A decentralized approach for programming interactive applications with JavaScript and blockly

Conference Paper · October 2012

DOI: 10.1145/2414639.2414648

3 authors:

Assaf Marron
Weizmann Institute of Science
65 PUBLICATIONS **748** CITATIONS

SEE PROFILE

Gera Weiss
Ben-Gurion University of the Negev
77 PUBLICATIONS **1,019** CITATIONS

SEE PROFILE

Guy Wiener
HP Inc.
9 PUBLICATIONS **95** CITATIONS

SEE PROFILE

A Decentralized Approach for Programming Interactive Applications with JavaScript and Blockly

Assaf Marron
Weizmann Institute of Science
assaf.marron@weizmann.ac.il

Gera Weiss
Ben-Gurion University
geraw@cs.bgu.ac.il

Guy Wiener
HP Labs
guy.wiener@hp.com

Abstract

We present a decentralized-control methodology and a tool-set for developing interactive user interfaces. We focus on the common case of developing the client side of Web applications. Our approach is to combine visual programming using Google Blockly with a single-threaded implementation of behavioral programming in JavaScript. We show how the behavioral programming principles can be implemented with minimal programming resources, i.e., with a single-threaded environment using coroutines. We give initial, yet full, examples of how behavioral programming is instrumental in addressing common issues in this application domain, e.g., that it facilitates separation of graphical representation from logic and handling of complex inter-object scenarios. The implementation in JavaScript and Blockly (separately and together) expands the availability of behavioral programming capabilities, previously implemented in LSC, Java, Erlang and C++, to audiences with different skill-sets and design approaches.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.1.3 [*Programming Techniques*]: Concurrent programming

General Terms Languages, Design, Human Factors

Keywords Behavioral Programming, JavaScript, Coroutines, HTML 5, Google Blockly, Visual Programming, Client-side, Web application, Browser

1. Introduction

The behavioral programming (BP) approach is an extension and generalization of scenario-based programming, which was introduced in [4, 11] and extended in [17]. In behavioral programming, individual requirements are programmed in a decentralized manner as independent modules which are interwoven at run time. Advantages of the approach include facilitation of incremental development [17], naturalness [10], and facilitation of early detection of conflicting requirements [18]. A review of research and tool development in behavioral programming to date appears in [14]. While behavioral programming mechanisms are available in several languages such as *live sequence charts* (LSC), Java, Erlang and C++, its usage for complex real-world application and development of relevant methodologies are only beginning.

The purpose of the research summarized in this paper was to examine behavioral programming in a specific application domain, and to adjust it to languages, technologies and work methods that are used in this domain. The paper also sheds light on the principles of programming behaviorally and the facilities required of behavioral-programming infrastructure.

The paper describes and demonstrates the implementation of behavioral programming in JavaScript and in Google's Blockly for the client side of Web applications, and then discusses general usability and design principles highlighted by these implementations. Clearly, in addition to the combined Blockly-and-JavaScript implementation shown here, behavioral programming can be used with JavaScript without

Blockly, or with Blockly with translation to another language, such as Dart. In this regard, Blockly is a layer above our JavaScript implementation, which can simplify development and facilitate experimenting with a variety of programming idioms. We hope that together with previously described ideas about scenario-based and behavioral programming, this paper will help drive the incorporation of these principles into a wide variety of new and existing environments for development with agents, actors, and decentralized control, and will help add them into the basic set of programming concepts that are understandable by and useful for novice and expert programmers alike.

We propose that decentralized scenario oriented programming techniques offer significant advantages over traditional programming metaphors in this specific domain. Consider, for example, an application with some buttons on a screen where there is a requirement that the software reacts to a sequence of button clicks in a certain way. Using a non-behavioral style with, e.g., JavaScript, one of the standard programming languages in this domain, the programmer would handle each button-click separately, and introduce special code to manage state for recognizing the specific sequence of events. We argue that with behavioral programming such a requirement can be coded in a single imperative script with state management being implicit and natural rather than explicit. See Section 5.1.

