# Behavioral Programming, Decentralized Control, and Multiple Time Scales

David Harel, Assaf Marron, Guy Wiener
Weizmann Institute of Science
{first name.last name}@weizmann.ac.il

Gera Weiss
Ben Gurion University of the Negev
geraw@cs.bgu.ac.il

## Abstract

Behavioral programming is a recently proposed approach for non-intrusive incremental software development. We propose that behavioral programming concepts, such as behavioral decomposition, synchronized execution of independent behaviors, and event blocking, can help in the incremental and natural coding of complex decentralized systems, complementing actor-oriented and agent-oriented approaches. We also contribute to the existing research on behavioral programming a method for coordinating behaviorally-programmed components which, due to different time scales or interaction with the external environment, cannot synchronize and thus cannot employ event blocking. We show that the resulting decentralized system retains many of the advantages present in a purely behavioral, fully synchronized system.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming

***General Terms*** Design, Languages

***Keywords*** Decentralized Control, Behavioral Programming, LSC, Java, Erlang, BPJ, Multiple Times Scales

## 1. Introduction

In the context of software engineering, the term *decentralized control* is sometimes used (see, e.g., the call-for-papers of the AGERE! workshop [1]) for approaches to programming focused on composition of independent control structures. This decentralization does not imply that software components be physically distributed, but rather that the software be organized in components that represent different facets of the system. A key motivation is to leverage decentralization towards ease in development and maintainability.

We start this paper by presenting the recently proposed approach of *behavioral programming*[27, 32], which we believe reflects many decentralized-control traits, and may contribute to the quest for useful decentralized programming metaphors, languages, and tools. As described in Section 2, the approach is based on programming software components, called behavior threads (b-threads), which specify system behavior as desired and forbidden sequences of events. The b-threads are composed at run-time using a simple, yet powerful, execution mechanism, without requiring direct message exchanges between them. Thus, behavioral programming may complement or serve as an alternative to actor-oriented and agent-oriented approaches to decentralized control. As we show, the main benefits of the proposed approach are alignment of components with requirements and the ability to add or remove inter-object behaviors by incorporating b-threads that specify the change directly.

We then focus on the topic of synchronization in behavioral programming. The execution mechanism applies a high rate of synchronization between b-threads, which may be a problem in real-time and distributed systems. Based on the observation that the term "high rate" in the preceding sentence is relative, we propose that breaking the system into groups of b-threads that synchronize at different rates enables scaling the behavioral programming approach to systems with many heterogeneous, possibly distributed, behaviors.

The paper is organized as follows. Section 2 provides an overview and formal definitions for behavioral programming, as well as small annotated examples in Java and Erlang. It also discusses how behavioral programming deals with conflicts between independently programmed behaviors. In Section 3 we discuss challenges that the fully synchronized behavioral programming approach faces when applied to complex decentralized control problems. Section 4 describes the solutions we propose to these challenges. Sections 5–7 contain examples that illustrate how these solutions fit in the design of decentralized reactive systems for real-world control problems, and enable the natural and incremental development style of behavioral programming.

## 2. Behavioral Programming

### 2.1 Overview

Behavioral programming is an approach that advocates that desired and undesired scenarios need not be composed along the lines of system structure. The approach stems from research in the area of software engineering for reactive systems, i.e., programs that constantly interact with their environment. Specifically, behavioral programming is based on languages for capturing formal requirements of reactive systems in a way that allows their execution. Initially, the purpose of execution was to create the possibility of simulations that facilitate early feedback from users for refining the requirements. Later, the ideas and tools were enhanced to facilitate construction of final systems from modules aligned with requirements. The concepts were first introduced with the (visual) language of *live sequence charts* (LSC) [17, 23], and were recently integrated into general purpose programming languages, such as Java [32] and Erlang [51]. Additional implementations include SBT [39] and in the PiCos environment. [48].

To illustrate the naturalness of constructing systems from behaviors, consider how children may be taught, step by step, to play

*SPLASH'11 Workshops,* October 23–24, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-1183-0/11/10. . . $10.00

strategy games. For example, in teaching the game of Tic-Tac-Toe, we need to describe rules, such as:

1. To play, one player marks a square with X, then the other player marks a square with O, then it is X's turn again, and so on;
2. Once a square is marked, it cannot be marked again;
3. When one of the players places three of his or her marks in a horizontal, vertical, or diagonal line, the player wins.

Now we may already start playing. Later, the child may infer, or the teacher may suggest, some tactics:

4. After marking two Os in a line, the O player should try to mark the third square (to win the game);
5. After the X player marks two squares in a line, the O player should try to mark the third square (to foil the attack);
6. When other tactics are not applicable, the player should prefer the center square, then the corners, and mark an edge square only when there is no other choice.

Such required behaviors can be coded in executable software modules using behavioral programming idioms and infrastructure, as detailed in Section 2. Full implementations of behavioral Java and Erlang programs playing Tic-Tac-Toe, coded in this manner, are described in [32] and [51], respectively. In [33], it is shown how model-checking technologies allow discovery of unhandled scenarios, enabling the user to incrementally develop behaviors for new tactics (and forgotten rules) until a software system is achieved that plays legally, and assures that the computer never loses (which, in the case of Tic-Tac-Toe, is indeed possible).

This example, and the more detailed ones in the rest of the paper, highlight the following advantages of behavioral programming. First, we were able to code the Tic-Tac-Toe application in modules that are aligned with the requirements (rules and tactics) as users and programmers perceive them. Second, we added new tactics and rules (and even more can be added) without changing, or even looking at, existing code. Third, the resulting product is modular, in the sense that tactics and rules can be flexibly added and removed to create versions with different functionality; e.g., versions that play at different expertise levels. Since programming often begins before all requirements are stabilized, we believe that the ability to implement new requirements without modifying (or even accessing) existing code, and the alignment of code with requirements, are significant advantages, especially in projects involving autonomous development groups.

The versatility of the LSC language and of the Java and Erlang implementation versions has been demonstrated in several ways including: the *Play-Engine* and *PlayGo* development environments [23, 31], language packages [32, 51]; application in various domains, including hardware, telecommunication, production control, tactical simulators, and biological modeling [6, 15, 16, 21, 29, 47]; tools for compilation [30]; smarter execution using look-ahead [25, 28]; learning [19], model-checking (for discovering and handling conflicts and underspecification in incremental development) [33]; and, trace visualization and comprehension [20, 43].

