Programming Coordinated Behavior in Java

David Harel[1], Assaf Marron[1], and Gera Weiss[2]

[1] Weizmann Institute of Science, Rehovot, Israel
[2] Ben Gurion University, Beer-Sheva, Israel

Abstract. Following the scenario-based approach to programming which centered around live sequence charts (LSCs), we propose a general approach to software development in Java. A program will consist of modules called *behavior threads (b-threads)*, each of which independently describes a scenario that may cross object boundaries. We identify a protocol and a coordination mechanism that allow such behavioral programming. Essentially, runs of programs are sequences of events that result from three kinds of b-thread actions: *requesting* that events be considered for triggering, *waiting* for triggered events, and *blocking* events requested by other b-threads. The coordination mechanism synchronizes and interlaces b-threads execution yielding composite, integrated system behavior. The protocol idioms and the coordination mechanism of b-threads are implemented as a Java library called BPJ. Throughout the exposition we illustrate benefits of the approach and discuss the merits of behavioral programming as a broad, implementation-independent paradigm.

1 Introduction

Behavioral and scenario-based descriptions are used in many different areas, including among others, system requirements and specifications, business workflow engines, user guides, and books of laws and regulations. In this paper we focus on applying such descriptions in computer programming. To facilitate coding of behaviors in an imperative language such as Java, we define the concept of *behavior thread* or *b-thread*. A b-thread is a programming construct that controls and coordinates the execution of a particular behavior, possibly involving multiple objects and components. We then introduce a Java library, BPJ, that enables the development of a system in Java as a collection of b-threads, each of which uses the library's API to independently execute a behavior or scenario assigned to it, while the synchronized and interlaced execution of all b-threads yields integrated, cohesive system behavior.

This work comes in the wake of existing languages for describing scenario-based behavior, such as *message sequence charts* (MSC) [22,32] and *live sequence charts* (LSC) [7,16]. In particular, the last dozen or so years have seen extensive work on LSCs and its underlying paradigm of scenario-based programming. One of the main lines of this research has lead to the possibility of using units of behavior as the backbone of the final executable [16]. Thus, in LSC, scenarios can become part of the application, going beyond the classical role of of conventional sequence charts as a requirements and specification language to be later tested against an implementation, typically of a completely different nature. To that end, the LSC language was designed to be rich in expressive power for general-purpose software development, and a central line of research has been to support *coordinated execution of multi-modal scenarios*[16]. Like modal verbs in a natural language and the operators of modal logic, LSC modalities can describe, among other things, what may, must and may not occur, and the total behavior is derived from combining all scenarios accordingly. For executing LSCs, the *play-out* method was devised, which constantly follows all active charts, and at each step selects an event that is enabled by some chart and does not violate any chart, and advances all charts waiting for that event [16].

The original play-out execution mechanism of LSCs (and its later "smart" versions [14,19]) were implemented in a special-purpose LSC tool, called the *Play-Engine*. Later, a compiler for LSCs was constructed, resulting in AspectJ code which can be interwoven into a Java application [27,13]. This brings executable scenarios closer to being usable in an actual system development environment. However, what still seems to be missing is a direct way for programmers to apply the ideas and methodology behind scenario-based programming in a conventional software development setting, without the need for a special purpose language or tool.

In this paper, we build upon the basis of scenario-based programming as established by LSCs, and the experience gained from experimenting with it in several application areas [2,4,5,9,15,33]. We first identify and provide formal definitions for the essence of the ideas (which we more generally call *behavioral programming* [17]). Then, we introduce the BPJ library and show that coordinated behaviors can be programmed in an ordinary programming language alongside the coding of individual objects with their own behavior. This programming can be done without relying on a special purpose language and environment, such as LSC and the Play-Engine, while maintaining an implementation that can reflect the behavioral descriptions as conceived by humans.

In contrast to previous work on LSCs, in the current paper we develop scenario-based programming bottom-up; i.e., we start from a general-purpose language (Java) and add programming tools that allow programmers to endow their programs with powerful behavioral elements. In particular, the focus here is on the coding and less on the specification, with the intention of bringing the advantages of scenario-based programming (e.g., incremental system construction, meaningful functionality at early development stages, and alignment with the way humans think) to broader programming contexts.

Outline: Section 2 outlines our approach. Section 3 defines general abstract semantics and idioms for behavioral programming and then introduces the BPJ library showing how the idioms are implemented in Java. In Section 4, features and advantages of the approach are illustrated through examples. A more detailed description of the BPJ library is given in Section 5. Section 6 briefly compares behavioral programming to other approaches and surveys related research. In Section 7 we conclude with a discussion of future research directions.

2 Outline of Our Approach

As a simple example, consider a program for an individual who wishes to drive from New York to Los Angeles. The program consists of two behaviors, or b-threads, that guide the individual along the trip: one b-thread contains the driving route and directions and the other is a daily plan for combining driving, eating and resting (see Figure 1).

```
— Start driving on I-78 W [135 mi]
— Merge onto I-81 S [36.6 mi]
— Take ramp onto I-76 W [152 mi]
— Merge onto I-70 W [613 mi]
— Merge onto I-44 W [497.2 mi]
— Continue to I-40 W [1,214 mi]
— Merge onto I-15 S [72.6 mi]
— Merge onto I-10 W [38.9 mi]
```

(a) Directions b-thread

```
Repeatedly:
— Drive for 5 h; look for restaurant
— Stop the car; have lunch
— Drive for 5 h; look for restaurant
— Stop the car; have dinner
— Drive for 2 h; look for hotel
— Stop the car
— Sleep until morning
```

(b) Day schedule b-thread

Fig. 1. Two independent behavior threads that run in a coordinated manner

Throughout the trip, the two b-threads are interwoven into what manifests itself externally as a single integrated behavior: actions from the two b-threads may run in parallel, such as when looking for a restaurant while driving the car along the route; two actions may be consolidated into a single one, as when driving according to schedule and according to directions; and two actions may suspend each other, as when not driving along the route while sleeping at a hotel. Note that the two b-threads do not communicate directly — in fact, they are quite oblivious of each other (e.g., consider the ease with which either b-thread can be thrown out and replaced with another, or not be replaced at all). The coordination and interweaving of actions is done by the individual, parallelizing, consolidating, and deferring actions as needed.

This seems like a natural approach to programming. It allows one to specify a desired complex behavior by decomposing it into an set of simpler, relatively independent behaviors, that can be executed collectively in a meaningful way.

The methodology proposed in this paper is to program such b-threads in Java. For example, we can code them as in Figure 2, where driving is a continuous activity that can be suspended and resumed; road signs, milestones, and time-ticks are external events that can be waited for; function invocations such as the calls to `turnOnto()` or `haveLunch()` trigger external actions; and programming constructs such as variables and loops are used in the usual way. The coordination between the b-threads is implicit through their control of the driving activity.

More generally, in our programming approach, coordination is done through signalling using event objects. Using the BPJ library we describe later, b-threads can request the triggering of events, can wait for events, and can block the

```
directions() {
    start(driving);

    for(d=0; d<135; d++) {
        waitFor(oneMile);
    }

    waitFor(I81S);
    turnOnto(I81S);

    for(d=0; d<36; d++) {
        waitFor(oneMile);
    }

    waitFor(I76W);
    . . .
}
```

```
daySchedule() {
    while( true ) {

        for(t=0; t<5; t++) {
            waitFor(oneHour);
        }

        waitFor(restaurant);

        suspend(driving);
        waitFor(fullStop);

        haveLunch();
        resume(driving);
        . . .
    }
}
```

Fig. 2. Writing behavior threads in Java

occurrence of events requested by other b-threads. For example, in the code in Figure 2, the starting, suspending and resuming of the driving operation is a higher level interface that hides the requesting and blocking of events. BPJ includes a coordination mechanism that weaves these seemingly independent requests into a single combined cohesive behavior.

