to behaviors (similarly to the case of a corporation).

We believe that this combination of (1) synchrony within a b-node, (2) asynchronous communication between b-nodes, and (3) translation of asynchronous messages and events back into behavioral events in each receiving b-node, allows one to retain the natural and incremental application development offered by BP, while overcoming some of the performance constraints associated with synchronizing many b-threads.

3.4. Example 1: Synchronizing with an external environment

To demonstrate how a behavioral program handles external events, we describe the architecture and implementation of a modest computer game. Specifically, we decompose the game into separate, independently programmed behaviors (all in a single b-node), and show how external events are introduced at the end of super-steps.

In this game, the goal is to land a rocket which descends vertically from the top of the screen back on its landing pad which is located on the ground, and moves left or right at random. The player can nudge the descending rocket left or right (but not upwards), and can fire a short exhaust burst to delay the fall. The player wins if the rocket ends up on the landing pad, and loses if it hits the ground. Figure 4 shows the player interface of the game.

3.4.1. Game architecture

We implemented this game in Erlang, using WxErlang as the GUI toolkit, and the `bp` module as the behavioral programming library. The application consists of the following processes:

- The `rocket` and `landing_pad` maintain and draw the positions of the rocket and the landing pad as determined by rocket and pad motions.
- A user-interface controller, implemented by extending the generic module `wx_object`, reacts to the player's clicking rocket-control buttons by sending native (not behavioral) events `user_right`, `user_left` and `user_up` to the `queue` process.

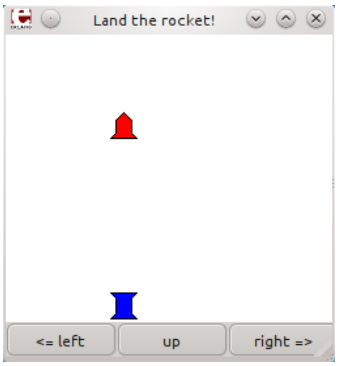


Figure 4: The computer game. The rocket, shown on the top part, should land on the pad, shown at the bottom. The player may move the rocket sideways, or fire a short burst that delays the rocket's descent.

- The `ticker` sends the native event `tick` to the `queue` process every N milliseconds.
- The `idler` b-thread detects the termination of super-steps by repeatedly requesting the `idle` event, while running at the lowest priority.
- The `queue` b-thread waits for the `idle` event, reads (using `receive`) its native message mailbox (waiting for a message if the queue is empty), and broadcasts the external event to the other b-threads by requesting a corresponding behavioral event using `bp:bSync`. This is done with the loop:

```
dispatch_loop() ->
   bp:bSync(#rwb{wait=[?IDLE]}),
   receive E -> bp:bSync(#rwb{request=[E]}) end,
   dispatch_loop().
```

- B-threads like `on_tick` or `go_left` enforce the game rules and control the actual movement of the rocket and the landing pad. These are described in further detail below.

These processes are depicted in Figure 5.

3.4.2. Game behaviors

Each behavior matches a single requirement, and is implemented as an Erlang function. The behavior functions are spawned to create the various b-threads. For readability, we tried to keep the code of the functions simple and straightforward. All the events are atoms. Each function fulfills a single role. We deliberately avoided generalizing several similar functions into a single parameterized function, in order not to burden the reader with a non-trivial design. Functions that were similar to ones listed below are omitted, in favor of a textual description.

The first behavior reacts to events representing the user's actions by requesting events representing rocket moves in the desired direction. For example, it translates the event

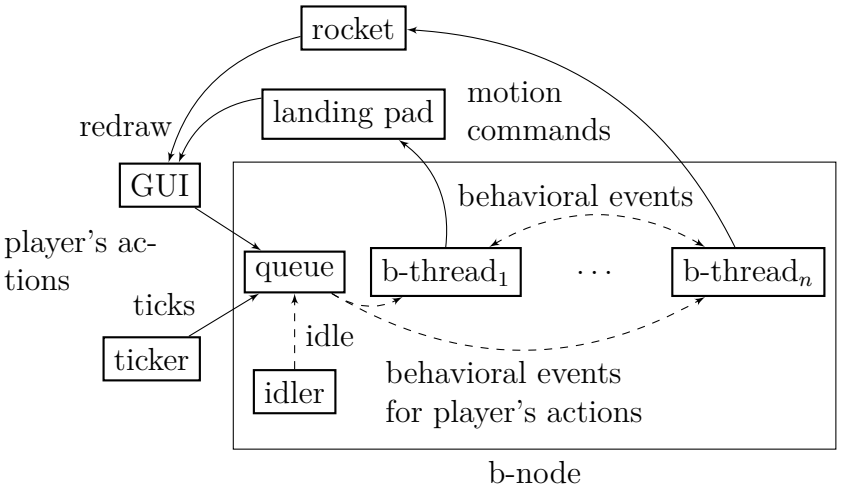


Figure 5: Architecture of the rocket game application. Solid arrows mark native Erlang messages. Dashed arrows mark triggering and waiting for behavioral events. Boxes are processes. The processes inside the b-node box are b-threads.

`user_left`, to `left`. (In the general case the translation may be more complex, e.g., requesting several events per user command.)

The following set of behaviors is responsible for actuating the movement of the rocket. The `go_left` b-thread describes how the rocket is moved to the left. The scenario repeatedly waits for the behavioral event `left`, and once triggered (i.e., the operation is allowed), it calls the function `rocket:left()`. Moving right or down are similar.

```
go_left() ->
   bp:bSync(#rwb{wait=[left]}),
   rocket:left(),
   go_left().
```

The `on_tick` behavior below repeatedly waits for a tick and then requests the event representing the rocket moving down, while also waiting for `tick` events in order to detect the case where the `down` event could not be triggered.

```
on_tick() ->
   bp:bSync(#rwb{wait=[tick]}),
   bp:bSync(#rwb{request=[down], wait=[tick]}),
   on_tick().
```

In addition to moving the rocket sideways, we also want to allow the player to simulate an exhause burst that delays the fall, by suspending the movement for one turn. The `go_up` behavior responds to the `up` event, and suspends the rocket by blocking its downward movement until the next tick.

```
go_up() ->
   bp:bSync(#rwb{wait=[up]}),
   bp:bSync(#rwb{wait=[tick]}),
   bp:bSync(#rwb{wait=[tick], block=[down]}),
   go_up().
```

We would now like to add some restrictions to the game. The following behavior prevents the rocket from moving beyond the left-hand side of the game board.

```
on_left_bound(X) -> % X: rocket's horizontal location
   if
      X == 0 ->
         bp:bSync(#rwb{wait=[right], block=[left]}),
         on_left_bound(1);
      true ->
         case bp:bSync(#rwb{wait=[left, right]}) of
            left  -> on_left_bound(X-1);
            right -> on_left_bound(X+1)
         end
   end.
```

When the rocket is at the left bound it blocks the `left` event until the rocket moves right. Otherwise the b-thread tracks the rocket's location based on the `left` and `right` events it observes.

The right-hand border is similar, with the 0 coordinate being replaced by some limit M, and changing event names accordingly. Preventing the rocket from moving below the bottom border is done by checking the vertical, rather than horizontal, position of the rocket and blocking the `down` event when 0 is reached.

To make the game less trivial, we want to prevent the rocket from making too many maneuvers. To this end, we limit its movement to at most one step left or right per turn. It waits for a `tick` and any move. After a move, other moves are blocked until a `tick` occurs again. Once a `tick` occurs, both moves are re-allowed (the blocked events set is empty).

```
block_mult_moves(Block) ->
   Moves = [user_left, user_right],
   case bp:bSync(#rwb{wait=[tick|Moves], block=Block}) of
      user_left -> block_mult_moves(Moves);
      user_right -> block_mult_moves(Moves);
      tick -> block_mult_moves([])
   end.
```

The behaviors that control the movement of the landing pad are similar to the ones handling the rocket, with one difference: the landing pad is not controlled by the player, but moves at random. After each tick, it requests that the pad be moved in a random direction, or not at all. This behavior is achieved by picking a random element `E`, that is `pad_left`, `pad_right` or an empty list, and calling `bp:bSync(#rwb{wait=[tick], request=[E]})`, followed by waiting for a tick if the pad moved before the end of the turn.

Finally, the following behaviors deal with the winning or losing. Note that the `detect_win_lose` function below keeps track of the entire game board solely by listening out for movement events. This redundancy with the tracking of rocket and landing pad positions in the b-threads `rocket` and `landing_pad` might seem wasteful, and one may prefer to encapsulate this functionality. However this design prevents potential race conditions associated to accessing shared resources (see additional discussion of race conditions in Section 6). This choice reflects the preference, in BP, to have b-threads depend as much as possible on events that are meaningful in the overall external behavior of the system, as opposed to requiring specialized inputs from internal components. Nevertheless, it may be fully desirable and appropriate to create such special events, by a central service, and broad-

cast them for use by all interested b-threads. This can be done for object tracking in our case, and especially, for example, when communicating exact positions that are determined and updated by a GPS.

```
detect_win_lose(P, X, Y) ->
   case bp:bSync(#rwb{wait=[pad_left, pad_right, left, right, down]}) of
      pad_left -> detect_win_lose(P-1, X, Y);
      pad_right -> detect_win_lose(P+1, X, Y);
      left -> detect_win_lose(P, X-1, Y);
      right -> detect_win_lose(P, X+1, Y);
      down ->
         Y1 = Y-1,
         if
            Y1 == 1 andalso X == P ->
               bp:bSync(#rwb{request=[win]});
            Y1 == 0 ->
               bp:bSync(#rwb{request=[lose]});
            true ->
               detect_win_lose(P, X, Y1)
         end
   end.
```

Another b-thread (not shown) ends the game by waiting for the `win` or `lose` events, and then blocking all movement events indefinitely. The program ends when the user closes the application window.

Summary. We have illustrated an application being decomposed into b-threads, using the super-step idea, where external events were triggered as behavioral events after all the internal events of a super-step were completed. The power of BP in handling rich scenarios can be further demonstrated in this example, by replacing the short b-thread implementing the random move of the pad by a lengthy sequence of predetermined right- and left-moves of the landing pad, which represent some covert plan of an adversary played by the computer.

For simplicity, this example is implemented in a single b-node. In a richer game, components like the rocket and the pad could be programmed in separate b-nodes. Nevertheless, note that some of the game rules do require synchrony between different behaviors with regard to time ticks, and this was readily implemented in the single b-node setup.

3.5. Example 2: Coordinating behaviors with different time scales

In this section we demonstrate how an application can be composed of behavioral components, namely b-nodes, that operate on different time scales and communicate via events. The example is part of the control software for a quadrotor, a flying vehicle powered by four rotors (see schematic drawing in Figure 6). The behaviorally-programmed piece is responsible for stabilizing the aircraft. It was experimentally tested by plugging behavioral modules written in Java into the comprehensive quadrotor-control simulation model developed by Bouabdallah et al. [9, 10]. This model is based on MATLAB/Simulink and simulates full control of the quadrotor flight including physical aspects. A high-level description of this use-case appeared also in [27].