## 2.2 The computation model

In this section we formalize the computation model of behavioral programming. Specifically, a behavior thread is a deterministic transition system and we apply a version of the product construction of automata to define the composition operator that yields an interleaved execution of a system of b-threads. This continues the discussion in [32] where Definitions 1 and 2 were introduced.

Recall that a *deterministic labeled transition system* is a quadruple $\langle S, E, \to, init\rangle$, where $S$ is a set of states, $E$ is a set of events, $\to$ is a (possibly partial) function from $S \times E$ to $S$, and $init \in S$ is the initial state. The runs of such a transition system are sequences of the form $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_i} s_i \cdots$, where $s_0 = init$, and for all $i = 1, 2, \cdots$, $s_i \in S$, $e_i \in E$, and the function $\to$ maps the pair $\langle s_{i-1}, e_i\rangle$ to $s_i$, written as $s_{i-1} \xrightarrow{e_i} s_i$. We say that $\langle S, E, \to, init\rangle$ is total if the transition function $\to$ is a total function.

Each behavior thread is modeled as a transition system, in which states are associated with event sets:

**Definition 1 (behavior thread [32]).** A *behavior thread* (abbr. *b-thread*) is a tuple $\langle S, E, \to, init, R, B\rangle$, where $\langle S, E, \to, init\rangle$ forms a deterministic total labeled transition system, $R\colon S \to 2^E$ is a function that associates each state with the set of events *requested* by the b-thread when in that state, and $B\colon S \to 2^E$ is a function that associates each state with the set of events *blocked* by the b-thread when in that state.

The set of all possible collective, interlaced runs of a set of behaviors threads is formalized as a composition operator:

**Definition 2 (runs of a set of b-threads [32]).** We define the *runs of a set of b-threads* $\{\langle S_i, E_i, \to_i, init_i, R_i, B_i\rangle\}_{i=1}^n$ as the runs of the labeled transition system $\langle S, E, \to, init\rangle$, where $S = S_1 \times \cdots \times S_n$, $E = \bigcup_{i=1}^n E_i$, $init = \langle init_1, \ldots, init_n\rangle$, and $\to$ includes a transition $\langle s_1, \ldots, s_n\rangle \xrightarrow{e} \langle s_1', \ldots, s_n'\rangle$ if and only if

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

and

$$\bigwedge_{i=1}^{n} \Big( \underbrace{\big(e \in E_i \implies s_i \xrightarrow{e}_i s_i'\big)}_{\text{affected b-threads move}} \wedge \underbrace{\big(e \notin E_i \implies s_i = s_i'\big)}_{\text{unaffected b-threads don't move}} \Big) \tag{2}$$

These definitions are meant only as an abstraction. In practice, we propose that b-threads use the full power of programming languages such as Java, Erlang and LSC to encode the logic succinctly and use appropriate software interfaces to (a) indicate when the program is "in a state" of the transition system; (b) assign requested and blocked events to each state; and (c) define transitions by assigning to each state a set of waited-for events. See Section 2.3 below for more details.

Note that while each b-thread is deterministic in its reaction to events, Definition 2 does not specify how events are selected, and thus there may be more than one run for a given set of b-threads. In execution, there could be multiple ways to select events and runs including random or planned selection. In this paper we focus on an approach guided by the desire to maintain simplicity and repeatability of computations. Specifically, we propose to add a priority scheme, in which the b-threads and the events are linearly ordered — inducing a lexicographic order also on the runs.

The default behavioral execution infrastructure of LSC (in the *Play-Engine* and *PlayGo*), the Java package (*BPJ*) and the Erlang module (bp) execute a set of b-threads by choosing, at each state of the composite system, the first event that is requested and is not blocked. More generally:

**Definition 3 (behavioral execution mechanism).** For a given set of b-threads, let $T$ be the transition system defined in Definition 2. A (deterministic) *behavioral execution mechanism* for $T$ is an event selection function $f\colon S \to E$, such that for each $s \in S$ there exists a transition $s \xrightarrow{f(s)} s'$ of $T$ (where $S$ and $E$ are as in Definition 2).

Figure 1 illustrates such an execution mechanism. The single run, induced by the single event that exits the sieve at each synchronization point, is as in Definition 2, with the added requirement that in each state the event selected is the one specified by $f$.

Priority-based selection is just one implementation of event selection. We can also introduce application-specific intelligence, or use various forms of look-ahead, as is done in the LSC-based

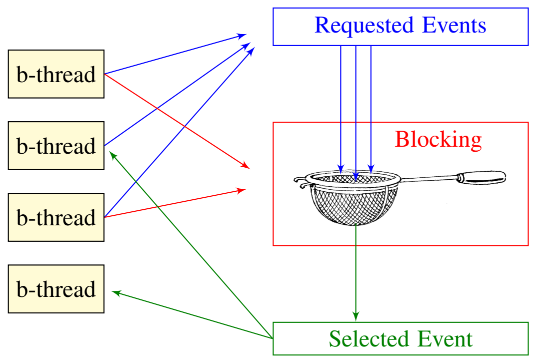

**Figure 1.** Collective execution of behavior threads using an enhanced publish/subscribe protocol: (a) all b-threads place their "bids", specifying requested events and blocked events; (b) a synchronization mechanism chooses an event that is requested but is not blocked; (c) b-threads waiting for the event are notified; (d) the notified b-threads progress to their next states, where they can place new bids.

techniques of *smart play-out* [28] and *planned play-out* [25], or adaptivity and learning as in [19].

## 2.3 Programming behaviorally in Java and Erlang

The implementation we propose for the computation model discussed in the previous subsection consists of software libraries and a methodology for building behavioral programs using them. Specifically, we now describe packages for behavioral programming in the Java and Erlang languages introduced in [32] and [51], respectively, and briefly review how the objects and functions of these packages can be used to construct systems from behaviors.

In a behavioral Java or Erlang program, each behavior thread is implemented as a Java thread or an Erlang process. The behavior threads synchronize by invoking a library method called `bSync` that takes the request, wait, and block bids as arguments (in Erlang, using the record type `rwb`). Invoking `bSync` first causes the b-thread to wait until all other b-threads invoke `bSync`. Then, the next event is selected (one that is requested and not blocked), b-threads that requested or waited for it are resumed, they proceed to their next `bSync`, and the process repeats. The selected event is passed to the b-threads via a field called `lastEvent` in Java and as a return value in Erlang.