In our technique, behavior threads are implemented as Java threads that request, wait for, or block events using the BPJ API consisting of method calls and simple data structures (the b-threads can also call other functions for external actions). The coordination mechanism fulfils the instructions of the b-threads by triggering those of the requested events that are not blocked.

Our "request-and-block" abstract idioms and the BPJ API may be reminiscent of some of the constructs used in approaches to concurrency (such as various kinds of locks, semaphores and message passing), however differences will show up as our exposition unfolds in the coming sections. The main one is the apparent absence in those other approaches of a concise, autonomous way for a process to block events that other processes may attempt to trigger. Second, our approach enables unification/consolidation of identical event requests, coming from different processes, into a single event execution. Another difference that will be shown is that in BPJ requests and blocks are triggered only when all b-threads are ready to process them, as opposed to queuing associated with protocols such as publish/subscribe.

Now, while our work focuses on constructs for *programming* b-threads in a textual language like Java, we find it useful to present the new idioms also in a formal definition based on transition systems. For example, the transition system in Figure 3 represents the trip b-threads. In such a transition system, the events are those that the corresponding program explicitly waits for, and the states represent the states of the program (program counter and variable values) when waiting for the next event. External actions carried out by the program are drawn inside the state rectangle, and are assumed to be executed as part of the transition into the state.

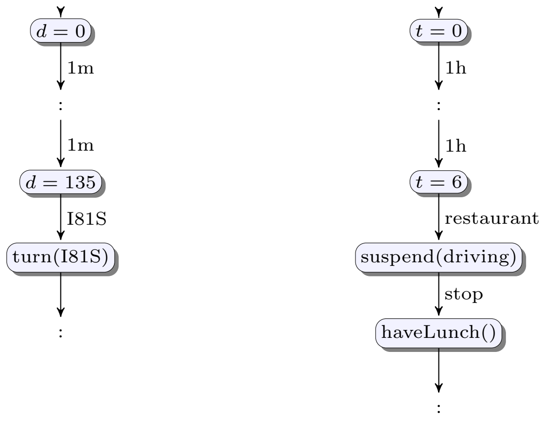


Fig. 3. Transition systems for the NY-to-LA trip behavior threads

Note that because of the nature of transition systems (e.g., states are represented explicitly), the number of states may grow excessively, so we do not propose them as a programming language. Instead, transition systems will be used throughout the paper in formal definitions and analysis, as well as in visual representation of simple programs, and will introduce and complement the discussion of the Java programming implementations.

3 Interlacing and Synchronizing Behaviors

The two main idioms proposed in this paper for usage by b-threads, relate to events, and they are *Request* and *Block*. Requesting an event is similar to executing a command in an imperative language, except that the actual triggering of the event is tentative, and can occur only when the event is not blocked. Blocking an event is an operation that suspends the occurrence of events that might be requested by other b-threads. If a blocked event is requested, its triggering is deferred until it is no longer blocked, or "forever", that is, until the entire system terminates. These idioms are complemented by a third idiom, *wait for* an event (also referred to as *watch* an event), that has the common semantics of asking to be notified or resumed when the event is triggered.

3.1 Abstract Semantics

Our definition of b-threads and their collective execution is based on labeled transition systems. Recall that a labeled transition system is defined (see [24]) as a quadruple $\langle S, E, \to, init\rangle$, where S is a set of states, E is a set of events, $\to$ is a transition relation contained in $(S \times E) \times S$, 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 $s_{i-1} \xrightarrow{e_i} s_i$.

Below, we formally define a b-thread as a labeled transition system in which events in each state can be marked as *requested* or as *blocked*. We then set the two basic rules for executing a set of b-threads: (1) An event occurs if and only if it is requested by some b-thread and is not blocked by any b-thread; (2) All b-threads affected by a given event undergo a state transition when the event occurs.

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

Definition 2 (runs of a set of b-threads). *We define the runs of a set of b-threads $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i\rangle\}_{i=1}^n$ as the runs of the labeled transition system $\langle S, E, \rightarrow, init\rangle$, 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 $\rightarrow$ includes a transition $\langle s_1, \ldots, s_n\rangle \xrightarrow{e} \langle s_1', \ldots, s_n'\rangle$ if and only if*

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

and

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

In general, there may be more than one run of a set of b-threads, depending on the order in which events are selected from the set of requested and unblocked events. We view this as a useful feature, which allows designers of systems to separate the specification of possible b-threads from the process of prioritization and choice of events.

To introduce predictability and repeatability into runs of a set of b-threads, we require a total (possibly dynamic) order of events, which controls the selection of the next event to be triggered. For simplicity, in this paper, the event selection order is induced by associating each b-thread and each event with a fixed numerical priority; a lower number means higher priority. Requested events are then ordered lexicographically: first by the priority of the requesting b-thread and then by the priority of the event itself.

As to graphical notation, we adopt the following for drawing diagrams of labeled transition systems. Rectangles and labeled arrows represent, respectively, states and conditional transitions. Within each state we specify the associated sets of requested events (denoted by $R = \cdots$) and blocked ones (denoted by $B = \cdots$). Above each b-thread we specify its priority.

3.2 A Java Implementation

The basic programming unit for implementing behavioral programming in Java is the coding of behavior threads as Java threads. For that purpose BPJ defines the `BThread` class, each instance of which is associated with a Java execution thread whose progress embodies a run of the behavior thread. The logic of the b-thread is implemented in a method `runBThread` provided by the programmer.

The events triggered by the b-threads are represented as objects of the class `Event`. Each b-thread object has three sets of events: *requested events* (`requestedEvents`) and *blocked events* (`blockedEvents`) correspond to the sets R and B in definitions 1 and 2 above, and *watched events* (`watchedEvents`) corresponds to the set of events appearing in the labels of the arrows in the transition diagram. Each b-thread calls the behavior synchronization method `bSync` passing to it as parameters information about the events it requests, waits for, and/or blocks. The method `bSync` also synchronizes the b-thread with all other b-threads and invokes the control mechanism that chooses and triggers the next event and notifies the b-threads waiting for that event.

The transition relations in Definitions 1 and 2 are represented in the Java code of behavioral programs implicitly. A thread of Java code that interfaces with other programs only through events can be mapped to the b-thread $\langle S, E, \rightarrow, init, R, B\rangle$ along the following lines:

– S is the set of states reachable by the Java execution thread, when its program counter is at a `bSync` call. Here, the program state is interpreted in the standard way, as the set of variable values plus the program counter.
– E is the set of event objects referred to in the Java code.
– The transition $s \xrightarrow{e} s'$ exists iff calling `bSync` when in state s implies that if `bSync` returns and sets the variable `lastEvent` to e then the process will be in state s' upon its next call to `bSync`. The set of all such events e for the current state s of the b-thread is reflected in the corresponding parameter to the method `bSync`.
– $init$ is the state associated with the first call to `bSync`.
– $R(s)$ and $B(s)$ are the values of the corresponding parameters to the method `bSync` when the thread is in state s.