The model of Bouabdallah et al. is modified, as illustrated in Figure 7. The b-node High-Level Control translates measured differences between actual and desired flight parameters (e.g., orientation, altitude) into four forces (thrust, roll, pitch and yaw — see Figure 6). It

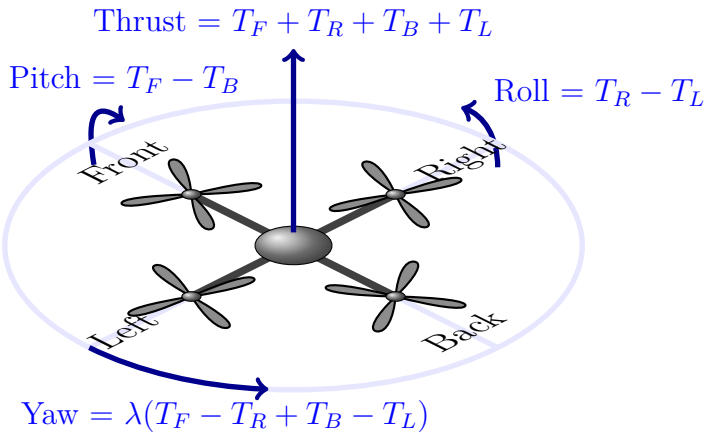


Figure 6: A schematic view of a quadrotor. Four rotors are mounted rigidly in a single plane and control is achieved only by varying their speeds. The designation of rotor labels is arbitrary, and the rotational forces of roll, pitch, and yaw are defined relative to them. The thrusts generated by the rotors are denoted by T_F, T_R, T_B and T_L, for the front, right, back, and left rotors, respectively. The relationships between these thrusts and the forces are indicated as the four formulas appearing in the figure near each force. The coefficient λ marks the ratio between the vertical thrust of a rotor and the rotational (yaw) force that the rotor generates (neighboring rotors rotate in opposite directions to balance the yaw force when all rotate at the same RPM).

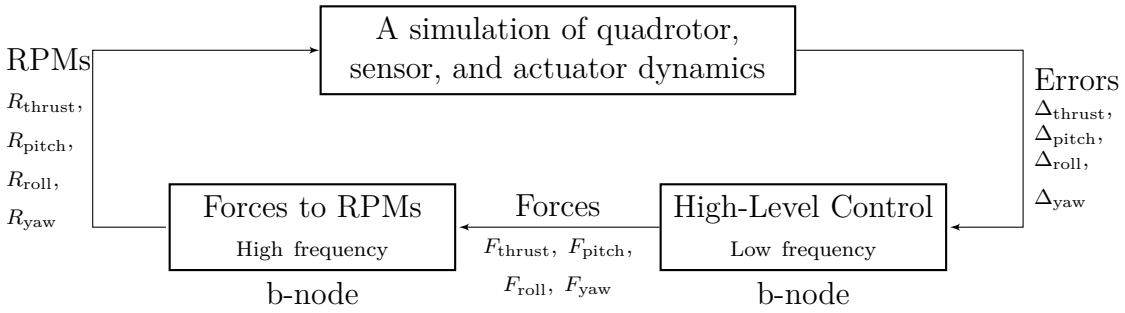


Figure 7: Block diagram for the quadrotor example. A controller for the quadrotor is composed of two behavioral components. The first, High-Level Control, takes the differences between actual and desired thrust, pitch, roll and yaw and dynamically translates them into the forces that have to be applied in order to correct the displacements. The second component, Forces to RPMs, translates these forces into the rotor speeds (RPMs) that will generate them. The behavioral program that we developed for this paper consists of the two b-nodes on the bottom, their interaction with the environment, and the communication between them.

consists of four b-threads: one for each of the yaw, pitch and roll axis and one for thrust. Each of these b-threads computes the respective force by applying a standard Proportional Integral

Derivative control based on the differences between measurements and desired value, as obtained, e.g, from the remote control or as dictated by a higher level navigation algorithm.

The b-node 'Forces to RPMs' operates at a higher frequency than the 'High-Level Control'. This component receives the four desired forces as external input events and translates them into rotor RPMs, via a fast coordinated sequence of events, where different b-threads balance the competing needs. For illustration, the reader may compare the action of our b-thread with the way sound-mixing is where a sound engineer deals with competing goals, such as balancing the guitar and the piano, and controlling overall volume by small adjustments of the available controls (without attempting to solve complex mathematical equations).

In the 'Forces to RPMs' b-node, each of the four forces is independently interpreted by a dedicated b-thread as a target, and is translated to desired changes to RPM of the various rotors. The b-threads attempt to attain their targets by requesting events that represent (small) increases or decreases of individual rotor RPM towards the target and, blocking events that work away from it, as shown in Figure 8. The composite behavioral execution mechanism interlaces the execution of all b-threads, transforming their possibly-conflicting requests into integrated control.

For example, to increase the pitch (raise the front), the system requests the events of increasing the front rotor's RPM and decreasing the back rotor's RPM, while blocking the opposite (complementary) events. Note that desired flight results depend only on the total numbers of events that increase or decrease the controlled value, and the difference between their numbers, while the actual mix of events may vary. Specifically in this example (see Figure 8), in order to allow finer control, the b-threads translate the input force into a sequence of events that affect the rotor speed by smaller increments. Furthermore, to accommodate behaviors that require concurrent actions on multiple rotors, all behaviors entertain a "slack", in which they allow a small number of undesired events to occur before blocking them completely. In this manner, the `ThrustBT` can increase the RPM of all rotors equally, one at a time, without interference from b-threads that try to create a difference between certain rotor RPMs.

In our simulation, the super-step is orchestrated as follows. A lowest-priority b-thread repeatedly waits for RPM-change events, as described above, and keeps track of all desired rotor RPMs by changing in-memory values by fixed amounts. This b-thread repeatedly requests an `output` event that contains those values, but due to its low priority, `output` is triggered only when there are no other events to trigger (i.e., the four force-control b-threads have attained their targets). The `output` event marks the completion of a super-step. Four inputs (target forces) are then presented again (by a high priority b-thread that polls an external queue), marking the beginning of the next super-step. Throughout, an actuator b-thread waits for `output` events and transmits the required signals to the rotors. The detailed simulations we carried out show that his behavioral solution indeed stabilizes the quadrotor, as can be seen in Figure 9.

Subsequently, we have begun to implement a behavioral programming environment written in C that can run directly on a quadrotor, and have shown that the design and the b-threads shown above indeed can stabilize a UAV. In fact, initial versions of this are already flying in our lab.

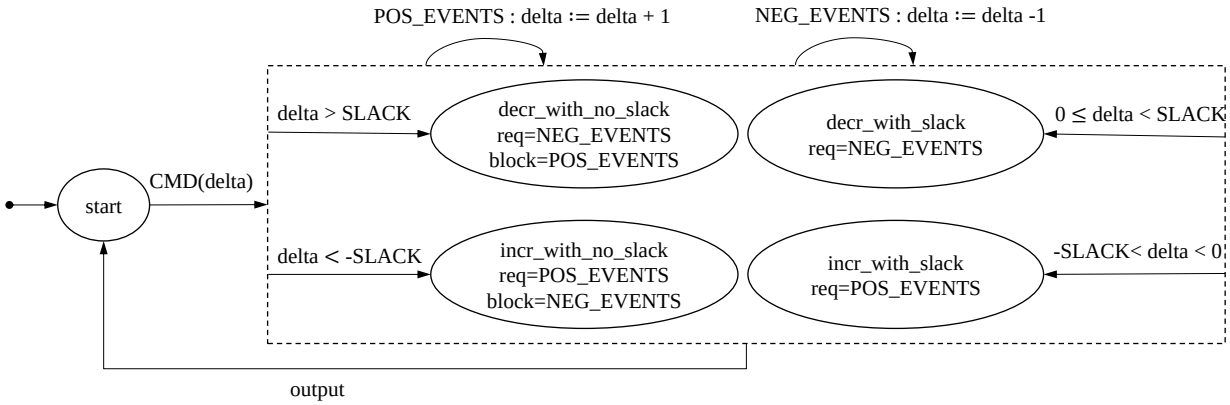


Figure 8: A template for the `yawBT`, `pitchBT`, `rollBT`, and `thrustBT` b-threads, written in a parametric manner. Each b-thread is instantiated with different values of the parameters: `CMD(delta)` indicates a command to change the force that this b-thread is responsible for (i.e. `CMD` is `ChangeYaw` for `yawBT`,`ChangePitch` for `pitchBT`, etc.). `POS_EVENTS` and `NEG_EVENTS` are the events that increase and decrease, respectively, this force. For example for pitch (and thus for `pitchBT`), `POS_EVENTS` is {`FrontRPMAdd`, `BackRPMSub`}. `SLACK` defines a range close to the b-thread's target where the b-thread allows the occurrence of events that conflict with its goal to allow for other b-threads to work towards their goals. We use statechart-like notations to allow for a compact and readable diagram. The initial state is `start` where the b-thread waits for a `CMD(delta)` event. It then transitions into one of the four other states depending on the sign of the change in force and whether it is within the slack value or not, as marked on the arrows going into the states. If the desired change in force is positive, and its absolute value is larger than `SLACK` then the b-thread requests the positive events and blocks the negative events. If the desired change is negative, the b-thread requests the events that decrease the force, and blocks the ones that increase it. If the absolute value of `delta` is smaller than `SLACK` the same events are requested, but no events are blocked. Once an event occurs, regardless of the state, the transitions on the container state show that the value of `delta` is modified according to the event, and the b-thread transitions into a new state according to the new conditions. When no events are requested by any of these b-threads an `output` event requested by another b-thread (not shown) is triggered transferring the actual RPM speeds to actuator b-threads, and all four b-threads in this diagram transition back to their initial state, waiting for the next command. The remaining `POS_EVENTS` are {`RightRPMAdd`, `LeftRPMSub`} for `rollBT`, {`FrontRPMAdd`, `BackRPMAdd`, `RightRPMSub`, `LeftRPMSub`} for `yawBT`, and {`FrontRPMAdd`, `RightRPMAdd`, `BackRPMAdd`, `LeftRPMAdd`} for the `thrustBT`. The `NEG_EVENTS` are the complementary ones.