This allows coding b-threads using the full power of the host language for control flow, such as loops and conditions, and for the logic that decides and constructs the bid at each synchronization point (represented as an invocation of `bSync`). From the programmer perspective, each behavior thread is a procedure that implements an aspect of the system's desired or undesired behavior (positive or negative scenario) by listening to (specific sequences of) events and by requesting and blocking events.

To illustrate this coding technique, consider a b-thread that increases water flow in a hot water tap by requesting five times the event `addHot` of turning the tap anticlockwise some small fixed amount. Another b-thread performs a similar action, with the event `addCold`, on the cold water tap. To increase the water flow in both taps in parallel, as may be desired for keeping the temperature stable, we activate the above b-threads alongside a third one, which forces the interleaving of events in the two scenarios. The third b-thread, for example, can be coded as "`repeatedly: {block AddCold until AddHot; block AddHot until AddCold}`". This pseudo code maps to a transition system with two states $q_1, q_2$ where $R(q_1) = R(q_2) = \emptyset$,

```
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=[addHot, addCold]}),
  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().
```

**Figure 2.** An example of using the `bp` module in Erlang.

$B(q_1) = \{\texttt{addCold}\}, B(q_2) = \{\texttt{addHot}\}$, $q_1 \xrightarrow{\texttt{addHot}} q_2$, and $q_2 \xrightarrow{\texttt{addCold}} q_1$.

Figure 2 illustrates how to implement such a system in Erlang, using the `bp` module. The behavior threads are implemented as four functions that are spawned to run in separate processes. The behavior thread `requestFiveAddHotEvents`, for example, calls `bSync` five times[1] and, in all of them, passes `addHot` as the only requested event and blocks nothing. The `requestFiveAddColdEvents` is similar and the `interleave` behavior takes care of keeping the temperature stable, as detailed above, by waiting for and blocking alternating events. The `wait` clause may also use a filter function for events, instead of an explicit list. The macro `?ALL` is shorthand for waiting for all possible events. The `display` behavior thread generates textual output for illustration purposes. In real applications it would be replaced by an actuator that translates the events into physical outputs, as we elaborate later in this paper. Note also the second parameter to the `bp:add` function representing the priority of the behavior thread, as discussed in the previous subsection. To see the role of priority consider, for example, what would happen if we didn't have the `interleave` behavior thread. In this case, because the priority of `requestFiveAddHotEvents` is higher than the priority of `requestFiveAddColdEvents`, we would get the five `addHot` events before the five `addCold` Events.

In Java, the same program can be coded as shown in Figure 4. Each behavior thread is an instance of a class that extends `BThread`. A Java thread is created for each behavior thread and the method `bp .startAll` starts all these threads. The method `bSync` implements the synchronization protocol, as described above. Its three parameters are the set of requested events, the set of waited-for events, and the set of blocked events. For programming convenience, using standard Java programming techniques, we allowed an event to also be treated as a (singleton) set containing only the event, and created the special event sets `none` and `all` with their obvious meaning.

[1] Throughout the paper we use *list comprehensions* for loops in Erlang. The expression `[X || _ <- seq(1,N)]` is a shorthand for "perform X for N times". For more details, see the official Erlang documentation.

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

**Figure 3.** Output of both the Erlang and the Java programs listed in Figures 2 and 4.

```java
static class requestFiveAddHotEvents extends BThread {
  public void runBThread() {
    for (int i = 1; i <= 5; i++) {
      bp.bSync(addHot, none, none);
    }
  }
}
static class requestFiveAddColdEvents extends BThread {
  public void runBThread() {
    for (int i = 1; i <= 5; i++) {
      bp.bSync(addCold, none, none);
    }
  }
}
static class Interleave extends BThread {
  public void runBThread() {
    while (true) {
      bp.bSync(none, addHot, addCold);
      bp.bSync(none, addCold, addHot);
    }
  }
}
static class Display extends BThread {
  public void runBThread() {
    while (true) {
      bp.bSync(none, all, none);
      System.out.println("Event: " + lastEvent);
    }
  }
}
public static void main(String[] args) {
  bp.add(new requestFiveAddHotEvents(), 1.0);
  bp.add(new requestFiveAddColdEvents(), 2.0);
  bp.add(new Interleave(), 3.0);
  bp.add(new Display(), 4.0);
  bp.startAll();
}
```

**Figure 4.** An example of using the `bpj` package in Java.

### 2.4 Dealing with conflicting behaviors

A concern often associated with coding and composing requirements in separate modules without special dependency considerations, is that conflicts may emerge, yielding undesired joint behavior or system failures. In [33] we presented a verification tool for behavioral programs written in Java. It enables early discovery of conflicts and under-specification in the requirements, and facilitates the usage of counterexamples in incremental development. Using (explicit-state) model checking, behavioral Java programs are verified directly, without being translated first into a language specific to the model-checker. We use an abstraction that focuses on program states at synchronization points, and treats the transitions between the states as atomic. Synthesis techniques have also been applied to behavioral programs in LSC [26, 38], to check for conflicts and, when possible, generate a deterministic automaton that implements the specification.

Moving from development to run-time, model-checking and planning algorithms [25, 28] can be used to look ahead and avoid conflicts where applicable. Deadlocks in which all requested events are blocked can be detected using designated low priority events which may trigger alerts or recovery actions. Divergent, i.e., run-away b-threads, which fail to synchronize, can be detected at run time using timers calibrated according to the different timescales (discussed in the coming sections) which are expected of the b-threads. Race condition are avoided in behavioral programming when behaviors communicate only through events, and do not use host language facilities to share data.

When a conflict actually exists and application changes are mandated, we can resolve conflicts and under-specification by coding refinements as if they are new requirements, in new behavior threads. E.g., when two (individually correct) behavior specifications mandate that opposing actions take place following a given event sequence, a conflict-resolving refinement scenario (or a priority setting) may guide the choice. When b-thread code must be changed (e.g., to handle code errors or changes in requirements, as in any other software), the task may be easier in behavioral programming, as b-threads are expected to be short and self contained, and when needed, are often also readily replaceable.

## 3. How Small is Your Zero Time?

The collective-execution mechanism of behavioral programming calls for the synchronization of all b-threads prior to triggering an event. This synchronization simplifies programming by avoiding race conditions and, combined with event blocking, facilitates the incrementality and naturalness of behavioral programming. However, such synchronization implies that system performance is constrained by the time it takes, at each step, for the slowest behavior to reach the synchronization point.