While translations between the two representations can potentially be automated, transition systems are not intended to be used for programming (as mentioned above) because of their verbosity in representing states. As a medium for representing b-threads, code is most often more succinct and understandable, as we show below. Moreover, b-threads coded in Java can benefit from the full power of the language, as discussed in Subsection 5.3.

Synchronization and Interlacing Algorithm. The state transitions in Definition 2 of collective execution of b-threads are implemented in an algorithm that synchronizes, suspends and resumes all b-threads, as depicted in Figure 4.

1. When a b-thread is started, it is registered as an executing (running) b-thread and is marked as not ready to synchronize with other b-threads. The method `runBThread` is then started in a dedicated Java execution thread.

2. The `runBThread` method repeatedly: (a) performs arbitrary processing, and (b) calls `bSync`, passing to it as parameters new values for the b-thread's sets of requested events, watched events and blocked events.

3. When `bSync` is called:

   (a) The parameters to `bSync` are copied into the corresponding variables of the b-thread object (`requestedEvents`, `watchedEvents`, and `blockedEvents`).

   (b) The b-thread is marked as ready to synchronize, and its Java execution thread is suspended.

   (c) When all running b-threads are marked as ready to synchronize, an event selection mechanism is invoked, that carries out the following:

      i. It searches for, and selects the first event that is in the `requestedEvents` set of some b-thread, and is not in the `blockedEvents` set of any b-thread. The order of this search is by the priority of the requesting b-thread, and within a given b-thread, by the order of the `requestedEvents` set.

      ii. It resumes all b-threads that contain the selected event in their `watchedEvents` set or `requestedEvents` set and marks all of them as not ready to synchronize.

4. When the `runBThread` method terminates, the b-thread is removed from the list of executing b-threads.

Fig. 4. Synchronization and interlacing algorithm

This execution algorithm was inspired by the operational semantics of the Play-Engine, which executes an LSC specification by maintaining the current locations (cuts) of all triggered charts and selecting the next event based on a variety of constraints as described in [16]. Specifically, the Play-Engine's central mechanism of advancing the LSC charts along lifelines while avoiding violations corresponds to BPJ's sequential execution of Java threads that request events that would generate advancement in an LSC chart, and block events that may cause violations of an LSC chart.

4 Features of Behavioral Programming - Through Examples

Before we systematically describe the elements of behavioral programming in Java, we introduce its main features through three examples. The first one illustrates what Java code for b-threads looks like, the second illustrates how independent b-threads can be incrementally assembled into an integrated application, and the third highlights the power of behavioral programming in development of reactive systems, and in error detection and correction. The full source code for the examples is available online at the library web site [18].

4.1 Example 1: A Simple Application

In Figure 5 we show the source code for a program that expands on the classical "Hello, World!" example. This behavioral program generates two independent sequences of greetings and then forces the issuance of the greeting in one interwoven sequence. The components of the program are described in the order of their coding.

1. `goodMorning` and `goodEvening` are instances of class `Event`, and are to be requested and waited for by the b-threads below. The information they carry is through their implementation of the method `toString`.
2. `sayGoodMorning` is a b-thread that issues the event `goodMorning` 10 times. This b-thread repeatedly requests the triggering of the event and waits for its occurrence.
3. `sayGoodEvening` is a b-thread that issues the event `goodEvening` 10 times. This b-thread is similar to `sayGoodMorning`, and illustrates independent development of a separate, autonomous behavior thread. Initially, we want all `goodMorning` events to precede all `goodEvening` events thus we set the priority of `sayGoodEvening` to be lower than that of `sayGoodMorning`.
4. `Interleave` is now added to force alternating the triggering of the `goodMorning` and `goodEvening` events. This b-thread alternatingly blocks each of the two events, illustrating how independent behaviors can be coordinated in BPJ.
5. `DisplayEvents` is a b-thread that watches for all events and displays corresponding messages. It illustrates waiting for events and how b-threads can translate events to actions (in this case simple printout).
6. `main` is the entry method of the application. It creates and starts the b-threads and assigns priorities. This method is modified as b-threads are added. The final output of the program is shown in Figure 6.

The method `bSync` used by the b-threads registers the calling b-thread's intent to request the triggering of the event(s) in the first parameter, wait for the event(s) in the second parameter, and block the event(s) in the third parameter. As detailed above, the method `bSync` synchronizes the calling b-thread with all other b-threads. When all b-threads are synchronized, an event triggering mechanism is invoked; it selects the first requested event that is not blocked and resumes all b-threads waiting for that event. Under some circumstances, explained in Section 5.2 this method throws `InterruptedException`, hence the class definitions require the corresponding `throw` clause.

```
import static bp.eventSets.EventFilterConstants.all;
import static bp.eventSets.EventFilterConstants.none;
import bp.BProgram;
import bp.BThread;
import bp.Event;

class HellowWorld {
    static Event goodMorning = new Event() {
        public String toString() {
            return "Good Morning!";
        }
    };
    static Event goodEvening = new Event() {
        public String toString() {
            return "Good Evening!";
        }
    };
    static class SayGoodMorning extends BThread {
        public void runBThread() throws InterruptedException {
            for (int i = 1; i <= 10; i++) {
                bpSync(goodMorning, none, none);
            }
        }
    }
    static class SayGoodEvening extends BThread {
        public void runBThread() throws InterruptedException {
            for (int i = 1; i <= 10; i++) {
                bpSync(goodEvening, none, none);
            }
        }
    }
    static class Interleave extends BThread {
        public void runBThread() throws InterruptedException {
            while (true) {
                bpSync(none, goodMorning, goodEvening);
                bpSync(none, goodEvening, goodMorning);
            }
        }
    }
    static class DisplayEvents extends BThread {
        public void runBThread() throws InterruptedException {
            while (true) {
                bpSync(none, all, none);
                System.out.println(BProgram.lastEvent);
            }
        }
    }
    public static void main(String[] args) {
        BProgram.add(new SayGoodMorning(), 1.0);
        BProgram.add(new DisplayEvents(), 2.0);
        BProgram.add(new SayGoodEvening(), 3.0);
        BProgram.add(new Interleave(), 4.0);

        BProgram.startAll();
    }
}
```

Fig. 5. A simple application - "Hello, World!"

4.2 Example 2: Incremental Development of a Full Application

To further demonstrate the overall behavioral-programming approach, and especially the role of collective execution of b-threads, we describe an incremental development of a program for the game of Tic-Tac-Toe. To highlight the

```
Hello World Event: Good Morning!
Hello World Event: Good Evening!
Hello World Event: Good Morning!
Hello World Event: Good Evening!
...
Hello World Event: Good Morning!
Hello World Event: Good Evening!
```

Fig. 6. Output of "Hello, World"

intuitive nature of the approach, the development stages are aligned with steps often taken when teaching the rules and strategy of the game to someone who is not familiar with it (perhaps a child; see [6]). At various points, we point out useful features and capabilities of behavioral programming.

For clarity of the illustrations and to emphasize the principles we show most of the b-threads as transition systems rather than in code.

The game involves two players, X and O, who take turns marking squares on a 3×3 grid whose squares are identified by pairs of rows and columns: $\langle 0,0\rangle, \langle 0,1\rangle, \ldots, \langle 2,2\rangle$. The player who manages to form a full horizontal, vertical or diagonal line with three of his/her marks wins. In our example, the first player is the user, X, and the second is the computer, O.