When proceeding on the actual implementation in a flying quadrotor, additional considerations of course emerge in order to meet the real-time constraints. The first and foremost consideration for BP in this regard is having efficient processing at the underlying BP infrastructure, controlling the synchronization of b-threads, collection of their declarations,

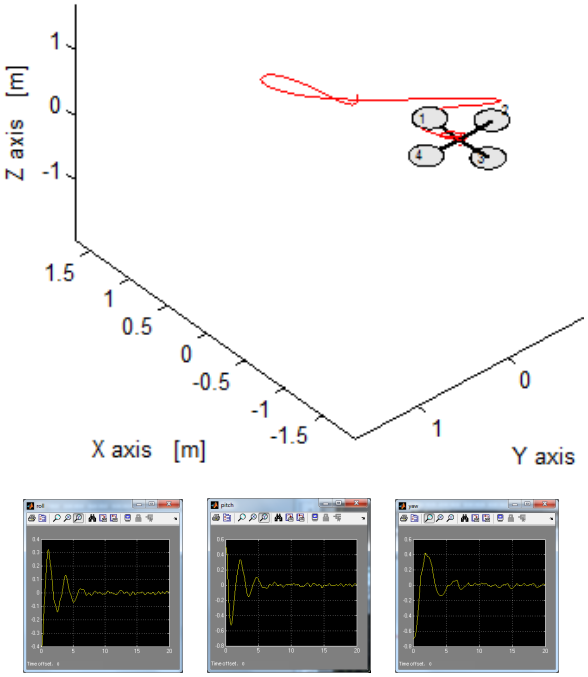


Figure 9: Simulated spatial path of the quadrotor, and plots of roll, pitch and yaw during the first 20 seconds of a behaviorally-controlled flight, as generated by the MATLAB model and tools of Bouabdallah et al. [9, 10].

event selection, and b-thread resumption. One such approach is to turn events into bits and event sets into easily managed bit vectors. Then come the additional real-time considerations of ensuring that the infrastructure and application allow the non-behavioral sensor and actuator processes to interact with the real world at the desired frequency, and that the b-threads of system behavior are efficient and predictable in completing their reactive scenarios before arrival of the next environment-driven event. For example, towards predictability, in our experimentation on the real quadrotor, we initiate the `output` event at a fixed time frequency, regardless of whether the b-threads have completed their negotiations over the right changes to all RPM values or not. Both the BP infrastructure optimization and more comprehensive design guidelines regarding real time behavioral applications are the subject of future research and are outside of the scope of the present paper.

Summary. In creating composite behavior, the quadrotor application uses very local behaviors, such as controlling roll and pitch, to achieve higher level goals such as maintaining stability. Longer term behaviors, such as traveling between stations in a multi-stop trip, or keeping maintenance and refueling schedules, can be constructed from similar elements. While all these facets can be programmed as compositions of b-threads, gluing them together is easier with an infrastructure that can support multiple time scales. Clearly, a

b-thread that controls a multi-stop navigation itinerary can suffice with occasional changing of the required speed and direction, and does not require the constant synchronization and attentiveness of the stabilization modules described above.

The quadrotor application also shows how behavioral programming can be used in developing hybrid control, i.e., a combination of discrete logic and actuation with the sensing of continuous system variables [36]. In [26] we discuss the application of BP to hybrid control, by using fuzzy logic to translate continuous information into discrete categories that can be refereed to in naturally-programmed scenarios.

3.6. Example 3: Incremental development of a multi-agent application

In this section we demonstrate how multiple behavioral programs coordinated only by external events can retain much of the incrementality of a single fully synchronized behavioral system, i.e., that functionality can be enhanced by adding b-threads and b-nodes.

We present the development of a multi-agent application, where agents include vehicles traveling in open terrain and advisors affecting the vehicles. First, we list the behaviors that are responsible for the movement of a vehicle. To each vehicle we then add behaviors that allow it to interact with external systems. Finally, again incrementally, we program the vehicle agents to react to an advisor agent responsible for directing particular aspects of the vehicles' travel. The communication between the vehicles and the advisor uses the architecture for external events described in Section 3.2. This small example can be readily extended to make it easily possible to add agents (vehicles, advisors), as well as new kinds of interactions among existing agents (inter-vehicle or inter-advisor).

3.6.1. Vehicle motion

For brevity, we list only a minimalistic scenario: moving towards a given goal. The movement is performed by the following processes. A `ticker` process sends an external `tick` event every N milliseconds, as in the example of Section 3.4. The b-thread `on_tick` (not shown) repeatedly waits for this event, samples the current position, say, using a GPS, and broadcasts the position by requesting the event `{pos, X, Y}`. This event also captures a form of feeding external information into the behavioral system. The behavior `head_north` repeatedly compares the current position with the location of the goal, and decides whether to request the event of moving north or not. Similar behaviors are responsible for moving south, west and east, all quite obliviously of each other. The `is` function matches all events of a given record type (all `pos` records in this case).

```
head_north() ->
   {pos, _, Y} = bp:bSync(#rwb{wait=is(pos)}),
   {_, Y1} = goal(),
   if
      Y < Y1 ->
         bp:bSync(#rwb{request=[north]}),
         move_north(); % Request granted, move north
      true -> ok % Don't move north
   end,
   head_north().
```

3.6.2. Enabling external communication

The behaviors listed above will cause the vehicle to move towards its goal uninterrupted. We would like to allow an external coordinator agent to affect this movement, but, unlike the game and quadrotor examples, here there is no natural super-step breakpoint within the internal events where the external environment can intervene. To modify the vehicle software for creating such a breakpoint we add two b-threads. One counts N_1 advancement steps of the vehicle and then sends an external event containing the current position of the vehicle to an external process.

```
report(S, P, N1) ->
   {pos, X, Y} = bp:bSync(#rwb{wait=is(pos)}),
   S ! {pos, P, X, Y},
   [bp:bSync(#rwb{wait=is(pos)}) || _ <- seq(1,N1-1)],
   report(S, P, N).
```

Another b-thread counts N_2 steps and peeks at the communication channel for any external incoming communication. If the channel is empty the process continues.

```
listen(N2) ->
   bp:bSync(#rwb{wait=is(pos)}),
   receive E -> bp:bSync(#rwb{request=[E]})
   after 10 -> ok % Timeout after 10 milliseconds
   end,
   [bp:bSync(#rwb{wait=is(pos)}) || _ <- seq(1,N2-1)],
   listen(N).
```

To demonstrate a possible external command, we assume that advisors may warn the vehicle, upon arriving at a certain line in its northward path, that there are too many vehicles beyond that line. To allow the vehicle to react, we add a new b-thread to the vehicle b-node. If the vehicle crossed the line, it calls the `hold` function.

```
on_warning()->
   {warn, T} = bp:bSync(#rwb{wait=is(warn)}),
   {pos, _, Y} = bp:bSync(#rwb{wait=is(pos)}),
   if
      Y >= T -> hold();
      true -> on_warning()
   end.
```

The implementation of `hold` listed below blocks the movement north, as may be requested by *any* b-thread, current or future, for 10 steps. The added behavior `on_warning` represents the policy of the vehicle's reaction to the `warn` event. It could be easily replaced by other policies, such as blocking the movement north until notified otherwise, or even to move southward.

```
hold() ->
   [bp:bSync(#rwb{wait=[tick], block=[north]}) ||
    _ <- seq(1,10)],
   on_warning().
```

3.6.3. Adding an advisor agent

We are now ready to add advisors that will monitor and try to affect the movement of several vehicles. For example, the following behaviors first report when a vehicle crosses a

line, and later ask vehicles positioned at the line to stall if there are more than N vehicles north of the line. The first advisor b-thread, `monitor_line`, deals only with behavioral events. It waits for any vehicle position event, and requests an event, internal to the advisor, that will eventually lead to inter-agent messages.

```
monitor_line(Y1) ->
   {pos, P, _, Y} = bp:bSync(#rwb{wait=is(pos)}),
   if
      Y >= Y1 -> % Report crossing
        bp:bSync(#rwb{request=[{cross, P}]});
      true -> ok
   end,
   monitor_line(Y1).
```

The second behavior listed below mixes behavioral and native communication methods.

```
monitor_crossing(N, Y1, L) ->
   {cross, P} = bp:bSync(#rwb{wait=P is not in L}),
   if
      length(L) >= N ->
         P ! {warn, Y1},
         monitor_crossing(N, Y1, L);
      true ->  monitor_crossing(N, Y1, [P|L]) % Add P to L
   end.
```

It maintains a list `L` of vehicles that have crossed the line. Each vehicle is represented by the address of its listening process. When another vehicle crosses (i.e., its address is not in the list), it checks if there are too many vehicles north of the line. If so, it sends an Erlang message to the crossing vehicle. This message is received by the `listen` behavior in the b-node that controls the vehicle. For brevity, we omit describing the full function of the `wait` clause. Figure 10 outlines the complete architecture of the vehicles application.

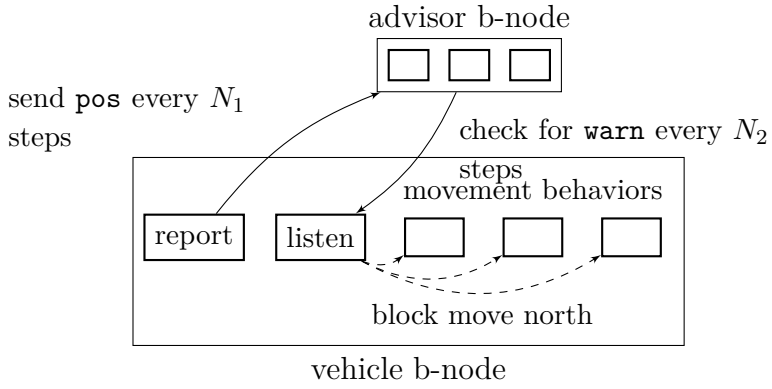


Figure 10: Multi-agent architecture for vehicles. The behavior `report` of each vehicle sends the current position to interested advisors every few steps. The `listen` behavior checks for a new warning, and may block specific events if it receives one.

Summary. In this example we demonstrated the construction of different b-nodes, each comprised of a set of constantly synchronized b-threads. The communication between the b-threads is carried out according to the methodology discussed in Section 3.2. All agents

are incrementally programmed to communicate, to interpret each other's messages and to react as desired by the developer, only by the addition of new, independently programmed b-threads.

4. Relaxing Synchronization Constraints through Eager Execution

In this section we propose an extension of BP that is an alternative to the one proposed in Section 3, termed *eager execution*. The idea is to achieve benefits similar to those afforded by using b-nodes and external event queues, but in a way that is automated and concealed from the user. In particular, programs that were written using the traditional execution mechanism can be run without any change under the eager execution mechanism, and this may result in improved performance.

At the core of the eager execution approach is the realization that it is often possible to predict the outcome of a synchronization point even without waiting for slower threads to synchronize. Through this, we may tackle the difficulties imposed by multiple time scales, asynchronous input or a distributed setting. Deciding which events may be triggered while some threads have not yet synchronized is achieved by using (at run time) information about these threads, which was generated automatically before the run. For example, if all unsynchronized threads are known to never block a particular event, and the currently synchronized threads request and do not block this event — then it can immediately be triggered, without waiting for the rest of the threads to synchronize.