Thus, in behavioral programming there is a convention that b-threads are allowed to take only a small amount of time between synchronization points. Note that this is a rather standard convention used, e.g., for listeners in GUI frameworks and interrupt handlers, where callback routines are expected to complete quickly. When interaction with the physical world is involved, we advocate using the concept of *logical execution time* (LET) proposed in [34], which relies on a "fast execution" assumption for separation of observable external behavior of a task from its physical execution.

When the application has a large number of such "fast" routines (as, e.g., in behavioral programming where "everything is a behavior") the question of how small the processing time should be is crucial. In way of dealing with this question, we observe that smallness is relative, and an application may contain behaviors that operate at different time scales. Examples of these include, among others, (1) handling external events whose source is not synchronized with the application; (2) interacting with (and waiting for) resources whose response time (including communications time) is slower than the internal event-rate of other behaviors; (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 behavioral programming will be exemplified in Sections 5, 6, and 7, respectively.

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

While answers to the multi-time-scale challenges are discussed at length in the coming sections — the answer to this second question is simple and immediate: This is indeed an application bug just like any other bug! The situation may be compared to that of a standard, non-behavioral, application where an infinite loop in a high priority process causes it to monopolize all system resources, stalling other processes. Nevertheless, the architecture proposed for addressing the first two questions may mitigate the disastrous effects of such an application error, allowing the system as a whole to continue operating, and perhaps even recover. Also, tools like the model checker presented in [33] and the trace visualizer presented in [20] help in catching such bugs.

In the next section, we propose ways to program behavioral applications at multiple time scales. While the issue is, of course, dealt with in other contexts, our challenge is to solve it while maintaining the naturalness and incrementality facilitated by behavioral programming, allowing behavioral systems where behaviors are not required to synchronize with all other behaviors at every step.

## 4. Dealing with Multiple Time Scales

In this section we present technical solutions that allow for coding all significant application logic only in well-encapsulated b-threads. The solutions are mostly adapted from established techniques and designs — some general, e.g., agent-based programming [36] and some specific to behavioral programming, including especially the Interplay method [7] for connecting multiple LSC Play-Engines via external events. Our goal is to benefit from natural and incremental development using behavioral programming, while accomplishing a level of decentralization, and possibly physical distribution, where not all behaviors need to be fully synchronized.

### 4.1 Handling external events

For synchronization with an external environment, we propose a solution based on the *super-step* approach, as in statecharts [24] (used also in, e.g., LSC execution [23]) and on the logical execution time (LET) concept used, e.g., in GIOTTO [34]. This solution is implemented by the behavioral programming 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, where 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 but are 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 [42]).

According to this philosophy, it is sufficient to consider a real-world occurrence of an external event only in relation to when super-steps begin and end (only these points in the computation have meaningful time-stamps). When sampling times need to be equidistant, as is the case when implementing algorithms that come from control theory, one can program the systems such that all super-steps take a constant amount of time, by adding delays (under an assumption that the sequence of internal events runs fast enough to always complete within the time frame).

One initial approach (which we modify later) to the implementation of the above philosophy is to let a lowest-priority, dynamically-added b-thread introduce the external event. For example, consider the implementation of the Tic-Tac-Toe game in [32]. An independent, non-behavioral Java thread controls the GUI. Whenever the player clicks a board square, the GUI process creates a new lowest priority b-thread which requests a corresponding behavioral `click` event. The request occurs in the first and only synchronization point of the b-thread, after which the b-thread terminates. Since the added b-thread is assigned the lowest priority, it is active only when no internal event is enabled. Even when multiple external arrive in close succession, they are always introduced into the behavioral process in successive, distinct, super-steps.

This approach is general and simple, though, depending on the implementation, it may present performance issues related to the creation of threads. Also note that the semantic definition in Section 2, as well as the model-checking tool, BPmc, presently assume that the set of b-threads is constant throughout the execution.

We thus modify the approach to allow the b-threads that introduce external events to participate in the entire run, in a manner similar to the way LSC execution is open to handle user interactions and other external events only at the end of a super-step:

- External events are first captured by non behavioral processes and are placed in a common queue, using standard (non-behavioral) programming constructs.
- A lowest priority b-thread repeatedly requests a predesignated event, called, say `idle`, which by convention (which can be enforced), is never blocked by other b-threads. This event marks the end of a super-step.
- A designated b-thread repeatedly waits (behaviorally) for the event `idle`, and then "peeks" at the external event queue using standard programming constructs. This peeking may be slightly delayed by other queue accesses but the delay is acceptable, as no internal events are enabled. If no new external event is found, the b-thread waits for another `idle` event. If an external event is found, the b-thread requests a corresponding behavioral event that represents the external one.

Alternatively, the first and third points above can be replaced by allowing a single b-thread to behaviorally wait for an `idle` event, and then using existing language constructs to wait for external events, deferring indefinitely the next synchronization point.

We suggest announcing the conclusion of a super-step using the `idle` event, rather than performing the activities directly at the lowest priority b-thread, to allow a richer variety of actions in separate b-threads. Such actions may include super-step logging, deadlock handling, blocking of external events, and additional peeking at external event queues.

### 4.2 Accommodating behaviors at different time scales

Many applications that include behaviors of different time-scales can be decomposed as follows.

The application is decomposed into groups of behaviors, such that all the behaviors in a group are on the same time-scale. If necessary or desired, such groups can be still broken down into sub-groups, say according to physical components or relevant events. Note that the behaviors in each such sub-group are, by definition, of the same time-scale. We call such a group of behaviors a *behavior node*, or *b-node*, for short.

All the b-threads in each b-node run in a synchronized manner as described in Section 2. Note that the b-threads in a b-node may run on multiple cores or computers, where the operating system, JVM, Erlang, etc., facilitate the constant inter-b-thread synchronization implemented in the behavioral programming collective-execution mechanism. This paper is focused on inter-b-node coordination and not on intra-b-node parallelism.

Communication between b-nodes is only through external events.

External events 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 broadcast it to all other b-nodes) using some agreed upon protocol. The protocol can be either general for all behavioral applications in a given host language (and perhaps supplied with the behavioral programming infrastructure), or domain specific (e.g. for wireless sensor networks of some kind), or even application-specific.

**Receive:** Per the design in Section 4.1, for each b-node a designated non-behavioral process listens to all external events directed at that b-node, and places them in b-node-specific queue. A designated b-thread peeks at this queue at the end of each super-step and requests a corresponding behavioral event.

B-threads that depend on external events, are programmed to wait (behaviorally) for the corresponding behavioral event.