Programming the Basic Rules. Our first step in programming the game would be to specify its environment and the basic moves. In our implementation of the game, each marking of a square by a player is represented by an event $\mathsf{O}_{\langle row,col\rangle} \in \mathit{AllOs} = \{\mathsf{O}_{\langle 0,0\rangle}, \mathsf{O}_{\langle 0,1\rangle}, \ldots, \mathsf{O}_{\langle 2,2\rangle}\}$ and $\mathsf{X}_{\langle row,col\rangle} \in \mathit{AllXs} = \{\mathsf{X}_{\langle 0,0\rangle}, \mathsf{X}_{\langle 0,1\rangle}, \ldots, \mathsf{X}_{\langle 2,2\rangle}\}$. Four additional events, called XWin, OWin, draw, and gameOver are introduced to capture the victory of the respective player, or a draw, and the subsequent conclusion of the game.

A game is played as a sequence of events; e.g., $\mathsf{X}_{\langle 0,0\rangle}$, $\mathsf{O}_{\langle 1,1\rangle}$, $\mathsf{X}_{\langle 2,2\rangle}$, $\mathsf{O}_{\langle 0,2\rangle}$, $\mathsf{X}_{\langle 2,0\rangle}$, $\mathsf{O}_{\langle 2,1\rangle}$, $\mathsf{X}_{\langle 1,0\rangle}$,XWin describes a game round in which X wins, and its final configuration is:

X $\langle 0,0\rangle$	$\langle 0,1\rangle$	O $\langle 0,2\rangle$
X $\langle 1,0\rangle$	O $\langle 1,1\rangle$	$\langle 1,2\rangle$
X $\langle 2,0\rangle$	O $\langle 2,1\rangle$	X $\langle 2,2\rangle$

The next step in developing our Tic-Tac-Toe program (or in teaching the novice player) is to articulate the goals and constraining rules of the game. This is done by B-Threads 1-4, which we now describe.

The goal of the first b-thread is to detect a situation where X won. In BPJ this can be designed in several different patterns. For example, a single b-thread can wait for all events, keep track of the board configuration and reexamine it with every event; or, each of 8 b-threads can be responsible for one of the 3 rows, 3 columns and 2 diagonals and watch for the 3 desired X events in any order; or, in a third design pattern, a dedicated b-thread may watch for the 3 desired X events in one of their 6 possible permutations (total of 48 b-threads).

To accomplish the second and third design patterns one does not have to maintain Java code for multiple similar b-threads. A single `runBThread` method can be coded with appropriate variables as part of the b-thread class definition, and multiple instances can be created using iteration, while controlling the variations through parameters passed when the b-thread is created or started. In diagrams of transition systems, multiple similar b-threads are represented by a single template b-thread, which includes a *scope*, a specification summarizing the finite set of concrete b-threads that are described by the template diagram.

The first b-thread in this example follows the third design pattern, and is embodied in 48 b-thread instances. Each instance watches out for one ordered triplet of events by player X (e.g., $\mathsf{X}_{\langle 2,2\rangle}, \mathsf{X}_{\langle 1,1\rangle}, \mathsf{X}_{\langle 0,0\rangle}$), announcing X's victory when it is completed. B-Thread 2 performs the same function for player O.

B-Thread 1 (`DetectWinByX`) :
48 instances with priorities 1.001, . . . ,1.048

Scope :
an instance for each ordered triplet $\langle x_1, x_2, x_3\rangle$ of X events that fill a horizontal, vertical, or diagonal line.

$$\begin{tikzcd}
{} \arrow[r] & \boxed{\begin{array}{l}R=\emptyset\\B=\emptyset\end{array}} \arrow[r, "x_1"] & \boxed{\begin{array}{l}R=\emptyset\\B=\emptyset\end{array}} \arrow[r, "x_2"] & \boxed{\begin{array}{l}R=\emptyset\\B=\emptyset\end{array}} \arrow[r, "x_3"] & \boxed{\begin{array}{l}R=\{\mathsf{XWin}\}\\B=\emptyset\end{array}} \arrow[r, "\mathsf{XWin}"] & \boxed{\begin{array}{l}R=\emptyset\\B=\emptyset\end{array}}
\end{tikzcd}$$

B-Thread 2 (`DetectWinByO`) :
48 instances with priorities 2.001, . . . ,2.048

Scope :
an instance for each ordered triplet $\langle o_1, o_2, o_3\rangle$ of O events that fill a horizontal, vertical, or diagonal line.

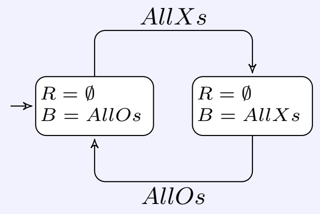
$$\begin{tikzcd}
{} \arrow[r] & \boxed{\begin{array}{l}R=\emptyset\\B=\emptyset\end{array}} \arrow[r, "o_1"] & \boxed{\begin{array}{l}R=\emptyset\\B=\emptyset\end{array}} \arrow[r, "o_2"] & \boxed{\begin{array}{l}R=\emptyset\\B=\emptyset\end{array}} \arrow[r, "o_3"] & \boxed{\begin{array}{l}R=\{\mathsf{OWin}\}\\B=\emptyset\end{array}} \arrow[r, "\mathsf{OWin}"] & \boxed{\begin{array}{l}R=\emptyset\\B=\emptyset\end{array}}
\end{tikzcd}$$

These two b-threads nicely illustrate a central feature of the approach:

Feature 1 (*behavioral autonomy*)**.** Behavioral programming makes it possible to encapsulate the implementation of each behavior requirement in an autonomous software object. For example, the above b-threads deal with winning conditions only, and are not concerned with issues irrelevant to that purpose (such as the enforcement of alternating turns, or the choice of an opening move).

We now describe a b-thread that forces players to alternate turns, by preventing a player from making two consecutive moves without an intervening move by the other player. This is achieved in B-Thread 3 by alternately blocking all O or all X events.

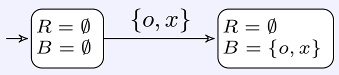
B-Thread 3 (`EnforcePlayersTurns`) : one instance with priority 3.0



Next, B-Thread 4 prevents a given square from being marked twice. Each of the nine b-threads defined by this template b-thread waits for either $\mathsf{X}_{\langle row,col\rangle}$ or $\mathsf{O}_{\langle row,col\rangle}$, for a particular square $\langle row, col\rangle$ and, when one of these is observed, the b-thread blocks them both forever.

B-Thread 4 (`DisallowSquareReuse`) :
9 instance with priorities 4.001,. . . ,4.009

Scope :
instance for each pair of events o and x in the same square

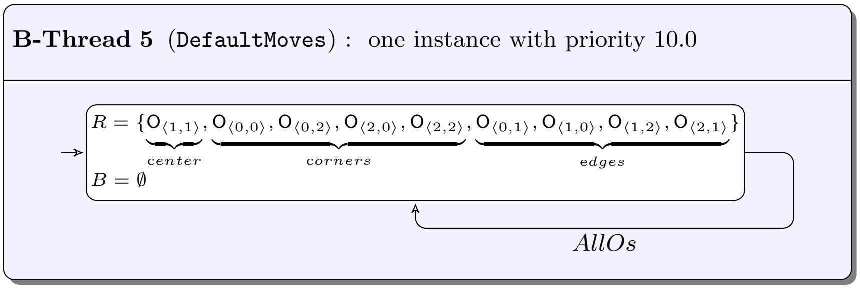


B-Threads 3 and 4 demonstrate a fundamental feature of behavioral programming:

Feature 2 (*positive and negative incrementality*)**.** In addition to adding new behaviors, and thus helping build up the desired dynamics of the system, a b-thread can *constrain* the dynamics by forbidding unwanted behaviors. This ability allows programmers to"sculpt" the system under construction, adding, removing or suppressing behaviors, with little or no need to modify the code of existing b-threads.