In the remainder of this section we rigorously define and characterize the eager execution mechanism. For a comparison between the strengths and weaknesses of eager execution and the approach presented in Section 3, see Section 5.

4.1. Relaxing Synchronization Constraints

As discussed in Section 3.1, a behavioral program's execution speed is constrained by its slower threads, because the behavioral execution mechanism needs to receive the declarations of requested and blocked events of all threads before triggering an event. However, in some cases these constraints may be relaxed, as we now demonstrate.

Let P denote a set of threads that constitute a behavioral program, and assume that at some point during P's execution a subset $P_{\text{sync}} \subset P$ of the threads has reached a synchronization point, while the rest are still executing. Further, assume that the execution mechanism has additional information about the events that the threads in $P \setminus P_{\text{sync}}$ will request and block when they synchronize. If, combining the information from threads in P_{sync} with the information about threads in $P \setminus P_{\text{sync}}$, the execution mechanism can find an event e that will be enabled at the highest priority when all threads have synchronized, then e can immediately be chosen for triggering.

The execution mechanism may then pass e on to the threads in P_{sync} to let them continue their execution, without waiting for the remaining threads to synchronize. Once any of these other threads reaches its synchronization point, the execution mechanism immediately passes it event e, since e was the event selected for that particular synchronization point. This is accomplished by having a designated queue for each of the b-threads, of events that

are waiting to be passed, and putting e in the queues corresponding to the as-of-yet not synchronized threads. The execution mechanism described is *eager*, in the sense that it uses predetermined information to choose the next event as early as possible.

When a thread BT reaches a synchronization point, if the corresponding queue is nonempty, the execution mechanism dequeues the next pending event e'. If BT requests or waits for e', it is passed to the thread, which then continues to execute. Otherwise, e' is ignored, and the execution mechanism continues with the next event pending in the queue. In order to reflect the semantics of BP, from the execution mechanism's global perspective BT is not considered synchronized as long as it has events pending in the queue. Specifically, the events that are requested or blocked by BT at this point are irrelevant for the selection of the next event; they have already been considered when the event e', which is now at the head of the thread's queue, was triggered by the execution mechanism, ensuring that e' was a valid choice. Observe that the eager execution mechanism strictly adheres to the semantics of BP, as described in Section 2.9: at every synchronization point, the triggered event is indeed the enabled event of highest priority [2]. The key point, however, is that the eager mechanism makes its decisions more quickly, and can thus produce more efficient runs.

Note that, under certain conditions, event queues for slow threads can grow to any arbitrary length. Consequently, we propose to cap the length of these queues, based on available system resources. When the cap is reached, the system must wait for the slower threads to catch up. Hopefully, the slow threads will not have requested or waited-for many of the events in their queues, allowing them to catch up quickly.

It remains to show how the execution mechanism knows which events could be requested and blocked by threads that are yet to synchronize. We propose two approaches: one based on *static information* and the other on *dynamic information*, discussed and demonstrated in Sections 4.2 and 4.3, respectively. For a rigorous formulation of the eager execution mechanism, we refer the reader to Appendix B.

4.2. Relaxing Synchronization using Static Information

In this approach, the execution mechanism is given in advance a static over-approximation of the events that a thread might block when synchronizing. More specifically, if a thread has states $s_1, \ldots, s_n$, this over-approximation is $\bigcup_{1 \leq i \leq n} B(s_i)$, where $B(s_i)$ is the set of events blocked in state s_i. This information is static, in the sense that the over-approximation does not change throughout the run. Also, since it does not depend on the threads' specific states, this approach is unaffected (complexity-wise) by the number of events stored in the threads' queues.

When a thread synchronizes, the execution mechanism attempts to resolve the synchronization point, based on the information gathered so far. It finds the highest-priority event that is requested and not blocked by threads in P_{sync}. This event may immediately be triggered, if two conditions hold: (1) it is guaranteed to remain enabled when more threads

[2] The eager execution mechanism is tightly coupled to the event selection scheme in use — in our case, a priority-based scheme. In a preliminary version of this section, published in *LPAR* 2013, we described a mechanism for a simpler event selection scheme, in which an arbitrary enabled event is selected.

synchronize, ascertained by checking that it does not appear in the over-approximations of blocked events of any of the unsynchronized threads, and (2) it is guaranteed to be the highest-priority enabled event at the synchronization point, ascertained by requiring that all threads that have higher priorities than the thread requesting the selected event have already synchronized.

If such an event exists, it can be triggered immediately. Otherwise, the execution mechanism waits for more b-threads to synchronize. This generally results in more events becoming eligible for selection, since the actual set of events that are blocked by a thread in a given state is always a subset of the over-approximation, which includes all the events that are ever blocked by the b-thread, and since additional requested events are revealed as additional b-threads synchronize. As soon as enough information is gathered to deduce that an event is the highest-priority enabled event at the synchronization point in question, it is immediately triggered and passed to all synchronized threads.

This mechanism can be improved further. In particular, it is sometimes possible to deduce that an enabled event e is the highest-priority enabled event even when a higher priority thread has not yet synchronized. This can be accomplished by keeping over-approximations of the requested events of threads, and checking that all events that may be requested by this higher-priority thread are already blocked by synchronized threads.

Observe that we only discuss over-approximating requested and blocked events, but not their under-approximations, although these can be used analogously. The reason is that typically there are no events that are requested/blocked in each and every state of the thread, and so the under-approximations are typically empty.

4.2.1. Example: External Input using Static Information

We revisit the rocket examples from Section 3.4, and show how it can be implemented using eager execution and static information.

External input is injected into the system asynchronously from two sources: the user clicking the buttons, and the periodic `tick` events. Using static information, each of these sources of input can be handled by a dedicated lowest-priority sensor thread that (non-behaviorally) waits for them to occur, and then requests a behavioral event representing them. These two threads do not block any events, and so their over-approximation of blocked events is simply the empty set.

When the program runs, the sensor threads may delay in reaching their synchronization points; indeed, these threads can be thought of as paused until their respective external inputs are received. However, this does not prevent the rest of the threads from triggering events: as the sensor threads have lowest priority and never block any events, whenever the remaining threads are synchronized a highest-priority enabled event (assuming one exists) can be triggered.

The system can still be thought of as progressing in super-steps. At first, all non-sensor threads are synchronized, but no event is enabled. Then, one of the sensor threads synchronizes, because of a clock tick or because of a user input. Its respective requested event is triggered, and this may result in a sequence of consecutive events triggered by the non-sensor threads. Eventually, the system completes its super-step, and returns to a state

where all non-sensor threads are synchronized but no events are enabled. The system then waits for the next input.

The actual implementation of the system is quite similar to that of Section 3.4; indeed, the non-sensor threads remain unchanged. The external `ticker` process is rewritten as a b-thread, whose role is to request a `tick` event every N milliseconds to orchestrate the rocket's descent, e.g.:

```
ticker() ->
   bp:bSync(#rwb{request=[tick]}),
   sleep(N),
   ticker().
```

Observe how the eager execution mechanism allows the b-thread to sleep between synchronization points without delaying the system; this was not possible using the traditional execution mechanism. Finally, a declaration is provided to the execution mechanism, by a function call and associated message exchange, stating that the thread never blocks any events. Similarly, sensor threads for each of the user buttons, left, right and up, wait for clicks and request events `left`, `right` and `up` to signal that they have been pressed. The manually programmed queue and the thread in charge of sampling it are omitted from the implementation; they are replaced by the internal queues that the eager execution mechanism maintains for threads that it approximates — in this case, the sensor threads.

4.2.2. Modularity using Static Information

Having shown how eager synchronization and static information can be used to accommodate behavioral programs with external sources of input, we next discuss how it can also support programs with multiple time scales.

In section 3.3 we saw an example where a behavioral program included threads that ran in different frequencies. These threads could not be simply run together, lest the slower thread slow down its faster counterpart. Generalizing, we consider programs that can be partitioned into disjoint sets of threads (which we termed *behavior modules*), each handling a different facet of the system. These modules may operate at different time scales, and may be assigned distinct computational resources (e.g., they can run on different hardware). We seek to use eager synchronization to alleviate run-time dependencies between these modules.

In order to characterize the effects that eager execution has on behavior modules, we offer the following definitions. Consider a behavioral program P consisting of a set behavior modules $M_1, \ldots, M_k$; thus, the threads in the program are $\bigcup_{i=1}^{k} M_i$. Denote by E_i the set of events that are *controlled* — i.e., requested or blocked — at some synchronization point of a thread of module M_i. Typically, these events are part of the "vocabulary" corresponding to that the facet of the system addressed by module M_i. The modular design of the program is termed *strict* if $E_1, \ldots, E_k$ are pairwise disjoint; i.e., $E_i \cap E_j = \emptyset$ for $i \neq j$. However, any thread can wait for any event. A strict modular design essentially means that while modules may signal one another (by waiting for each other's events), they do *not* control each other's events; i.e., they are assigned sufficiently independent duties. Note that this requirement parallels the situation in the b-node approach, as discussed in Section 3.3.

In order to support priorities in the context of such modular designs, we must permit

b-thread priorities that are *partially* ordered (i.e., to relax the requirement that for any two distinct priorities, one is larger than the other). An enabled event e requested by thread BT is eligible for triggering if no other thread BT', with a priority that is larger than that of BT, requested a different enabled event e'. In a strict modular design we require that the priorities of threads in different modules be independent (i.e., unordered). This reflects the fact that, as the modules have sufficiently independent duties, when two events are requested by two different modules one is not preferable to the other (logically speaking). This again parallels the situation in the b-node approach, where priorities are local to each b-node. Note that since b-threads synchronize one at a time, at most one module can be ready for event selection at any given time.

The eager execution mechanism has desirable properties when it comes to behavioral programs having a strict modular design. Considering a module M_i of a strict modular design, the eager execution mechanism results in an implementation in which the threads in M_i never need to wait for a thread in another module to synchronize in order for an event in E_i to be triggered (i.e., for the sake of triggering M_i's own events). This means that as soon as a module's threads have synchronized, enabled events that are controlled by the module may be triggered immediately. Hence, modules may operate at different time scales. This property is formalized by the following proposition:

Proposition 1. *Let P be a behavioral program that has a strict modular design and is executed with the eager execution mechanism. If all b-threads of module M_i are synchronized, then an event $e \in E_i$ is eligible for triggering if and only if it is eligible for triggering upon the arrival of any other thread at its synchronization point.*

See Appendix D for a proof of the proposition.