When the desired behavior of a b-node is not naturally associated with super-steps, the b-node may "open a window" for accepting external events in other ways, e.g., by periodically requesting an event that replaces the `idle` event above, and is waited for by the b-thread responsible for peeking at the external-event queue.

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 (internally 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. As blocking is a key enabler of incrementality in behavioral programming, the ability to propagate event-blocking is an important feature of the proposed decentralized architecture.

This design for communication between behavioral program reflects several assumptions that we believe are common and natural in development situations. First, consider a decentralized node busily working autonomously. When an external event arrives that is expected to change the behavior of the node, it is acceptable that one or more autonomous events 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 that are tolerated when sensing events or handling interrupts in computer systems.

The second assumption is that even a node that wishes to be extremely attentive to external events should not be synchronized with the source of these events, say, as b-threads are synchronized within a b-node. Consider a manager-employee analogy for the relationship between two b-nodes. Employees that follow their managers everywhere are inefficient and disruptive. Put in other words, excessive synchronization between b-nodes reduces parallelism in the system. We believe that the proposed design nicely balances the usage of autonomous behavioral components with efficiency. In particular we expect that the delay in reacting to messages is tolerable in a computer application that is decomposed according to behaviors, as it is in the life of a corporation.

## 5. Example 1: Synchronizing Behavioral Programs with an External Environment

To demonstrate how a behavioral program handles external events, we describe now the architecture and implementation of a small computer game. Its goal is to land a rocket back on its landing pad. The rocket descends from the top of the screen towards the bottom. The player can move the rocket left and right, or fire a short exhaust burst, that delays the fall. The player cannot, however, move the rocket upward. The landing pad is located on the ground, and moves left and right randomly. If the player manages to place the rocket on top of the landing pad, he or she wins. If the rocket hits the ground, the player loses. Figure 5 shows the player interface of the game.

### 5.1 Game Architecture

We implemented the game application in Erlang, using WxErlang as the GUI toolkit, and the `bp` module as the behavioral programming library. This single b-node application consists of the processes described below and is depicted in Figure 6.

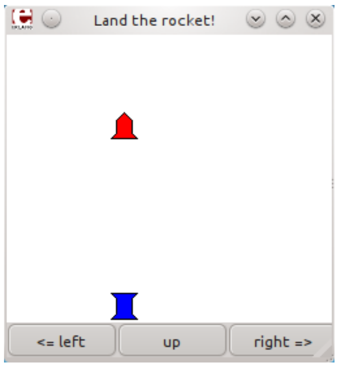

**Figure 5.** The computer game. The rocket, shown on the top part, should land on the pad, shown at the bottom. The player may move the rocket sideways, or fire a short burst that suspends its fall.

- The `rocket` and `landing_pad` processes maintain and draw the positions of the rocket and the landing pad as determined by rocket and pad motions.
- A user-interface controller process, 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.
- The `ticker` process sends the native event `tick` to the `queue` process every $N$ milliseconds.
- The `idler` b-thread process detects the ending 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 its native messages mailbox (or waits for a message if it is empty), and broadcasts the external event to the other b-threads by requesting a corresponding behavioral event, using the following loop:

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

- B-threads such as `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.

### 5.2 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, and are replaced with a textual description. The full code for the example can be found at `http://www.cs.bgu.ac.il/~geraw`.

The first behavior reacts to events representing the user's actions by requesting events that represent rocket moves in the desired direction. In our example, it translates the event, e.g., `user_left`, to `left`. However, in the general case the translation may be more complex, e.g., requesting several events per user command.

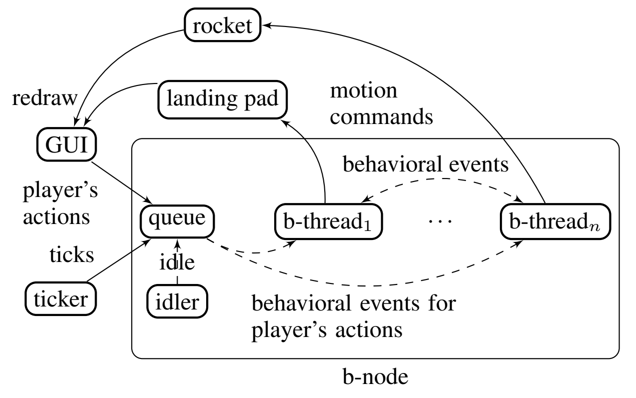

**Figure 6.** Handling external events in the game application. Solid arrows mark native Erlang messages. Dashed arrows mark triggering and waiting for behavioral events. Boxes are processes. The processes inside the b-node box are b-threads.

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

```
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.

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

In addition to moving the rocket sideways, we also want to allow the player to suspend the fall for one turn, simulating an exhaust burst. The `go_up` behavior responds to the `up` event, and suspends the rocket by blocking its downwards 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().
```

Now, we would like to add some restrictions to the game. The following behavior prevents the rocket from moving beyond the left side of the game board. Preventing the rocket from crossing the right border is done in the same way, replacing the 0 coordinate with some right limit $M$, and changing event names accordingly. Preventing the rocket from moving below the lower border is done similarly, by checking the vertical, rather than horizontal position of the rocket, and blocking the `down` event when it reaches 0.

```
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.
```

To make the game less trivial, we want to prevent the rocket from making too many maneuvers. To this end, we limit the movement of the rocket to either one step left or right per turn.

```
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 is 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 either `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 are responsible for detecting and handling winning or losing the game. Note that the `detect_win_lose` function below keeps track of the entire game board solely by listening to movement events. While this might seem somewhat wasteful, it prevents potential race conditions, due to accessing shared resources, such as the current positions of game elements.

```
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 above `win` or `lose` events, and then blocking all movement events indefinitely. The example program ends when the user closes the application window.

***Summary*** This illustrates the decomposition of the game into separate, independently programmed behaviors, and the introduction of external events at the end of super-steps. The power of behavioral programming 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 long sequence of predetermined right- and left-moves of the landing pad, that represent the secret plan of an adversary played by the computer.