Programming the Strategy. We now complete the construction of a game-playing program by adding b-threads that capture the strategy of making moves.

As the reader will notice, the program's playing improves as b-threads are added[1]. First come B-Thread 5, that requests default moves.


B-Thread 5 (`DefaultMoves`) : one instance with priority 10.0
$R = \{\underbrace{\mathsf{O}_{\langle 1,1\rangle}}_{center}, \underbrace{\mathsf{O}_{\langle 0,0\rangle}, \mathsf{O}_{\langle 0,2\rangle}, \mathsf{O}_{\langle 2,0\rangle}, \mathsf{O}_{\langle 2,2\rangle}}_{corners}, \underbrace{\mathsf{O}_{\langle 0,1\rangle}, \mathsf{O}_{\langle 1,0\rangle}, \mathsf{O}_{\langle 1,2\rangle}, \mathsf{O}_{\langle 2,1\rangle}}_{edges}\}$
$B = \emptyset$
AllOs

This one-state b-thread simply requests the marking of all squares. The order of the requested events in the set R determines their priorities in our strategy: mark the center square first, then the corners, and only then the edge squares. Requested moves will be triggered only when not blocked and when there are no higher priority (unblocked) requests. The b-thread progresses regardless of whether the event was triggered due to its own request or due to some other b-thread's request (relying on the semantics that events are triggered once, regardless of the number of requesting b-threads; see more about the concept of unification in Section 5 and in LSCs [16]).

With the addition of this b-thread, the program can now play legally and can complete any game (though it will likely lose) or, more generally:

Feature 3 (*early/partial execution*)**.** Initial or partial versions of behavioral programs can be executed and can often display meaningful system behavior. This allows users and developers to observe the system's behavior at any stage of development, even the earliest ones, and take necessary actions, such as confirming that requirements are met, adjusting requirements, correcting errors, etc.

Note that B-Thread 5 encodes a rule-based strategy appropriate also for human players, as opposed to strategies directed only at computer implementations, such as minimax. This brings us to the next general feature:

Feature 4 (*alignment with human thinking*)**.** The autonomy of b-thread specifications enables the coding of "focused" program logic that specifies only and all relevant events (regardless of other requirements or of redundancy with other b-threads) which aligns well with how humans often think about behaviors.

We now proceed to insert richer strategy items, at higher priorities. For an immediate win, B-Thread 6 watches for two O's in one line and requests the marking of the third square, and to avoid an immediate loss, B-Thread 7 watches for two X's in one line and requests the marking of the third square.

[1] For clarity and brevity of the presentation we did not include a beginner-level b-thread that plays only random moves. We could have coded this first, and then have it overridden by the higher priority b-threads.

B-Thread 6 (`CompleteLineWithTwoOs`) :
48 instances with priorities 6.001, . . . ,6.048

Scope :
an instance for each ordered triplet $\langle o_1, o_2, o_3 \rangle$ of O events that fill a horizontal, vertical, or diagonal line.

$$\begin{tikzcd} {} \arrow[r] & \boxed{\begin{array}{l} R=\emptyset \\ B=\emptyset \end{array}} \arrow[r, "o_1"] & \boxed{\begin{array}{l} R=\emptyset \\ B=\emptyset \end{array}} \arrow[r, "o_2"] & \boxed{\begin{array}{l} R=\{o_3\} \\ B=\emptyset \end{array}} \arrow[r, "o_3"] & \boxed{\begin{array}{l} R=\emptyset \\ B=\emptyset \end{array}} \end{tikzcd}$$

B-Thread 7 (`PreventCompletionOfLineWithTwoXs`) :
48 instances with priorities 7.001, . . . ,7.048

Scope :
an instance for each ordered pair of X events, $\langle x_1, x_2 \rangle$, and O event, o_3, such that that $\{x_1, x_2, o_3\}$ fill a horizontal, vertical, or diagonal line.

$$\begin{tikzcd} {} \arrow[r] & \boxed{\begin{array}{l} R=\emptyset \\ B=\emptyset \end{array}} \arrow[r, "x_1"] & \boxed{\begin{array}{l} R=\emptyset \\ B=\emptyset \end{array}} \arrow[r, "x_2"] & \boxed{\begin{array}{l} R=\{o_3\} \\ B=\emptyset \end{array}} \arrow[r, "o_3"] & \boxed{\begin{array}{l} R=\emptyset \\ B=\emptyset \end{array}} \end{tikzcd}$$

Feature 5 (*explicit priorities*)**.** Note that it is essential that B-Threads 6 and 7 are of higher priority than those of the default moves, and that the winning move is of higher priority than the defensive one. Specifying b-threads' priorities explicitly and externally allows for encapsulation and control of this dependency between b-threads, and enhances the incrementality feature.

Another b-thread that illustrates the importance of priority is the B-Thread 8 (see code in [18]) that detects draw conditions. It waits for nine moves, and declares a draw, relying on having a lower priority than the b-threads that detect wins by X or O.

The Tic-Tac-Toe program includes two additional b-threads (see code in [18]), numbered 9 (`InterceptSingleFork`) and 10 (`InterceptDoubleFork`), that defend against situations where a future marking by player X will present him/her with the choice of winning in one of two different lines. Each of these b-threads watches all moves of both players, and uses ordinary Java code to keep track of the board configuration and see if a risky situation emerges. When it does, the b-thread requests an O move that prevents undesired opponent moves.

Lastly three more b-threads (`PushButton`, `MarkBoard` and `DeclareWinner`) handle various aspects of the user interface.

The b-threads described above, that use rich Java code in addition to BPJ idioms, demonstrate another central feature of behavioral programming:

Feature 6 (*application integration*)**.** Ordinary applications, simple or complex, can be refactored so that they become endowed with some capabilities of behavioral programming. This is done by moving application code into a `runBThread` method, and then coordinating with other b-threads using the event sets and `bSync`. Conversely, behavioral programs can benefit from the full power of the underlying language and external services, such as communications, database access, and user interface, by adding the code that uses these services to the existing b-thread code.

We conclude the list of features and the exposition of BPJ with two more capabilities that emerge from the operational semantics of behavioral programming:

Feature 7 (*consolidation/unification of identical execution requests*)**.** In BPJ, when an event is triggered, all b-threads that requested it it are notified (in addition to all b-threads explicitly waiting for it). All these b-threads progress, and are "satisfied", as a result of this single occurrence, unconcerned with the identity of the b-thread that requested the specific event. For example, in our Tic-Tac-Toe program, a low priority request to mark an edge square, that was placed at the beginning of the game, may be fulfilled only as a result of a higher priority move. However the low priority b-thread will proceed happily as a result, oblivious of the delay and other game circumstances. The power of this consolidation/unification is also highlighted in the NY-to-LA trip example, as driving along the route may be consolidated with driving according to schedule, without "duplication of effort", without negative performance effects, and without creating inter-b-thread dependencies in the code.