The notion of modules is quite similar to that of b-nodes, except that inter-module communication is not performed by the application, via manually programmed external events and queues, but rather internally, by the BP infrastructure. For instance, consider the design of the quadrotor of Section 3.3, as depicted in Figure 7. An alternative design, using strict eager execution modular design, is depicted in Figure 11. Each set of threads in charge of a facet of the system — i.e., simulation, high-level control and forces to RPM — resides in a separate module, and the threads can thus process events at their respective sampling rates. For communication purposes, they can listen out for (but not request or block) events controlled by the other modules.

4.3. Relaxing Synchronization using Dynamic Information

In this approach, the execution mechanism is given complete *state graphs* of threads, which are generated before the program is executed. The labeled vertices of a state graph correspond to the thread's synchronization points and the events that it requests/blocks at these points, and the edges, labeled with non-blocked program events, describe the thread's transitions from one synchronization point to another. The graph thus provides a complete description of the thread from the execution mechanism's point of view — that is, a complete description of the events requested and blocked by the thread, but without any calculations

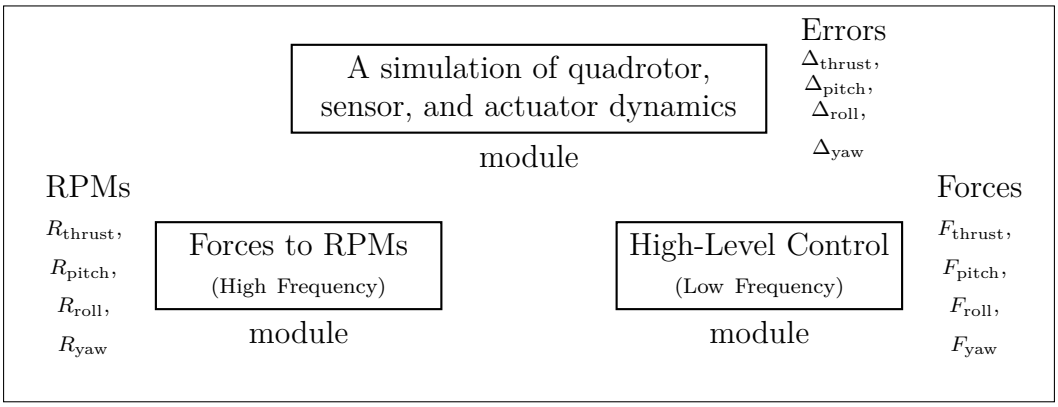


Figure 11: An alternative block diagram for the quadrotor example, implemented using eager execution and a strict modular design. The simulation module *controls* the `Error` events, the High-Level control module *controls* the `Forces` events, and the Forces to RPMs module *controls* the `RPM` events. Thus, each of the modules can trigger these events regardless of the other modules. Modules can wait for events controlled by other modules, which allows them to inter-communicate. Communication is not performed directly between the modules, but rather through the execution mechanism and by waiting for broadcasted events.

or input/output actions performed by the thread when not synchronized. For more details on these state graphs, see [25].

During runtime, the execution mechanism keeps track of the threads' positions in the graphs, allowing it to approximate the events they will request and block at the next synchronization point — even before they actually synchronize. This method is dynamic, in the sense that the approximations for a given thread can change during the run, as different states are visited. As is the case with static information, the complexity of this approach is unaffected by the number of events stored in the threads' queues. In order to make triggering decisions, it is enough for the event selection mechanism to remember the "current" state for each thread (that is — the state it would reach after processing all events in its queue), and this state can be updated every time an event is added to the queue.

We stress the fundamental difference between running a thread and simulating its run using its state graph. In the former, transitions can be considered immediate, whereas in the latter, additional non-trivial computation may be performed, and the transitions can take longer. For instance, in a recent behavioral application implementing a web-server, we used threads to calculate checksums over TCP segments and to read web pages from the disk. Both actions were time consuming, and hence the threads performing them did not immediately arrive at the next synchronization point. In such cases, simulating threads using their state graphs is faster than actually running them.

Recall that our definition of a thread dictates that its transitions be deterministic. Therefore, simulating a thread through its state graph yields precise predictions of its requested and blocked events at each synchronization point. In the nondeterministic model, where

threads may depend on coin tosses or inputs from the environment, it may be impossible for the execution mechanism to determine a thread's exact state until it synchronizes. However, the execution mechanism can approximate the thread's requested and blocked events by considering all the states to which the nondeterministic transitions might send it. If, due to a previous transition, the thread is known to be in one of states $s_1, \ldots, s_n$, then the blocked events may be over-approximated by $\bigcup_{1 \leq i \leq n} B(s_i)$ — similarly to what is done when using static approximations. Analogously, the requested events may be under-approximated by $\bigcap_{1 \leq i \leq n} R(s_i)$. For more details see Appendix C. As before, if these approximations leave no enabled events that are eligible for triggering, the execution mechanism waits for more threads to synchronize.

The other details are as they were in the static approximation scheme. Once an event is triggered, it is immediately sent to all synchronized threads, and is placed in the queues of the threads that are yet to synchronize.

4.3.1. Example: Optimization using Dynamic Information

To demonstrate the dynamic approximation mechanism, we consider a vending machine system, described next. The basic operation of the machine includes receiving coins from customers and dispensing requested products. Once every 3 hours the machine goes into a special maintenance phase: it checks its internal temperature and humidity using sensors, and, if these values are outside a safe range, corrects them using actuators. For simplicity, we assume that every measurement indicates values that are unsafe, and so every maintenance phase ends in corrective action.

The maintenance phase is implemented as follows. Once every 3 hours, a timer thread requests an `initiate_maintenance` event; using static information as described in Section 4.2, this thread does not slow the system down, and does not interfere with the dispensing of products. A `measurer` threads waits for `initiate_maintenance` events and reads the current temperature and humidity values using sensors. It then requests events indicating the required corrective action, which a `corrector` thread then performs. Code snippets appear in Fig. 12 and Fig. 13. The vending functionality of the machine is managed by threads orthogonal to the ones described; we omit the details. All threads in the system have equal priorities.

Next, consider the following requirement: due to mechanical limitations, it is forbidden to dispense products during the temperature measurement and correction phase, or the correction might be interrupted. Therefore, the `measurer` thread blocks events of type `dispense_product` (that signify the dispensing of products) during temperature measurement and correction. During humidity measurement, however, this limitation does not apply.

Measurement and correction operations take a non-zero amount of time; hence, there is a time window during the maintenance cycle in which the `measurer` and `corrector` threads are not synchronized. Under the traditional execution mechanical, this would delay any vending operations in the machine, which is undesirable; we thus seek to use the eager execution mechanism to minimize this downtime. The key point, however, is the distinction between the two phases of the maintenance cycle — as dispensing is only allowed during

```
while true ->
   bp:bSync(#rwb{wait=initiate_maintenance}),
   if temperatureTooHigh() ->
      bp:bSync(#rwb{request=decrease_temperature, block=dispense_product});
   true ->
      bp:bSync(#rwb{wait=increase_temperature, block=dispense_product});
   end,
   bp:bSync(#rwb{wait=temperature_corrected, block=dispense_product}),
   if humidityTooHigh() ->
      bp:bSync(#rwb{request=decrease_humidity});
   true ->
      bp:bSync(#rwb{request=increase_humidity});
   end,
   bp:bSync(#rwb{wait=humidity_corrected}).
```

Figure 12: The main method of the **measurer** thread. Upon triggering of the `initiate_maintenance` event this thread wakes up, asks for the appropriate temperature correction, and waits for confirmation. Afterwards, an analogous process is performed for the humidity level. Observe that the `dispense_product` event is blocked during the temperature phase, but not during the humidity phase.

one of them.

In this case, static information does not suffice: as the `measurer` thread blocks the `dispense_product` event at some of its states, the static over-approximation would include this event — and so `dispense_product` events would not be triggered during humidity measurement and correction. Dynamic information, on the other hand, resolves this issue, as it allows us to distinguish between the two phases; see Table 1 for performance comparison.

Table 1: Performance of the vending machine program using the different execution mechanisms. The measurements were performed using a customer simulator, purchasing 250 products in random intervals. The table depicts the time the experiment took, the number of maintenance rounds performed during the experiment, and the average delay — the time between making an order and receiving the product. The improvement column measures the reduction in delay compared to the traditional execution mechanism. Note: this experiment was conducted using a C^{++} implementation, equivalent to the one depicted in Figures 12 and 13.

Execution	**#Servings**	**Time (min)**	**#Maintenance**	**Delay (sec)**	**Improvement**
Traditional	250	15:40	59	1.68	—
Static	250	12:30	50	0.85	50%
Dynamic	250	9:20	37	0.18	90%

We point out that the `measurer` thread's transitions are not deterministic — as they depend on input received through the `temperatureTooHigh` and `humidityTooHigh` subroutines. As previously explained, this does not pose a problem, as the execution mechanism calculates an over-approximation based on all the successor states of the thread's last known state.

```
while true ->
    case bp:bSync(#rwb{wait=?ALL}) of
      increase_temperature ->
         increaseTemperature(),
         bp:bSync(#rwb{request=temperature_corrected});
      decrease_temperature ->
         decreaseTemperature(),
         bp:bSync(#rwb{request=temperature_corrected});
      increase_humidity ->
         increaseHumidity(),
         bp:bSync(#rwb{request=humidity_corrected});
      decrease_humidity ->
         decreaseHumidity(),
         bp:bSync(#rwb{request=humidity_corrected});
    end.
```

Figure 13: The main method of the **corrector** thread. The thread waits for events `increase_temperature`, `decrease_temperature`, `increase_humidity` or `decrease_humidity`; if they are triggered, it responds by adjusting the temperature or humidity (this part is abstracted away in the subroutines). Then, the thread requests an event signaling that the request has been handled, and goes back to waiting for new requests.

4.3.2. Automated Graph Spanning

Recall that the dynamic approximation method includes spanning the state graphs of threads and integrating these graphs into the program. Manual spanning of state graphs is prone to error, and is rather tedious in large systems with many events. However, this spanning can also be performed automatically.

Automatic spanning is performed by separating the thread under inspection from its siblings, and then iteratively exploring its state graph until all its states and transitions have been found. Starting at the initial state, we check the thread's behavior in response to the triggering of each event that is not blocked by the thread in that state. After the triggering of each event, the thread arrives at a new state (synchronization point) — and, with proper bookkeeping, it is simple to check if the state was previously visited or not. Every new state discovered is added to a queue to be explored itself, in an iterative BFS-like manner. In the case of nondeterministic threads, each event is checked with every possible "non-behavioral configuration". For instance, if the triggering of event e sends the thread to either state s_1 or s_2 based on a coin toss, we simulate the triggering of e with each possible outcome of the coin toss in order to discover all possible successor states. Declarations of the possible non-behavioral configurations need to be supplied by the programmer.