## 6. Example 2: Coordinating behaviors with different time scales

In this section we demonstrate how an application can be composed of behavioral components 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 7). The behaviorally-programmed

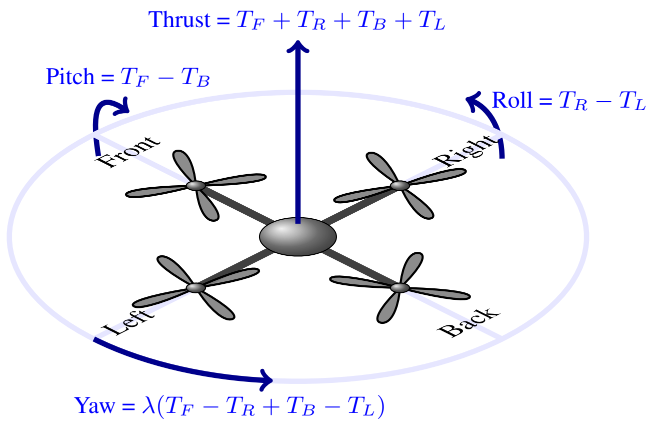
Thrust = $T_F + T_R + T_B + T_L$
Pitch = $T_F - T_B$
Roll = $T_R - T_L$
Front
Right
Left
Back
Yaw = $\lambda(T_F - T_R + T_B - T_L)$

**Figure 7.** 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 relation between these thrusts and the forces are indicated as the four formulas near each force. The coefficient $\lambda$ 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).

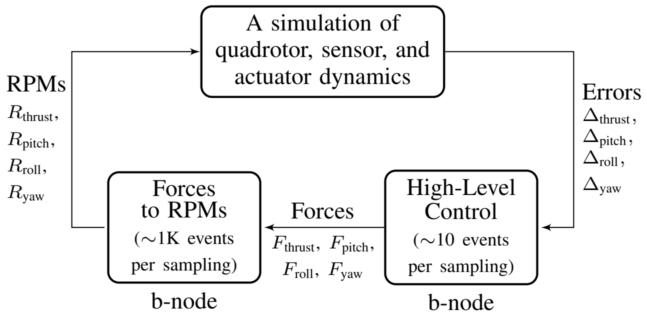
A simulation of quadrotor, sensor, and actuator dynamics
RPMs
$R_{\text{thrust}}$, $R_{\text{pitch}}$, $R_{\text{roll}}$, $R_{\text{yaw}}$
Errors
$\Delta_{\text{thrust}}$, $\Delta_{\text{pitch}}$, $\Delta_{\text{roll}}$, $\Delta_{\text{yaw}}$
Forces to RPMs
(∼1K events per sampling)
b-node
Forces
$F_{\text{thrust}}$, $F_{\text{pitch}}$, $F_{\text{roll}}$, $F_{\text{yaw}}$
High-Level Control
(∼10 events per sampling)
b-node

**Figure 8.** Block diagram for the quadrotor case-study. A simulation of the quadrotor is composed with two behavioral components. The first component, High-Level Control takes the differences between actual and desired thrust, pitch, roll and yaw and dynamically translates them to forces that are needed to be applied for correcting the displacements. The second component, Forces to RPMs, translates these forces to rotor speeds (RPMs) that should generate such forces. The dotted line indicates the boundaries of the behavioral program that we developed for this paper (the two behavioral modules, their interaction with the environment, and the communication between them.

piece is responsible for stabilizing the aircraft. It was experimentally tested by plugging behavioral modules into the comprehensive quadrotor-control simulation model developed by Bouabdallah et al. [10, 11]. This model is based on MATLAB/Simulink and simulates full control of the quadrotor flight including physical aspects.

The model of Bouabdallah et al. is modified, as illustrated in Figure 8. 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 7). It consists of four b-threads: one for each of the yaw, pitch and roll axis and one for trust. 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 given, e.g, from the remote control or dictated by a higher level navigation algorithm.

```
do forever {
    event = waitFor(an event with desired roll force);
    attainTarget(event.val,
        {RightRPMAdd, LeftRPMSub},
        {RightRPMSub, LeftRPMAdd}
    );
}
```

**Figure 9.** Pseudo-code for a force-control b-thread for roll. The b-thread waits for an event that indicates a desired (target) force in a particular direction (roll, in this case). It then calls a method `attainTarget` with parameters that indicate the events that increase the controlled force (second parameter: {`RightRPMAdd`, `LeftRPMSub`} and events that decrease it (third parameter: {`RightRPMSub`, `LeftRPMAdd`}). The events represent adding to or subtracting from the RPM of the front, back, right or left rotors as their names suggest. Three additional b-threads (not shown), `PitchBT`, `YawBT` and `ThrustBT`, control the three other forces in a similar manner. They are identical except for the parameters to attainTarget. The events that increase the controlled forces are: for pitch: {`FrontRPMAdd`, `BackRPMSub`}; for yaw: {`FrontRPMAdd`, `BackRPMAdd`, `RightRPMSub`, `LeftRPMSub`}, and for thrust: {`FrontRPMAdd`, `RightRPMAdd`, `BackRPMAdd`, `LeftRPMAdd`}. In all cases, the events that decrease the controlled force are the opposite events.

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 think of how sound-mixing is done towards many competing goals, such as balancing the guitar and the piano, and controlling overall volume. This is done by small adjustments of the available controls (without attempting to solve mathematical equations).

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) increase or decrease of individual rotor RPM towards the target and, blocking events that work away from it, as shown in Figures 9 and 10. 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 rotors RPM and decreasing the back rotors 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 the code in Figure 10), 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

```
attainTarget(
  target, // Desired number of steps to target
  positiveE, // Events that go towards target
  negativeE, // Events that go against target
) {
  // Initially we are -target away from target
  delta = -target;

  do {
     requestedE = blockedE = emptySet;

     // Request the events that will fix the
     // current deviation
     if delta < 0 then requestedE=positiveE;
     if delta > 0 then requestedE=negativeE;

     // If deviation exceeds slack,
     // block further deviations
     if delta < -SLACK then blockedE = negativeE;
     if delta > SLACK then blockedE = positive;

     // Place new bids and synchronize with the
     // other b-threads
     bSync(
        request = requestedE,
        wait = positiveE ∪ negativeE ∪ {Output},
        block = blockedE
     );

     if lastEvent in positiveE then delta++;
     if lastEvent in negativeE then delta--;
  } until (lastEvent is Output)
}
```

**Figure 10.** Pseudo code for the method `attainTarget` This is a common function that attains a desired difference between positive and negative events. It continuously requests and blocks events depending on the current deviation.

time, without interference from b-threads that try to create a difference between certain rotor RPMs.

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 these 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 11.

***Summary*** The quadrotor application uses very local behaviors, like controlling roll and pitch, in creating composite behaviors, like maintaining stability. Longer term behaviors such as traveling between cities in a multi-stop trip, or a 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 occasionally changing the required speed and direction, and does not require the constant synchronization and attentiveness of the stabilization modules described above.