Feature 8 (*exploitation of multi-core architectures*)**.** The implementation of b-threads as independent Java threads leverages the multiple processors now commonly available in multi-core architectures. Given that b-threads can include arbitrary Java code, they can be used for coordinating the parallelization of independent behaviors. This parallel use of multiple processors can also help reduce the synchronization delays caused by waiting for b-threads to reach their next state.

4.3 Example 3: Patching a Reactive System

To further illustrate how a system's behavior can be incrementally composed of independently programmed b-threads, the following example, a simple maze-solving robot, focuses on application patching. As shown in Figure 7, the maze is a set of square rooms; adjacent rooms may or may not be separated by a wall.

The robot (depicted here as Disney's aptly named WALL-E$^{(TM)}$) is capable of (only) two operations, represented by events: `forward` for moving forward one room, and `turn` for turning a quarter circle (90 degrees) clockwise in place. Note that to turn a quarter circle to the left, the robot needs to turn clockwise three times. The existence of a wall is manifested by blocking the `forward` event, as explained below. In our example, the robot is oblivious of the concept of a maze and is simply programmed to follow the wall on its left according to the well known "left-wall-rule". The robot's program is comprised of three b-threads, with a strict priority order: (1) keeping to the left wall by an infinite loop that waits for a `forward` event and then requests a `turn` event three times; (2) always trying to advance, by requesting a `forward` event in an infinite loop, and (3) in a lower priority b-thread (that can trigger events only when the previous two can't), requesting a 90 degree right turn (`turn` event) in an infinite loop.

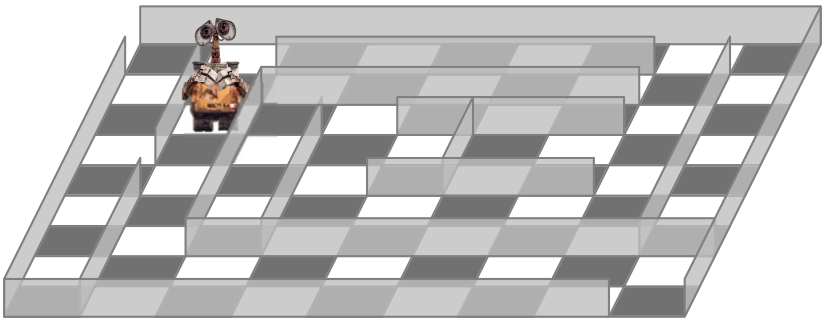


Fig. 7. The "wall follower" robot in a maze. When it enters a room (a square) it scans the exits and takes the first open way starting from the room on its left, clockwise.

In a final implementation the robot would have a sensor that detects whether there is a wall ahead and a b-thread that blocks the `forward` event when a wall is detected. However, for testing and debugging, a fourth b-thread, representing the environment, replaces the sensor. The environment b-thread maintains the maze's map and waits for all `forward` and `turn` events to update the robot's current location and orientation (e.g. Room $\langle 1, 2 \rangle, South$). It then blocks the `forward` event if the robot is facing a wall.

A "bug" is then introduced, by specifying a new requirement: unevenness in robot's wheels sometimes causes it to turn 90 degrees clockwise when it completes a forward move. The robot's gyro detects the situation and announces it as an `drift` event. The environment b-thread is modified accordingly to both randomly request this event and to update the robot's orientation.

The correction for the "bug" can then be applied using a fifth b-thread at top priority, which waits for `drift` event and then requests six `turn` events immediately after the `drift`: three of which coincide with the already-planned left turn (see discussion of unification of events in Feature 7), and the additional three reposition the robot to the correct orientation.

The incrementality feature of behavioral programming is thus shown to open up the possibility of patching "completed" applications, by programming the desired new behavior in a way that targets it directly to the applicable situation, without modifying existing b-threads.

5 The BPJ Library

In this section we provide more details about the components of the BPJ Java library. As mentioned above the library and application examples are available online at [18].

5.1 Classes

Events. Events are represented in BPJ as objects of the class `Event`. This base class does not carry significant data or capabilities. Subclasses of `Event` can be defined, in order to distinguish between various types of events and to associate additional information and behavior with event objects.

Behavior Threads. Behavior Threads are specified using the `BThread` class, each instance of which is associated with a Java execution thread whose progress embodies a run of the behavior thread.

As mentione above, each b-thread object has three sets of events: *requested events* and *blocked events* correspond to the sets R and B in definitions 1 and 2 above, and *watched events* corresponds to the set of events appearing in the labels of the arrows in the transition diagram.
The `BThread` class has three main methods:

1. `runBThread` is an abstract method that implements the logic of the b-thread, and is overridden when designing concrete b-threads. It runs in a dedicated Java execution thread. The parameters of the method, if any, are application specific.
2. `bSync` is a library method. It updates the sets of requested, watched and blocked events, and calls a method `bSync` of the `BProgram` object to synchronize the b-thread with all other b-threads and invoke the event triggering mechanism.
3. `startBThread` is a library method that can be called by any part of the code (inside or outside of a b-thread), to start a new b-thread and activate its `runBThread` method in a dedicated Java execution thread. The parameters to this method are passed without change to the `runBThread` method.

Behavioral Programs. In BPJ, a set of b-threads is referred to as a *behavioral program.* It is implemented in the class `BProgram`, and its semantics is per definition 2 and the algorithm in Subsection 3.2. This class has one static object that holds all the participating b-threads. The class includes the library method `bSync` that synchronizes each calling b-thread with all other b-threads and invokes the control mechanism that chooses and triggers the next event and notifies the b-threads waiting for that event. The method `bSync` of the class `BProgram` is only called internally, by the `bSync` method of the class b-thread.

Sets of Events. In BPJ, sets of events are represented in two basic ways. Either as an enumerable collection of objects of type `Event`(using by default the Java Collection Framework) or by creating a class that implements the BPJ interface `EventFilterInterface` and uses the `contains` method for specifying membership of events in the set (e.g., by examining the values of object variables).

The set of requested events must be represented as an enumerable collection of concrete events, since events to be triggered are selected from this set in order of priority. By contrast, watched and blocked events can be represented either as an enumerable set or through the `EventFilterInterface`, by rule-based definitions. The rule-based representation of sets allows for compact specification of very large, possibly infinite, sets, and enables watching for, and handling, events for which not all information is known in advance.

The BPJ library includes generic filters, such as `EventFilterForClass`, that facilitates specification of the set of all events of a certain class (using `instanceof` to implement the `contains` method), the constant filter `none` that represents the empty set (used for example when a b-thread is not blocking any event), and the constant filter `all` that returns `true` for all events (used for example when a b-thread wishes to watch all events in the system).

5.2 Interrupting Events

In addition to the above event sets, each b-thread object has a set of events called the *interrupting events* set (`interruptingEvents`). When a triggered event is contained in the interrupting events set of a b-thread an exception is raised in the b-thread and the latter typically terminates. This simplifies handling termination conditions that are common to many states. For example, in the Java code for Tic-Tac-Toe, almost all b-threads specify the `gameOver` event as an interrupting event.

5.3 Incorporating Data and Algorithms

It is important to note that the BPJ library allows b-threads as ordinary Java programs, to also include conventional data structures and algorithms for arbitrary processing, as shown in the examples.