Our choice of the domain of interactive technology is influenced also by the current debate about the relative merits of Flash and JavaScript/HTML5 technologies (see, e.g., [28]). We believe that the technological discussion conducted mainly in industry should be accompanied with academic revisiting and analysis of software engineering and methodology aspects.

About the terms *block* and *blocking*

As we are dealing with languages and programming idioms, the reader should note that the term *block* appears in this paper in two different meanings: (a) a brick or a box - referring to programming command drawn as a two-dimensional shape; and (b) a verb meaning *to forbid* or *to prevent*, associated with the behavioral programming idiom for declaring events that must not happen at a given point in time. It is interesting to observe that these meanings are individually commonly used and are appropriate for the intent, that finding alternative terms for the sole purpose of disambiguation, is unnecessary, in the least, and in some cases, artificial and even detrimental to the understandability of the text. In this context, of course, the language name Blockly fits nicely with its proposed use in programming behaviorally. Still to minimize confusion, we avoided using the terms *block* and *blocking* in two other common software-related meanings, namely, (c) stopping a process or a subroutine while waiting for an event or resource; and, (d) a segment of program code which contains all the commands between some end-markers such as curly braces or `begin` and `end`.

2. Behavioral Programming

In this section we outline the technique of Behavioral Programming for the development of reactive systems. Formal definitions and comparisons with other programming techniques appear in [14, 17, 19].

A preliminary assumption in our implementation of behavioral programming is that an application, or a system, is focused on processing streams of events with the goal of identifying and reacting to occurrences of meaningful scenarios. Detected event sequences are then used to trigger abstract, higher level, events, which in turn may trigger other events. Some of these events are translated to actions that the system takes to effect changes in the external world. This cycle results with a reactive system that translates inputs coming from its sensors to actions performed by its actuators.

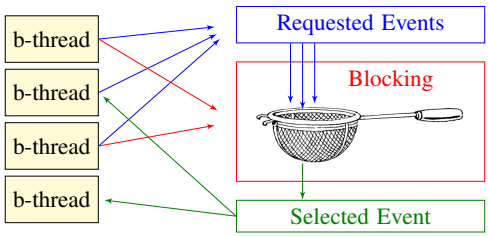


Figure 1. Behavioral programming execution cycle: all b-threads synchronize, declaring requested and blocked events; a requested event that is not blocked is selected and b-threads waiting for it are resumed.

More specifically, in a behavioral program, event sequences are detected and generated by independent threads of behavior that are interwoven at run time in an enhanced publish/subscribe protocol. Each *behavior thread* (abbr. *b-thread*) specifies events which, from its own point of view must, may, or must not occur. As shown in Figure 1, the infrastructure consults all b-threads by interweaving and synchronizing them, and

selects the events that will constitute integrated system behavior without requiring direct communication between b-threads. Specifically, all b-threads declare events that should be considered for triggering (called *requested events*) and events whose triggering they forbid (*block*), and then synchronize. An event selection mechanism then triggers one event that is requested and not blocked, and resumes all b-threads that requested the event. B-threads can also declare events that they simply "listen-out for", and they too are resumed when these waited-for events occur.

This facilitates incremental non-intrusive development as outlined in the example of Figure 2. For another example, consider a game-playing program, where each game rule and each player strategy is added in a separate module that is oblivious of other rules and strategies. Detailed examples showing the power of incremental modularity in behavioral programming appear, e.g., in [17–19].