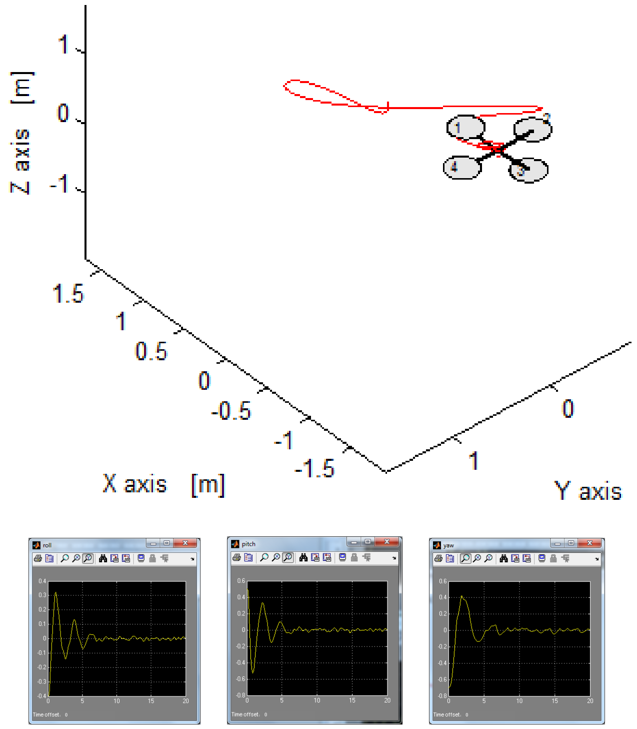

**Figure 11.** Simulated spatial path of the quadrotor, and plots of roll, pitch and yaw during the first 20 seconds of a behaviorally-controlled flight, as generated by the MATLAB model and tools of Bouabdallah et al. [10, 11].

The quadrotor also shows how behavioral programming can be used in developing hybrid control, i.e., a combination of discrete logic and actuation with sensing of continuous system variables [41]. In future work we will discuss the usage of behavioral programming in hybrid control, specifically, using fuzzy logic to translate continuous information into discrete categories that can be refereed to in naturally-programmed scenarios.

## 7. 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. 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. Then, we add to each vehicle 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 4. This small example can be readily extended for adding additional agents (vehicles, advisors) as well as new kinds of interactions among existing agents (inter-vehicle or inter-advisor).

### 7.1 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 in Section 5. 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 is also 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 if 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().
```

### 7.2 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. However, unlike in 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 b-thread 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 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:sync(#sync{wait=is(warn)}),
  {pos, _, Y} = bp:sync(#sync{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:sync(#sync{wait=[tick], block=[north]}) ||
   _ <- seq(1,10)],
  on_warning().
```

### 7.3 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 reports when a vehicle crosses a line, and later ask vehicles 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:sync(#sync{wait=is(pos)}),
  if
    Y >= Y1 -> % Report crossing
      bp:sync(#sync{request=[{cross, P}]});
    true -> ok
  end,
  monitor_line(Y1).
```

The second behavior listed below mixes behavioral and native communication methods. It maintains a list `L` of vehicles that have crossed the bound. 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 the full function of the `wait` clause. Figure 12 outlines the complete architecture of the vehicles application.