Technically, isolating the threads and exploring their state graphs can be performed by running them in an separate execution environment, dedicated to this purpose. This mechanism has been implemented in the BPC framework for BP in C^{++} (see `http://www.b-prog.org`); implementing it in Erlang remains for future work.

5. Comparing B-Nodes and Eager Execution

Sections 3 and 4 presented two possible approaches for coping with BP's synchronization requirements in the face of external input and multiple time-scales within a single program. In this section we discuss the similarities, differences and possible synergies between the two approaches.

The b-node approach to program design (Section 3) is a two-layer approach, which, in essence, goes beyond a single behavioral program. Each b-node constitutes a distinct behavioral program, with its own vocabulary of internal events, within which b-threads are synchronized and BP's idioms can be used. Then, as an additional layer, external events are sent asynchronously between the b-nodes to signal the occurrence of certain internal events. This solution therefore requires an additional vocabulary of external events, to be used across b-nodes. Each external event typically corresponds to certain events that are internal to the b-nodes. Moreover, each b-node has auxiliary threads for handling asynchronous communication, as well as the translation of internal events to external inter-b-node events and vice versa.

In contrast, the eager execution mechanism of Section 4 allows alleviating the synchronization requirements between the b-threads of a single behavioral program with no external means. For those sets of threads within the program, called modules, that are sufficiently independent — i.e., they form a strict modular design — eager execution results in an execution in which distinct modules are executed independently of one another (as far as each module's controlled events are concerned; of course, a dependency arises when one module waits for another module's controlled events). In this solution, there is a single vocabulary of events that is used across the program, and communication between the modules is facilitated by BP's native idioms.

It turns out that the two approaches are closely related. As demonstrated in the example at the end of Section 4.2.2 (see Figure 11), a program designed according to the b-node approach induces an equivalent program with a strict modular design. Similarly, it is also possible to transform a strict modular design into a b-node based design: each module is transformed into a b-node, and modules that would wait for events controlled by other modules are adjusted to wait for matching external events instead. However, when performing these transformations, one must take into account that, unlike the eager execution mechanism, the external message passing mechanism is not guaranteed to preserve event order; for instance, if module M_1 signals module M_2 twice, by requesting events e_1 and e_2 in that order, the eager execution approach guarantees that event e_1 is received before e_2. This guarantee does not necessarily hold in the b-node case.

Despite these similarities, each approach offers its distinct benefits. The eager execution approach uses automated tools, and is thus simpler to use, whereas the more complicated implementation details of simultaneously executing multiple modules are hidden within the execution mechanism itself, and the user does not need to implement external mechanisms and auxiliary events. Thus, the automated approach allows the specification and direct coding of richer scenarios, without having to break the scenario at the b-node boundary.

Moreover, even when the design is not strictly modular, eager execution generally relaxes

some of the synchronization that arises between threads — especially when the dynamic approach of Section 4.3 is used. These relaxations may yield performance improvements, as demonstrated in Section 4.3.1.

The major drawback of eager execution with respect to the b-node approach is that every thread must generally communicate with a *global* execution mechanism for each triggered event. While this is still significantly better than synchronizing all b-threads at each step of the execution, it may limit the applicability of the approach in such cases where communication is costly or unreliable. In contrast, the b-node approach allows programmers to fine-tune their programs in the face of such constraints: the execution mechanisms are local to each b-node, and, at points where inter-node communication is needed, messages can be routed directly to the desired recipient. This produces programs that are able to run in a highly distributed fashion.

In order to enjoy the benefits of both worlds, one may combine the two approaches within a single system. One way to do this is to apply the b-node approach of Section 3 but to implement an eager execution mechanism within each b-node. This may yield performance improvements over the basic b-node approach. Another way entails applying the eager execution approach, and enhancing it to support distributed execution with selective message exchanges between disparate modules, as in the b-node approach (see Appendix E). Yet another approach to the combination, aimed at automating the b-node approach, is to specify b-threads as if they are in a single b-node and then use automated tools to determine b-node boundaries (along the lines of [31]). Automated tools could then add the necessary processes for creating physically distributed b-nodes as in Section 3.

When eager execution or automated construction of behavioral modules or nodes are available, there is the additional advantage of the specified scenarios being readily able to cross the boundaries of objects, nodes or other components, enriching the expressiveness of the specification with regard to overall system behavior. Then, distributing the nodes according to physical constraints may be done as a separate task — automated when possible, manual when not. The resulting b-nodes may or may not be related to components. They may all be local behaviors related to a single object that run in very different time scales. A b-node may also involve multiple physical objects if the communication between them is fast enough relative to the required time scale, such that synchronization does not pose a performance issue. Furthermore, when a system-wide scenario has to cross multiple nodes that do not use the same event names, it can still be specified in BP — splitting the scenario into separate parts. Future development may include the automatic splicing such separate scenarios into a single one for visualization purposes, or automatic decomposition of a cross-node scenario based on user-provided hints of where external events and asynchronous communication should be inserted.

6. Positioning BP relative to Mainstream Actor and Agent Programming

The BP-based design patterns presented here coexist with, complement, and leverage several existing actor-oriented and agent-oriented concepts and models presented to-date. In comparing pure behavioral programming (without b-nodes) to the actor model of Agha [3],

for example, we observe that (a) generally, b-threads are not explicitly aware of each other, and communicate only indirectly, using the request, wait and block idioms; (b) BP focuses on interweaving independent behaviors towards a desired sequence of events, and is less focused on issues related to the parallel execution of any part of the independent behaviors (in fact, some implementations of the behavioral execution mechanism are single-threaded; e.g., the Java model-checker in [24], and the JavaScript implementation [38]); (c) in BP, asynchrony is allowed only between b-nodes, while within a b-node, behavior coordination imposes full synchronization (c.f., statecharts [22]). Furthermore, In Agha [3, Chapter 6], there is a discussion of how the problems of deadlocks and divergence (infinite loops) are dealt with in the actor model, and why the problem of shared memory is a non-issue. As discussed in detail in our work on model-checking [24], in BP we expect that deadlocks and infinite loops (liveness violations) will either be readily found in the code of individual b-threads or be directly attributed to a problem in the specifications, as the modules are aligned with the requirements. Further, a behavioral program where b-threads do not share data has the desired property that arbitrary delays between any two synchronization points in any b-thread do not change the sequence of events generated, if the external events arrive at the same super-steps. This eliminates many types of race conditions.

BP principles have already been implemented in several environments and languages. It would be interesting to explore the synergy between BP and agent-oriented-specific languages, such as AgentSpeak and Jason [7], 2APL [15], GOAL [29], SIMPA [41], Indigolog [8], JIAC [8], and Axum [1]. Such synergy could emerge from interfacing agents and behavioral programs, from turning b-nodes into agents and using agent programming languages to handle communication between behavioral nodes, or from introducing blocking idioms into agent-oriented languages.

Agents are often portrayed with human-like cognitive capabilities; e.g., Belief-Desire-Intention designs [12, 40, 15], goal oriented agents [29], autonomous agents with mental states [43], and agents with purpose and emotions [46]. Since the intelligence of an entity is often described through its handling of a particular scenario, based on our experience so far we believe that BP idioms can support the programming of rich cognitive concepts in a simple and natural way.

In [32], Alan Kay highlights the naturalness of programming with rules. Behavioral programming is often reminiscent of rule-based systems but it extends the concept by allowing the ability to easily monitor, and react to, entire scenarios without requiring complex state management in the rules. Furthermore, in BP, event-blocking facilitates expressiveness and behavioral composition.

The appendix of Bordini et al. [8] contains criteria for comparing agent-oriented platforms and languages. It would be interesting to check if and how the BP approach can be assessed subject to these criteria.

An agent-oriented coordination mechanism similar to behavioral programming (however without the crucial event-blocking), and which also uses the term super-step, is proposed in [39].

The coordination and event selection of BP can be seen as analogous in some ways to the tuple-space model. An implementation of BP using a tuple-space model can be found

in the work of Shimony et al in the Picos environment [42].

As mentioned in [27] specifying behaviors as modifications to base programs is presented in the study of superimpositions [11], and subsquently in aspect oriented programming (AOP) [34]. BP offers practical ways for implicit, indirect control of each behavior over all other behaviors, without direct reference from a controlling module to a controlled one. We believe that BP can contribute towards implementing symmetric aspects, complementing the currently prevalent asymmetric nature of AOP that distinguishes base code from aspects. In addition, BP allows for more intuitive state management, in the triggering of behaviors by sequences of events, as compared to standard AOP, where join-points commonly represent individual events, and triggering behaviors following rich sequences of events requires non-trivial handling in the aspect code.

7. Conclusion and Future Work

We have described behavioral programming as a design pattern in Erlang, and have presented a visualization tool that helps in the comprehension of programs designed in this manner. We then offered general design patterns for dealing with synchronization issues in the context of execution in different time scales and of environment-generated events. We have argued that connecting behaviorally programmed nodes using a simple messaging infrastructure is a promising approach to the incremental development of complex systems. We demonstrated how systems can be decomposed into b-nodes, such that the run-time synchronization requirements are substantially reduced. We also demonstrated how this decomposition can preserve behavioral programming's natural, incremental development and the alignment of behavior modules with requirements.

Future work can progress in several directions: Studying the impact of BP design patterns in large realistic case studies, exploring ways to automatically partition large, fully behavioral systems into non-synchronized nodes, developing formal methods and tools to verify (e.g., model-check) behavioral programs constructed in this manner wholly or compositionally, and exploring the addition of behavioral synchronization and event blocking to more mainstream actor- and agent-based platforms.

Acknowledgements

We thank the anonymous reviewers for valuable comments and suggestions on an earlier version of this paper. These comments have led to a substantial enhancement of the paper. We would like to thank Einat Fuchs, whose lecture about coordination of cockroach locomotion [17] inspired some of the ideas in this paper. We thank Dan Brownstein and Nir Svirsky for their programming work on the MATLAB simulation and the control software of the behaviorally-controlled quadrotor, and Nadav Shechter and Oren Othnay for their work on flying a real quadrotor with BP.

The research of the D. Harel, A. Kantor, G. Katz, A. Marron and G. Wiener was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, by an Advanced Research Grant to Harel from

the European Research Council (ERC) under the European Community's Seventh Framework Programme (FP7/2007-2013) and by the Israel Science Foundation. The research of G. Weiss was supported by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University, by a reintegration (IRG) grant under the European Community's FP7 Programme, and by the Israel Science Foundation.

References

[1] *Axum Programmer's Guide*. Microsoft, 2010.

[2] M. Abadi and Y. A. Feldman. Automatic recovery of statecharts from procedural code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 238–241. ACM, 2012.

[3] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[4] J. Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[5] A. Ashrov, A. Marron, G. Weiss, and G. Wiener. A use-case for behavioral programming: an architecture in JavaScript and Blockly for interactive applications with cross-cutting scenarios. *Science of Computer Programming*, 2014.