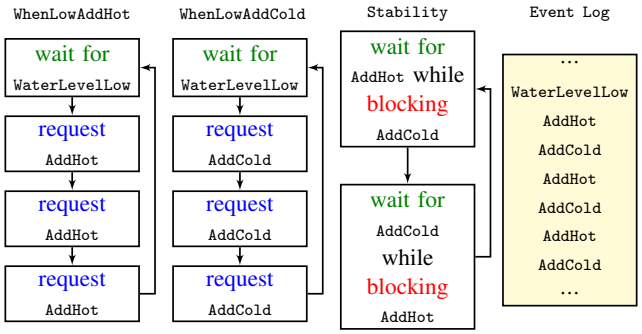


Figure 2. Incremental development of a system for controlling water level in a tank with hot and cold water sources. The b-thread `WhenLowAddHot` repeatedly waits for `WaterLevelLow` events and requests three times the event `AddHot`. `WhenLowAddCold` performs a similar action with the event `AddCold`, reflecting a separate requirement, which was introduced when adding three water quantities for every sensor reading proved to be insufficient. When `WhenLowAddHot` and `WhenLowAddCold` run simultaneously, with the first at a higher priority, the runs will include three consecutive `AddHot` events followed by three `AddCold` events. A new requirement is then introduced, to the effect that water temperature should be kept stable. We add the b-thread `Stability`, to interleave `AddHot` and `AddCold` events using the event-blocking idiom.

In behavioral programming, all one must do in order to start developing and experimenting with scenarios that will later constitute the final system, is to determine the common set of events that are relevant to these scenarios. While this still requires contemplation, it is often easier to identify these events than to determine objects and associated methods. By default, events are opaque and carry nothing but their name, but they can be extended with rich data and functionality. Further, the incremental traits of BP and the smallness of b-threads (see Section 5) facilitate subsequent adding and changing of event choices.

The behavioral programming technique facilitates new automated approaches for planning in execution [12, 15], program verification and synthesis [13, 18, 22], visualization and debugging of concurrent execution [5], natural language coding of scenarios [9], and program repair [20] .

3. Infrastructure Implementation

3.1 Coordinating behaviors written in JavaScript

In principle, the concepts of behavioral programming are language independent and indeed they have been implemented in a variety of languages using different techniques. However, certain language facilities are needed in order to control the execution, synchronization and resumption of the simultaneous behaviors. In LSC this is done by the control mechanism which, interpreter-like, advances each chart along its locations (see, e.g. [16]). In Java [17], Erlang [32] and C++ the mechanism is implemented as independent threads or processes and uses language constructs such as `wait` and `notify` for suspension and resumption. When executed in a browser, a JavaScript application is typically executed as a single thread in a single process, hence another mechanism is needed. Note that for the present proof-of-concept, the choice is indeed JavaScript, but the language-independent principles can be implemented also in other technologies, say, Flash ActionScript, if appropriate constructs are available for suspension and resumption.

In JavaScript 1.7 [27] (currently supported in the Firefox browser) the `yield` command was introduced which allows the implementation of generators and coroutines, and we chose to use it for our BP implementation - providing suspension and resumption in single-threaded multi-tasking. Briefly, the `yield` mechanism allows a method to return to its caller, and upon a subsequent call, continue from the instruction immediately after the most recent return. Thus, in the context of coroutines, coordinated behavioral execution can be described as follows:

Behavior threads are implemented as JavaScript coroutines. For example, the b-thread `Stability` of the water-tap example is shown in the following code-snippet. The b-thread two calls to `yield` correspond to the two boxes of this b-thread in Figure 2.

```
function() {
```

```
    while (true) {
        yield({
            request: [],
            wait: ["addHot"],
            block: ["addCold"]
        });
        yield({
            request: [],
            wait: ["addCold"],
            block: ["addHot"]
        });
    }
}
```

The infrastructure executes in cycles. In each cycle, the b-threads are called one at a time. The coroutine of each b-thread returns, using the `yield` command, specifying this coroutine's declaration of requested events, blocked events and waited-for events. Once each of the b-threads has been called and has returned in this manner, and its declarations have been recorded, the infrastructure selects for triggering an event that is requested and not blocked. The infrastructure then resumes all b-threads that requested or waited for the event, by calling them (and only them), one by one, as part of the new cycle.

In fact this description summarizes the majority of what was needed for our implementation of behavioral programming in JavaScript. More details appear in Appendix A. The b-threads are, of course, application specific and are provided by the application programmer.