5.4 Conflict and Deadlock Resolution

Since incrementality of development and independence and autonomy of b-threads are a central theme of BPJ, helping the developer avoid unplanned conflicts and deadlocks is of particular interest. In addition to applying methodologies that are applicable to ordinary models and programs, we propose to attack this in several ways:

– *Model checking in Java:* Since the BPJ implementation uses pure Java infrastructure, it seems natural to apply the model checking capabilities of Java Pathfinder [35] directly to the behavioral program to detect deadlocks.
– *"Smart" execution:* It is interesting to explore the possibility of relaxing some of the strict prioritization requirements, where the developer feels that these don't matter, or hard to determine. We would then introduce *smart play-out* techniques, similar to those used for LSCs (which were based on model-checking and on AI planning algorithms [14,19]) to determine event triggering order. Such predictive mechanisms will be set up to choose between events of equal priority in a way that avoids future conflicts, or satisfies other desired results.

– *Dynamic recovery:* The existence of a central event triggering mechanism allows for detection of potential deadlock situations at run time. In these cases, subject to programmed rules, either error messages can be issued, certain b-threads can be terminated, or events may be triggered, to initiate recovery. Such rules can use data about relationships between the b-threads and events, and error messages can be quite informative about causes and potential remedies of the deadlock. Finally, note that if conflicts and deadlocks are discovered in an application, they may sometimes be resolved not by changing existing b-threads, but also externally, by changing b-thread priorities or by adding new b-threads.

5.5 Performance

In the current initial implementation, the execution rate can reach thousands of events per second, with thousands of participating b-threads. Performance of behavioral programs depends among others on the number of b-threads, relationships between requested and blocked events, the complexity of filter functions, and the non-BPJ-related processing performed by b-threads. One of the approaches for controlling and optimizing performance is to organize b-threads in synchronization groups to reduce dependencies and delays associated with synchronization.

6 Related Work

6.1 Scenario-Based Languages

As mentioned in the introduction, several languages exist for specifying scenarios, including message sequence charts (MSCs) [22] and UML sequence diagrams [32], which are used in system development to model interactions between objects or processes. With the advent of live sequence charts (LSCs) [7] liveness and safety were added to such languages, through multi-modality; e.g., distinguishing universal from existential behavior and what must happen from what may happen, in the process also providing for concise specification of forbidden behavior.

In the work culminating in [16], a GUI-based method for capturing LSCs (*play-in*) and a method for executing an LSC specification (*play-out*) were developed. Both are implemented in the Play-Engine tool. As mentioned above, the operational, interpreter-style semantics of play-out is especially relevant to our work here. In fact, its essence, especially the coordination process by which the multi-modality is translated into run-time decisions, substantially influenced the choice of idioms proposed in this paper. A more detailed mapping between LSCs and Java programs using BPJ is out of the scope of this paper.

A few years after the Play-Engine's development, the S2A compiler was constructed [27,13], offering an alternative way to execute LSC specifications. The compiler converts the scenarios specified in (a subset of) the LSC language into Büchi automata that represent all possible transitions between cuts of the original scenarios. The automata and a coordinator function that implements the play-out logic, are then encoded in Java, and are interwoven with a base program using aspects. While this approach allows for integration with a conventional Java application, the Java code that represents the LSC scenarios is not designed to be used by the programmer for further refining the scenario itself.

6.2 Programming Scenarios in Other Languages

There are many specialized languages that support programming concurrent scenarios for a variety of application areas, such as business workflow engines (e.g., BPEL [31]), Expert Systems and Decision Support System (e.g. [10]) simulators (e.g., MATLAB Simulink [34]), and general automation and robot control (e.g., LabVIEW [26]). These environments have rich interfaces for specifying sequences of actions, and rules and priorities for triggering and parallel execution of events and scenarios.

The main difference from those languages is that, to the best of our knowledge, the underlying interfaces therein do not include explicit event blocking idioms, similar to those of BPJ. Of course, as was in done BPJ using Java, it is possible to implement scenario-based programming or behavioral programming in any rich enough language. All that is needed is a sufficient expressive power to implement Definitions 1 and 2 above. It may be worthwhile to explore the applicability of the BPJ library and equivalent request-and-block idioms in such contexts.

Given that direct equivalent of request-and-block idioms were not identified, detailed comparison of the existing idioms in each of these systems to BPJ and the compactness in which specific tasks can be programmed is outside of the scope of this paper. Such a comparison may include, for example, comparing BPJ b-threads to rules in AI and expert systems and the differences between, say, rule-disabling idioms [23] in expert systems, which disable *entire rules* , and event-blocking in BPJ, which prevents the triggering of a requested event.

6.3 Interprocess-Communication and Parallel-Processing Idioms

The manner in which b-threads request, listen out for, and get notified of triggered events is a form of publish/subscribe protocol (see, e.g., [8]). In this context, one can view BPJ as an enhanced publish/subscribe engine that allows participants (b-threads) to block messages (events), which is not possible in common publish/subscribe, and where sending of messages is deferred until all relevant participants are ready, i.e. completed processing the previous message. In addition, the BPJ mechanism enables features such as the unification/consolidation of identical requests into a single execution, described above.

The idioms we propose are meant for use in coding b-threads as descriptions of system behavior, and are not targeted for interprocess communication and concurrency per-se. However, as our work does propose primitives for communication between b-threads, it would be of interest to carry out a more thorough comparison with some of the well-known approaches to specifying and analyzing concurrency, such as *communicating sequential processes* (CSP) [20], the *calculus of communicating systems* (CCS) [28], the π-*calculus* [29], the programming languages Erlang [1], Esterel [3], Lustre [11], Signal [21], Orc [25], UNITY [30], and many more. As far as our current familiarity with these approaches goes, we have not found constructs in any of these that are in sufficiently close correspondence with our idioms. In any case, if found desirable, we do not think there would be significant obstacles to implementing the request-and-block idioms in these languages.

6.4 Object-Oriented and Other Programming Approaches

The BPJ approach is inclusive; i.e., the programmer can develop parts of the application in classical OO, procedural, or other approaches, and other parts with b-threads. In addition, the b-thread components can still contain non-BPJ code.

To stress the differences between the BPJ approach and conventional OO programming, we discuss how the Tic-Tac-Toe and the robot examples can be expected to be programmed without BPJ.

A classical OO Tic-Tac-Toe program would likely have a main loop controlling the players' turns, updating the GUI and maintaining the data structures for the current game-board configuration. Within such a loop, either a mathematical technique such as the minimax algorithm would be used to analyze possible moves, or, various heuristics or a decision tree would be applied to decide on the next move. In the former - there is little or no similarity between the "classical" program and our behavioral one. In the latter one may view the various decisions, considered in order, as a form of prioritized scenarios or b-threads. In either case, the modularity and structural uniformity of the behavioral approach seem to stand out in such a comparison. In the BPJ approach, each b-thread can readily exist as a separate file and requires little or no explicit dependency on other b-threads, or even on the existence of a game board. By contrast, in the OO approach, the different heuristics (or the related method calls) are likely to be coded together as an integral part of the application, and they are all likely to use a shared data structure for the game board. Note that in BPJ there are no method calls between b-threads, and dependencies between the b-threads in the example are only via events.