```
monitor_crossing(N, Y1, L) ->
  {cross, P} = bp:sync(#sync{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.
```

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

## 8. Related Work

The approach presented here for using behavioral programming in rich decentralized control, coexists with, complements, and leverages many 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], we observe that (a) generally, b-threads are not explicitly aware of each other, and communicate not via explicit messages but indirectly using the request/wait/block idioms; (b) behavioral programming focuses on

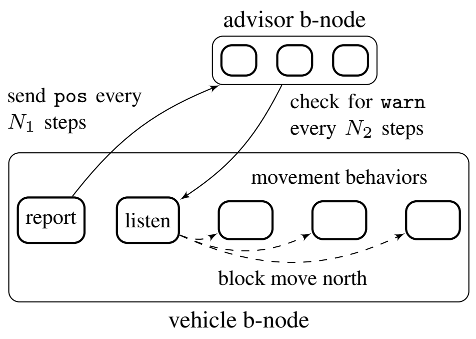

**Figure 12.** Multi-agents architecture for vehicles. The behavior `report` of each vehicle sends the current position every few steps to interested advisors. The `listen` behavior checks for a new warning, and may block specific events if it receives one.

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 model-checker in [33]); (c) in behavioral programming, asynchrony is allowed only between b-nodes, while within a b-node, behavior coordination imposes full synchronization (c.f., statecharts [24]). Further, In Agha [3, Chapter 6], there is a discussion of how the problems of deadlocks and divergence are dealt with the in the actor model, and why the problem of shared memory is a non-issue. In Section 2 we have a similar discussion for behavioral programming.

Behavioral programming principles have been implemented already in several environments and languages. It would be interesting to explore the synergy between behavioral programming and agent-oriented-specific languages, such as AgentSpeak and Jason [8], 2APL [18], GOAL [35], SIMPA [46], Indigolog [9], JIAC [9], and Axum [2]. Such synergy could emerge from interfacing agents and behavioral programs, from turning b-nodes into agents and using agent programming languages to handle communications between behavioral nodes, or from introducing blocking idioms into agent-oriented languages.

It should be noted that the proposed decomposition of complex systems into asynchronous b-nodes, each of which being comprised of synchronized b-threads, is targeted mainly at the logical organization of the system, towards manageable, incremental development. Performance gains that may emerge from parallel distributed execution are a secondary focus in the current context

In [5], Armstrong discusses the Erlang view of the world: "Everything is a process that lacks shared memory and influences one another only by exchanging asynchronous messages". As emerges from the our implementation in Erlang, the great possibilities in scalability and ease of programming stemming from implementing b-threads with Erlang processes are obvious. As to shared memory, indeed some of our analysis here and developments such as the BPmc model checker [33] assume that b-threads communicate only via events. Nevertheless, the usage of other host languages such as Java, does enable the developer to use shared memory and related features when desired.

Agents are often portrayed with human-like cognitive capabilities; e.g., Belief-Desire-Intention designs [13, 18, 45], goal oriented agents [35], autonomous agents with mental states [49], and agents with purpose and emotions [50]. Since the intelligence of an entity is often described through its handling of a given scenario, based on our experience so far we believe that behavioral programming idioms can support the programming of rich cognitive concepts in a simple, natural way.

In [37], 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, whole scenarios without requiring complex state management in the rules. Further, in behavioral programming event-blocking facilitates expressiveness and behavior composition.

Other behavior-based decomposition can be seen in behavior-based architectures in robotics and in hybrid-control, including Brooks's subsumption architecture [14], Branicky's behavioral programming [12], and leJOS [40], (see review in [4]) which construct systems from behaviors. Our behavioral programming approach may serve as a formalism, implementation or possible extension, of elements of such architectures.

The appendix of Bordini et al. [9] contains criteria for comparing agent-oriented platforms and languages. It would be interesting to check if and how the behavioral programming approach can be assessed subject to these criteria.

A coordination mechanism similar to behavioral programming (however without the crucial event-blocking) which also uses the term super-step, is proposed in [44].

## 9. Conclusion and Future Work

We have shown that connecting behaviorally programmed nodes using a simple messaging infrastructure, is a promising approach to the incremental development of complex systems.

The proposed architecture and our examples may also support a slightly different claim about behavioral programming. Composing a system purely from behavior threads raises concerns about efficiency and possible disastrous effects of small delays, emanating from the need to frequently synchronize all behaviors. We believe that many complex systems that may be originally conceived as purely behavioral, can be also nicely decomposed into b-nodes, such that the run-time synchronization requirements are substantially reduced. Further, we believe that this decomposition can preserve the natural, incremental development and the alignment of behavior modules with requirements.

Future work can progress in several directions: studying the impact of behavioral programming 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, leveraging behavioral programming concepts in an effort to advance general decentralized control metaphors.

## Acknowledgments

We would like to thank Einat Fuchs, whose lecture about coordination of cockroach locomotion [22] inspired some of the ideas in this paper. We thank Dan Brownstein and Nir Svirsky for their programming work on the MATLAB simulation and control software of the behaviorally-controlled quadrotor. Thanks to Amir Kantor for his valuable comments on the paper.

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

## References

[1] AGERE! workshop (Actors and Agents Reloaded). call-for-papers. http://apice.unibo.it/xwiki/bin/view/AGERE/ . 2011.

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

[3] G. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[4] R. Arkin. *Behavior-based robotics*. MIT Press, 1998.

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

[6] Y. Atir and D. Harel. Using LSCs for scenario authoring in tactical simulators. In *SCSC*, 2007.

[7] 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*, 2004.

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

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

[10] S. Bouabdallah. Design and control of quadrotors with application to autonomous flying. *Lausanne Polytechnic University*, 2007.

[11] S. Bouabdallah, P. Murrieri, and R. Siegwart. Design and control of an indoor micro quadrotor. In *ICRA*, 2004.

[12] M. Branicky. Behavioral programming. In *Working notes AAAI spring symp. on hybrid systems and AI*, 1999.

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

[14] R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 1986.

[15] A. Bunker, G. Gopalakrishnan, and K. Slind. Live sequence charts applied to hardware requirements specification and verification. *STTT*, 2005.

[16] P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. *Software and System Modeling*, 2008.

[17] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *J. on Formal Methods in System Design*, 2001.

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

[19] N. Eitan and D. Harel. Adaptive behavioral programming. *Conf. on Tools with Artificial Intelligence*, 2011.

[20] N. Eitan, M. Gordon, D. Harel, A. Marron, and G. Weiss. On visualization and comprehension of scenario-based programs. In *IPCP*, 2011.

[21] J. Fisher, D. Harel, E. J. A. Hubbard, N. Piterman, M. J. Stern, and N. Swerdlin. Combining state-based and scenario-based approaches in modeling biological systems. In *CMSB*, 2004.

[22] 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.

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

[24] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[25] D. Harel and I. Segall. Planned and traversable play-out: A flexible method for executing scenario-based programs. In *TACAS*, 2007.

[26] D. Harel and I. Segall. Synthesis from live sequence chart specifications. *Journal of Computer System Sciences*, 2011. To appear.

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

[28] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *FMCAD*, 2002.

[29] D. Harel, H. Kugler, and G. Weiss. Some methodological observations resulting from experience using LSCs and the play-in/play-out approach. *Scenarios: Models, Transformations and Tools*, 2005.

[30] D. Harel, A. Kleinbort, and S. Maoz. S2A: A compiler for multi-modal UML sequence diagrams. In *FASE*, 2007.

[31] D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: towards a comprehensive tool for scenario based programming. In *ASE*, 2010.

[32] D. Harel, A. Marron, and G. Weiss. Programming coordinated scenarios in Java. In *ECOOP*, 2010.

[33] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-checking behavioral programs. *EMSOFT*, 2011.

[34] T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using giotto. *Control Systems Magazine, IEEE*, 2003.

[35] K. Hindriks. Programmingrationalagents in goal. *Multi-Agent Programming:*, 2009.

[36] N. Jennings. An agent-based approach for building complex software systems. *CACM*, 2001.

[37] A. Kay. Programming and programming languages. Technical report, VPRI Research Note RN-2010-001., 2010.

[38] H. Kugler and I. Segall. Compositional synthesis of reactive systems from live sequence chart specifications. *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[39] H. Kugler, C. Plock, and A. Roberts. Synthesizing biological theories. *CAV*, 2011.

[40] LEJOS. Java for lego mindstorms. http://lejos.sourceforge.net/.

[41] D. Liberzon. *Switching in systems and control*. Springer, 2003.

[42] 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.

[43] S. Maoz and D. Harel. On tracing reactive systems. *SoSyM*, 2010.

[44] W. Miao and W. Tong. Agent based servicebsp model with superstep service for grid computing. 2007.

[45] 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.

[46] A. Ricci, M. Viroli, and G. Piancastelli. simpa: A simple agent-oriented java extension for developing concurrent applications. *LADS*, 2008.

[47] A. Sadot, J. Fisher, D. Barak, Y. Admanit, M. J. Stern, E. J. A. Hubbard, and D. Harel. Toward verified biological models. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 2008.

[48] B. Shimony, I. Nikolaidis, P. Gburzynski, and E. Stroulia. On coordination tools in the picos tuples system. In *2nd workshop on Software engineering for sensor network applications*, 2011.

[49] Y. Shoham. Agent-oriented programming. *Artificial intelligence*, 1993.

[50] M. Travers. Programming with agents: New metaphors for thinking about computation. 1996.

[51] G. Wiener, G. Weiss, and A. Marron. Coordinating and visualizing independent behaviors in erlang. In *Erlang workshop*, 2010.