[6] D. Barak, D. Harel, and R. Marelly. Interplay: Horizontal scale-up and transition to design in scenario-based programming. *Lectures on Concurrency and Petri Nets*, pages 66–86, 2004.

[7] R. Bordini, J. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley and Sons, 2007.

[8] R. Bordini, M. Dastani, J. Dix, and A. Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, 2009.

[9] S. Bouabdallah. *Design and control of quadrotors with application to autonomous flying*. PhD thesis, Ecole Polytechnique Federale De Lausanne, 2007.

[10] S. Bouabdallah, P. Murrieri, and R. Siegwart. Design and control of an indoor micro quadrotor. In *International Conference on Robotics and Automation*, volume 5, pages 4393–4398. IEEE, 2004. ISBN 0780382323.

[11] L. Bouge and N. Francez. A compositional approach to superimposition. In *POPL*, 1988.

[12] M. Bratman, D. Israel, and M. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 1988.

[13] 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.

[14] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[15] M. Dastani. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[16] N. Eitan, M. Gordon, D. Harel, A. Marron, and G. Weiss. On visualization and comprehension of scenario-based programs. *Int. Conf. on Program Comprehension (ICPC)*, 2011.

[17] E. Fuchs, P. Holmes, T. Kiemel, and A. Ayali. Intersegmental coordination of cockroach locomotion: adaptive control of centrally coupled pattern generator circuits. *Frontiers in Neural Circuits*, 2010.

[18] R. Fujimoto. Parallel and distributed simulation. In *Proc. Winter Simulation Conference (WSC)*, pages 118–125, 1995.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995. ISBN 0201633612.

[20] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8 (3):231–274, 1987.

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

[22] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *TOSEM*, 5(4), 1996.
[23] 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.
[24] 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.
[25] D. Harel, G. Katz, A. Marron, and G. Weiss. Non-intrusive repair of reactive programs. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2012.
[26] 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.
[27] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Communications of the ACM*, 55(7): 90–100, 2012.
[28] T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using Giotto. *Control Systems Magazine, IEEE*, 2003.
[29] K. Hindriks. Programming rational agents in GOAL. *Multi-Agent Programming:*, 2009.
[30] N. Jennings. An agent-based approach for building complex software systems. *CACM*, 2001.
[31] G. Katz. On module-based abstraction and repair of behavioral programs. *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 518–535, 2013.
[32] A. Kay. Programming and programming languages. Technical report, VPRI Research Note RN-2010-001., 2010.
[33] R. Keller. Formal verification of parallel programs. *Comm. Assoc. Comput. Mach.*, 19(7):371–384, 1976.
[34] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP)*, 1997.
[35] H. Kugler, C. Plock, and A. Roberts. Synthesizing biological theories. In *CAV*, 2011.
[36] D. Liberzon. *Switching in systems and control*. Springer, 2003.
[37] J. Lygeros, G. Pappas, and S. Sastry. An introduction to hybrid system modeling, analysis, and control. *Preprints of the First Nonlinear Control Network Pedagogical School*, 1999.
[38] A. Marron, G. Weiss, and G. Wiener. A decentralized approach for programming interactive applications with JavaScript and Blockly. *Proceedings of the 2nd AGERE! workshop (Programming systems, languages and applications based on actors, agents, and decentralized control abstractions) at ACM SIGPLAN SPLASH*, pages 59–70, 2012.
[39] W. Miao and W. Tong. Agent based serviceBSP model with superstep service for grid computing. In *GCC*, 2007.
[40] A. Rao, M. Georgeff, A. A. I. Institute, T. Department of Industry, and A. Commerce. *Modeling rational agents within a BDI-architecture*. Australian Artificial Intelligence Institute, 1991.
[41] A. Ricci, M. Viroli, and G. Piancastelli. simpA: A simple agent-oriented java extension for developing concurrent applications. *Languages, Methodologies and Development Tools for Multi-Agent Systems*, pages 261–278, 2008.
[42] B. Shimony, I. Nikolaidis, P. Gburzynski, and E. Stroulia. On coordination tools in the PicOS tuples system. *SESENA*, 2011.
[43] Y. Shoham. Agent-oriented programming. *Artificial intelligence*, 1993.
[44] A. D. Stefano and C. Santoro. Using the Erlang language for multi-agent systems implementation. In *IAT*, pages 679–685, 2005.
[45] A. D. Stefano, F. Gangemi, and C. Santoro. Eresye: artificial intelligence in erlang programs. In *Erlang Workshop*, pages 62–71, 2005.
[46] M. Travers. *Programming with Agents: New metaphors for thinking about computation*. PhD thesis, MIT, 1996.

Appendix A. Example: Coordinated sequential processing

In this section we present an example for using BP for bulk processing of a large number of records to perform business operations, where multiple, independent sequential processes are implemented as b-threads.

Business operations include time based events (e.g., generation of periodic correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g., interest accrual or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Such batch processing systems are used to process daily billions of transactions for enterprises around the world. Easy interweaving of such processes can be of great value. For example, consider the printing of different notices, paper advertisements or coupons, for insertion in each customer's envelope. Sequential processes may independently customize individual messages to customers in a large database, but they need to be coordinated, so that all messages to a given customer are printed consecutively. The BP design pattern enables such coordination with minimal dependency across the different sequential processes.

In BP terms, sequential batch processing can be formulated as an iterative bidding/consensus process where, for each record of data, a set of independent b-threads collaborate by using the request/wait/block idioms to express their views of how the record should be processed. More specifically, the system can be programmed using a sequencer b-thread that controls the sequencing of the records, and b-threads that model various considerations regarding how the records should be processed.

To demonstrate the technique, we present an implementation of the Sieve of Eratosthenes algorithm:

```
sequencer(I) when I < 100 ->
  sync(#rwb{request=[I]}),
  T = sync(#rwb{request=[prime,not_prime]}),
  io:format("~w is ~w ~n", [I,T]),
  sequencer(I+1);

sequencer(I) -> io:format("---~n").

pFactors(I) -> pFactors(2*I,I).

pFactors(N,I) ->
  sync(#rwb{wait = [N]}),
  sync(#rwb{block=[prime], wait = [N+1]}),
  pFactors(N+I,I).


factory(I) ->
  I = sync(#rwb{wait = [I]}),
  T = sync(#rwb{wait = [prime,not_prime]}),
  if
    T == prime ->
       add(spawn(fun() -> pFactors(I) end), 1);
    true -> ok
  end,
  factory(I+1).

run() ->
```

```
init(),
add(spawn(fun() -> sequencer(2) end), 2),
add(spawn(fun() -> factory(2) end), 3),
start().
```

The `sequencer` is a b-thread that leads the sequential processing of the natural numbers and attempts to declare each one a prime. The `pFactors` b-thread blocks the multiples of a prime number from being declared as prime. The `factory` b-thread is responsible for spawning and registering a `pFactors` b-thread whenever a prime number is discovered. Note that this kind of dynamic addition of b-threads extends the basic collaboration scheme described above, and requires further attention in definition and development. The `start` method starts an instance of the `sequencer` and the `factory` b-threads. This code does not conform with the assumptions outlined in Section 2.10, and therefore cannot be automatically visualized by `bs_wis`.

Appendix B. Eager Execution Formalized

We now formally define the the eager execution mechanism. For simplicity, all definitions in this section exclusively consider programs with deterministic b-threads of equal priorities; handling nondeterministic threads and threads with varying priorities is similar.

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. In order to define the eager execution mechanism, we construct a labeled transition system (LTS) denoted by $\widehat{\mathrm{LTS}}(P) = \langle \widehat{Q}, \widehat{q_0}, \widehat{\delta} \rangle$, which is defined next. We use some of the notation introduced in Section 2.9.

The set of states is given by $\widehat{Q} := (Q^1 \times \Sigma^*) \times \cdots \times (Q^n \times \Sigma^*)$. Each state is thus a tuple consisting for each thread of its state and the contents of its event queue. Let $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$ be a state. We use the standard notation $\delta^i(q^i, u^i)$ to denote the state in Q^i after applying the 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 tuple comprised of these states by $\overline{q} := \langle \delta^i(q^i, u^i) \rangle_{i=1}^n$; we refer to it as the *indication* of q. Note that $\overline{q}$ naturally corresponds to a state in Q, which is the set of states of $\mathrm{LTS}(P) = \langle Q, q_0, \delta \rangle$ defined in Section 2.9. We slightly abuse notation and write that $\overline{q} \in Q$. Naturally, the initial state is $\widehat{q_0} := \langle (q_0^1, \varepsilon), \ldots, (q_0^n, \varepsilon) \rangle \in \widehat{Q}$.

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. As previously mentioned, eager execution has various forms (depending on the analysis technique that is used); each form is characterized by its specific choice for these approximations. 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 state $\delta^i(q^i, u^i)$, and that $\mathcal{B}^i(q)$ is a superset of the blocked events at that state. That is,

$$\mathcal{R}^i(q) \subseteq R^i(\delta^i(q^i, u^i)) \qquad B^i(\delta^i(q^i, u^i)) \subseteq \mathcal{B}^i(q)\,. \tag{B.1}$$

Moreover, we require that in case a thread is synchronized, the two approximations are precise. More formally, if $u^i = \varepsilon$ for some $i \in [n]$ (where $[n]$ denotes the set of indices $\{1, \ldots, n\}$),

so that in particular $\delta^i(q^i, u^i) = q^i$, then we require

$$\mathcal{R}^i(q) = R^i(q^i) \qquad \mathcal{B}^i(q) = B^i(q^i)\,. \tag{B.2}$$

These two requirements are sufficient for our purposes. One may easily verify that the eager execution with either static or dynamic analysis technique complies with the requirements. From these, we obtain that the *approximated enabled events*, defined in the following, are contained in the enabled events at the indication state $\overline{q} \in Q$; i.e.,

$$\mathcal{E}(q) \;:=\; \bigcup_{i=1}^{n} \mathcal{R}^i(q) \setminus \bigcup_{i=1}^{n} \mathcal{B}^i(q) \subseteq E(\overline{q})\,. \tag{B.3}$$

In case all threads are synchronized, i.e., $u^i = \varepsilon$ for all $i \in [n]$, we obtain

$$\mathcal{E}(q) = E(\overline{q})\,. \tag{B.4}$$

The nondeterministic transition function $\widehat{\delta} : \widehat{Q} \times (\Sigma \dot{\cup} \{\varepsilon\}) \to 2^{\widehat{Q}}$ 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:

- 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 $r^{i_0} = \delta^{i_0}(q^{i_0}, a)$, and for all other $i \in [n] \setminus \{i_0\}$ it holds that $r^i = q^i$ and $v^i = u^i$. These transitions correspond to threads with queued events processing these events — they change states, while the other threads do not move.