The wall-follower robot example exhibits similar modularity and incrementality, and emphasizes the point in relying on standard BPJ interfaces to mesh the robot and the environment into a single application. Robotic systems often require a significant degree of concurrency, implementing multiple parallel activities and behaviors. To that end, the software typically waits for external events, handles interrupts, coordinates concurrent processes, and sends commands to actuators. The example suggests that a general-purpose language with the built-in synchronization and event triggering mechanism of BPJ may be useful for robot programming.

Lastly, specification of b-thread priority may be seen as analogous to choosing the right join points in aspect oriented programming, or deciding the order of lines of source code in a classical program, yet it appears to offer better combination of encapsulation and flexibility.

7 Conclusion and Future Directions

We have described the BPJ library, which facilitates scenario-based programming, or behavioral programming, in Java, using b-threads and the request-and-block idioms. The motivation for development of this library is twofold. On one hand, it enriches conventional object-oriented programs with the capabilities of b-threads, and on the other hand, it enriches scenario programming with the power of a general-purpose language. We believe that this integration of objects and behavior threads and implementation of modal specifications in imperative code, expands the range of applicability of behavioral programming, and can serve as a design pattern in many languages towards the goal of liberating programming, as envisioned in [12].

The advantages of behavioral programming include incremental development, with minimal changes to existing b-threads, encapsulation and autonomy of behavior with minimal dependency between b-threads, the ability to observe meaningful behavior from the earliest stages of development, explicit external control over behavior priority, alignment of b-threads with how people typically think about system behavior, more direct encoding of requirements and use-cases, and exploitation of multi-core architectures.

Due in part to the autonomy feature, behavioral programs can more readily "explain" their decisions (and behavior). Events that caused transitions in a recently executed chain of b-threads, can provide important insight into the rationale for the program's progress, something that may be harder to infer from a usual trace. This may be useful in developing and debugging behavioral applications, in using b-threads for monitoring, and in developing systems capable of learning. An additional application of the autonomy feature can be post-deployment system customization, where an end-user can modify the system behavior by adding b-threads.

We consider the work described here as an initial infrastructure — a sort of "assembly language" for programming behaviors and scenarios with b-threads. Indeed, in some cases, relying only on the basic idioms of this paper may create verbose and cumbersome code. Hence, future directions for research include devising higher level (possibly graphical) macro-like representations for useful recurring patterns such as suspension and resumption of continuous activities like the driving activity in the example in Section 2.

It may also be valuable to embed variants of our idioms in other languages, such as general object-oriented, special purpose, or functional and logic programming languages. As an example, for the functional language Erlang [1], it would be of interest to explore if scenario-based programming or behavioral programming can become a design pattern that leverages Erlang's ease and efficiency of using concurrent processes for independent activities.

As to the actual work on BPJ, we envision additional research and development on IDE support, formal verification and automated (partial) synthesis. In addition, translation from LSCs or UML sequence diagrams to Java programs using BPJ seems practical, and such compilers may further simplify the transition from requirement specifications to design and development.

Acknowledgements. We would like to thank Amir Kantor for suggesting the analysis of the proposed idioms through transition systems, Michal Gordon for many valuable comments, Shahar Maoz for reviewing and commenting on early drafts of this paper, Mira Balaban for helping clarify the paper's focus on programming, Uri Avraham for insightful observations on the composition of scenarios, Guy Weiner for suggesting to document the relationship between the transition system and the Java program more explicitly.

The research of all three authors was supported in part 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 to DH from the European Research Council (ERC) under the European Community's FP7 Programme. In addition, GW's research was supported in part by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in ERLANG. PrenticeHall, Englewood Cliffs (1993)
2. Atir, Y., Harel, D.: Using LSCs for scenario authoring in tactical simulators. In: SCSC, pp. 437–442 (2007)
3. Berry, G.: The foundations of Esterel. In: Proof, Language, and Interaction, pp. 425–454 (2000)
4. Bunker, A., Gopalakrishnan, G., Slind, K.: Live sequence charts applied to hardware requirements specification and verification. STTT 7(4), 341–350 (2005)
5. Combes, P., Harel, D., Kugler, H.: Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. Software and System Modeling 7(2), 157–175 (2008)
6. Crowley, K., Siegler, R.S.: Flexible Strategy Use in Young Children's Tic-Tac-Toe. Cognitive Science 17(4), 531–561 (1993)
7. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. J. on Formal Methods in System Design 19(1), 45–80 (2001)
8. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The Many Faces of Publish/Subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
9. Fisher, J., Harel, D., Hubbard, E.J.A., Piterman, N., Stern, M.J., Swerdlin, N.: Combining state-based and scenario-based approaches in modeling biological systems. In: Danos, V., Schachter, V. (eds.) CMSB 2004. LNCS (LNBI), vol. 3082, pp. 236–241. Springer, Heidelberg (2005)
10. Giarratano, J., Riley, G.: Expert systems: principles and programming. Brooks/-Cole Publishing Co., Pacific Grove (1989)
11. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The Synchronous Data-Flow Programming Language LUSTRE. Proc. IEEE 79(9), 1305–1320 (1991)
12. Harel, D.: Can Programming Be Liberated, Period? IEEE Computer 41(1), 28–37 (2008)
13. Harel, D., Kleinbort, A., Maoz, S.: S2A: A compiler for multi-modal UML sequence diagrams. In: FSE, pp. 121–124 (2007)
14. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-out of Behavioral Requirements. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002)

15. Harel, D., Kugler, H., Weiss, G.: Some Methodological Observations Resulting from Experience Using LSCs and the Play-In/Play-Out Approach. In: Scenarios: Models, Transformations and Tools, pp. 26–42 (2003)
16. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003)
17. Harel, D., Marron, A., Weiss, G.: Behavioral Programming (in preparation)
18. Harel, D., Marron, A., Weiss, G.: The BPJ Library, `http://www.cs.bgu.ac.il/~geraw`
19. Harel, D., Segall, I.: Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 485–499. Springer, Heidelberg (2007)
20. Hoare, C.A.R.: Communicating Sequential Processes. CACM 21(8), 666–677 (1978)
21. Houssais, B.: The synchronous prog. language SIGNAL, a tutorial. In: IRISA (2002)
22. ITU. International Telecommunication Union Recommendation Z.120: Message Sequence Charts (1996)
23. Jagadish, H.V., Mendelzon, A.O., Mumick, I.S.: Managing Conflicts Between Rules. J. Comput. Syst. Sci. 58(1), 13–28 (1999)
24. Keller, R.: Formal verification of parallel programs. CACM 19(7), 371–384 (1976)
25. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc Programming Language. In: FMOODS/FORTE, pp. 1–25 (2009)
26. LabVIEW. Getting Started with LabVIEW (June 2009)
27. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: FSE, pp. 219–230 (2006)
28. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
29. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes. Inf. Comput. 100(1), 1–40 (1992)
30. Misra, J.: A foundation of parallel programming. In: International Summer School on Constructive Methods in Computer Science, pp. 397–433 (1988)
31. OASIS. Web Services Business Process Execution Language V2.0 (May 2007)
32. OMG. Unified Modeling Language Superstructure Specification, v2.0 (August 2005)
33. Sadot, A., Fisher, J., Barak, D., Admanit, Y., Stern, M.J., Hubbard, E.J.A., Harel, D.: Toward Verified Biological Models. IEEE/ACM Trans. Comput. Biology Bioinform. 5(2), 223–234 (2008)
34. TheMathWorks. The Simulink 7 Reference (2009)
35. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering 10, 203–232 (2003)