- If $\sigma \in \Sigma$, and moreover $\sigma \in \mathcal{E}(q)$, then $\widehat{\delta}(q, \sigma)$ is defined to be the singleton $\widehat{\delta}(q, \sigma) = \{\langle q^i, u^i\, \sigma \rangle_{i=1}^n\}$. 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.

We conclude by formally proving that the eager execution mechanism yields runs that are valid accord to BP's original semantics, by comparing $\widehat{\mathrm{LTS}}(P)$ defined above and $\mathrm{LTS}(P)$ from Section 2.9. Technically, we claim that each complete run of $\widehat{\mathrm{LTS}}(P)$ is a complete run of $\mathrm{LTS}(P)$; i.e., $\mathfrak{L}(\widehat{\mathrm{LTS}}(P)) \subseteq \mathfrak{L}(\mathrm{LTS}(P))$. This is a consequence of the following lemmata.

When considering $\widehat{\mathrm{LTS}}(P)$, $q \xrightarrow{\sigma} q'$ stands for $q' \in \widehat{\delta}(q, \sigma)$, as customary when discussing transition systems (for any states $q, q' \in \widehat{Q}$ and a possibly silent event $\sigma \in \Sigma \cup \{\varepsilon\}$). Also, recall that $q \in \widehat{Q}$ is a *terminal state* if for all $\sigma \in \Sigma \cup \{\varepsilon\}$ it holds that $\widehat{\delta}(q, \sigma) = \emptyset$. Similar notations and terminology apply to $\mathrm{LTS}(P)$.

Lemma 1. *Let $q, q' \in \widehat{Q}$ and $\sigma \in \Sigma \cup \{\varepsilon\}$ such that $q \xrightarrow{\sigma} q'$ in $\widehat{\mathrm{LTS}}(P)$.*

1. *If $\sigma = \varepsilon$, then $\overline{q'} = \overline{q}$.*

2. *If $\sigma \in \Sigma$, then $\overline{q} \xrightarrow{\sigma} \overline{q'}$ in $\mathrm{LTS}(P)$.*

Proof. 1: Denote $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$, and suppose that $\sigma = \varepsilon$. By the definition of $\widehat{\delta}$, we obtain that $q' = \langle r^i, v^i \rangle_{i=1}^n$, where all the coordinates are the same as in q, except for the one corresponding to $i_0 \in [n]$. In the latter coordinate we get $\delta^{i_0}(r^{i_0}, v^{i_0}) = \delta^{i_0}(\delta^{i_0}(q^{i_0}, a), v^{i_0}) = \delta^{i_0}(q^{i_0}, a\, v^{i_0}) = \delta^{i_0}(q^{i_0}, u^{i_0})$, as needed.

2: Now, suppose $\sigma \in \Sigma$. According to the definition of $\widehat{\delta}$, $\sigma \in \mathcal{E}(q)$ and $q' = \langle q^i, u^i\, \sigma \rangle_{i=1}^n$. By (C.3) and by the definition of δ, we get that in $\mathrm{LTS}(P)$ it holds that $\overline{q} \xrightarrow{\sigma} \langle\, \delta^i(\delta^i(q^i, u^i), \sigma)\, \rangle_{i=1}^n = \langle\, \delta^i(q^i, u^i\, \sigma)\, \rangle_{i=1}^n = \overline{q'}$. □

Corollary 1.

1. *Let $r_0 \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_k} r_k$ be a finite execution of $\widehat{\mathrm{LTS}}(P)$ ($k \geq 0$). There exists a finite execution $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_t} s_t$ of $\mathrm{LTS}(P)$ ($t \geq 0$) such that $\overline{r_k} = s_t$ and $\sigma_1\, \sigma_2 \cdots \sigma_k = a_1\, a_2 \cdots a_t$.*

2. *Let $r_0 \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots$ be an infinite execution of $\widehat{\mathrm{LTS}}(P)$. There exists an execution $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$ of $\mathrm{LTS}(P)$ such that $\sigma_1\, \sigma_2 \cdots = a_1\, a_2 \cdots$.*

sketch. 1: By induction on k. For $k = 0$ the claim follows from the fact that $\overline{\widehat{q_0}} = q_0 \in Q$; the induction step follows from Lemma 1.

2: By an inductive construction of the execution, which similarly follows from Lemma 1. □

Lemma 2.

1. *If $q \in \widehat{Q}$ is a terminal state in $\widehat{\mathrm{LTS}}(P)$, then $\overline{q}$ is a terminal state in $\mathrm{LTS}(P)$.*

2. *There is no infinite sequence $q \xrightarrow{\varepsilon} q' \xrightarrow{\varepsilon} q'' \xrightarrow{\varepsilon} \cdots$ in $\widehat{\mathrm{LTS}}(P)$.*

Proof. 1: As q is terminal, by the definition of $\widehat{\delta}$ it holds that all the queues in q are empty (otherwise, $\widehat{\delta}(q, \varepsilon) \neq \emptyset$); i.e., $q = \langle q^i, \varepsilon \rangle_{i=1}^n$. Let $a \in \Sigma$. Because q is terminal, $a \notin \mathcal{E}(q)$. Thus, by (C.3), $a \notin E(\overline{q})$, and therefore by the definition of δ, $\delta(\overline{q}, a) = \emptyset$.

2: For each state $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$, consider the total size of the queues, denoted by $\varphi(q) := \Sigma_{i=1}^n |u^i| \in \mathbb{N}$. Given such an infinite sequence of states, φ is strictly decreasing (by the definition of $\widehat{\delta}$), which contradicts the well-foundness of the natural numbers. □

Corollary 2. *Let $r_0 \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots$ be a* complete *(finite or infinite) execution of $\widehat{\mathrm{LTS}}(P)$. There exists a* complete *(finite or infinite, respectively) execution $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$ of $\mathrm{LTS}(P)$ such that $\sigma_1\, \sigma_2 \cdots = a_1\, a_2 \cdots$.*

The corollary follows from Corollary 1 and Lemma 2. It is equivalent to $\mathfrak{L}(\widehat{\mathrm{LTS}}(P)) \subseteq \mathfrak{L}(\mathrm{LTS}(P))$, as needed.

Appendix C. 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 rightarrow 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 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.

Appendix C.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.

Appendix C.1.1. Relaxing Synchronization using Static Information

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 4.2.2 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.

Appendix C.1.2. Relaxing Synchronization using Dynamic Information

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.

Appendix C.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. For simplicity, we assume all the threads in the program have equal priorities.

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 $\widehat{\mathrm{LTS}}(P) = \langle \widehat{Q}, \widehat{q_0}, \widehat{\delta} \rangle$, 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 $\mathrm{LTS}(P) = \langle Q, q_0, \delta \rangle$ 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,

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

  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

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

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

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

- $\widehat{\delta} : \widehat{Q} \times (\Sigma \dot{\cup} \{\varepsilon\}) \to 2^{\widehat{Q}}$ 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 $\widehat{\delta}(q, \sigma) = \{\langle q^i, u^i\, \sigma \rangle_{i=1}^n\}$. 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 $\widehat{\mathrm{LTS}}(P)$ is a complete run of $\mathrm{LTS}(P)$; i.e., $\mathfrak{L}(\widehat{\mathrm{LTS}}(P)) \subseteq \mathfrak{L}(\mathrm{LTS}(P))$. The actual proof is similar to that of the deterministic case.

Appendix D. Modularity Formalized

In this section we use the formalization of eager execution described in Section 4.2.2 in order to rigorously formulate and prove Proposition 1.

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 $C^i := \left(\bigcup_{s \in Q^i} B^i(s)\right) \cup \left(\bigcup_{s \in Q^i} R^i(s)\right)$. 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 $\widehat{\mathrm{LTS}}(P)$ in Appendix B. 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

$$\mathcal{B}^i(q) \subseteq E_j \,. \tag{D.1}$$

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

We now turn to prove the following technical proposition that, when applied iteratively, implies Proposition 1.

Proposition 2. *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 (C.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 (C.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 (C.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 (C.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 (D.1), 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. □

Appendix E. The Distributed Execution Mechanism

The concept of behavior modules discussed in Section 4.2.2 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 [13, 18], 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 4.3). 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. The priority-based event selection mechanism has this property.

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 3. *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 Appendix E.1 we formally define the model and prove Lemma 3.

Appendix E.1. 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: ∀i, s_i ← q_0^i
2: LastEvent ← φ
3: while true do
4:    Sync ← φ
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 ≠ φ and |δ^j(s_j, LastEvent)| > 1 then
9:          Broadcast s'_j
10:      s_j ← s'_j
11:      Sync ← Sync ∪ BT^j
12:   for BT^ℓ ∉ M_i do
13:      if LastEvent ≠ φ then
14:         if |δ^ℓ(s_ℓ, LastEvent)| > 1 then
15:            Update s_ℓ according to broadcasts from other agents
16:         else
17:            s_ℓ ← δ^ℓ(s_ℓ, LastEvent)
18:   E ← ⋃_{j=1}^n (R^j(s_j)) − ⋃_{j=1}^n (B^j(s_j))
19:   LastEvent ← 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 3:

Proposition 3. *Let $P = \{BT^1, \ldots, BT^n\}$ be a behavioral program, divided into machines $M_1, \ldots, M_k$ with coordinator agents $C_1, \ldots, C_k$. Let $f : 2^\Sigma \setminus \{\emptyset\} \to \Sigma$ be a deterministic event selection function. Then agents $C_1, \ldots, C_k$ produce a* cohesive *run; that is, there exists a unique run $e_1 e_2 \ldots$ such that at synchronization point i,* every *coordinator C_ℓ triggers e_i. Further, the sequence $e_1 e_2 \ldots$ 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, $S^i_m = \langle s^i_m(BT^1), \ldots, s^i_m(BT^n)\rangle$. 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 $|\delta^\ell(s^i_1(BT^\ell))| = 1$, and if so it deduces that $s^{i+1}_1(BT^\ell) = \delta^\ell(s^i_1(BT^\ell))$. In this case, C_2 will learn the state of BT^ℓ when that thread synchronizes, and it will hold that $s^{i+1}_1(BT^\ell) = s^{i+1}_2(BT^\ell)$.

The other option is that thread BT^ℓ is performing a nondeterministic transition, i.e. $|\delta^\ell(s^i_1(BT^\ell))| > 1$. 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 $s^{i+1}_1(BT^\ell) = s^{i+1}_2(BT^\ell)$.

Further, upon receiving the synchronization request from a local thread BT^t, agent C_1 uses its stored state graphs to check whether $|\delta^t(s^i_1(BT^t))| > 1$. 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 $S_1^{i+1} = S_2^{i+1}$. □ □

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.

Appendix E.2. 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 using static or dynamic thread information 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.