Since JavaScript requires that the `yield` command be visible in the source of the coroutine at least once, in the present implementation we chose not to hide it within a method dedicated to behavioral synchronization and declarations, such as the `bSync` method in the Java implementation. Indeed, we feel that this diversity in command name usage across languages emphasizes that BP benefits, such as ease of state management and incrementality (see Section 5), can be gained by implementing and using BP principles in any language and with a variety of idioms.

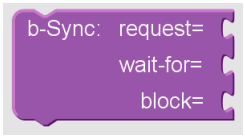
3.2 Behavioral blocks for Blockly

The Google Blockly environment [8] is built along principles similar to those of the popular Scratch language [30]. Other languages and environments in this family include, among others, BYOB/SNAP! [26], MIT App Inventor [1], and Waterbear [6]. In these languages, the programmer assembles scripts from a menu of block-shaped command templates that are dragged onto a canvas. The Blockly blocks contain placeholders for variables and sub-clauses of the commands and can express scope of program-segment containment, relying on the notation of a block containing physically other blocks, with possible nesting. The popularity of the Scratch language suggests that this style of coding is more accessible to children than standard programming languages, and perhaps even other visual languages, such as LSC. However, we also feel that the combination of visualization and language customization make Blockly an excellent platform for demonstrating coding techniques that would otherwise require pseudo-code or abstraction, and it may also provable suitable for complex applications.

While Scratch and BYOB are interpreted (in SmallTalk and now also in Flash), Blockly and Waterbear diagrams are translated into code (we use JavaScript) which can later be manipulated and executed natively without dependency on the development environment.

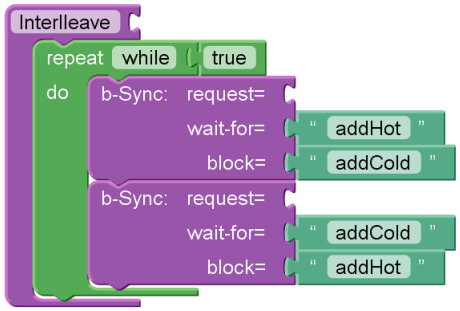
Our implementation of behavioral programming in Blockly includes new (types of) blocks: the `b-thread` block

for the b-thread logic (the string `b-thread` in the template can replaced by the programmer with the b-thread's name or description); the `b-Sync` block



for synchronization and bidding of requested, waited-for, and blocked events; and, a `lastEvent` block where b-threads that wait for a number of events can examine the one that indeed happened.

For illustration, the `Stability` b-thread of the water-tap example, is coded in Blockly as



and is automatically compiled into the JavaScript code shown in Section 3.1.

The advantages of programming in this manner are discussed in Section 5. We will only mention here that the visualness of Blockly adds to the usability of BP principles, while behavioral decomposition should simplify the development of complex applications in Blockly .

As the Google Blockly environment is in early development stages, we had to also add some basic capabilities, such as list concatenation, that are not specific to behavioral programming.

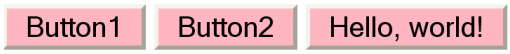
4. Programming an Interactive Application Behaviorally - an Example-driven Tour

In this section we present the underlying design principles for applications coded with Blockly, JavaScript, and HTML, via a review of several small applications. The code and additional explanations are available online at `www.b-prog.org`

4.1 Sensors and Actuators

A key design principle is separating the "real" world from the application logic using appropriate interfaces for translating occurrences in the environment into entities that can be processed by the application, and application decisions into effects on the environment.

We begin our example-driven tour with examination of the sensors and actuators of a simple application which consists of the following three buttons:



The requirements for this application are that when the user clicks Button1 the greeting should be changed to "Good Morning" and when the user clicks Button2 the greeting should be changed to "Good Evening". We first have to create a sensor for the button clicking:

```
<input
   value   = "Button1"
   type    = "button"
   onclick = "bp.event('button1Click');"
/>
```

The clicking is captured by the standard use of the HTML verb `onclick` and the ensuing activation of JavaScript code. The function `bp.event` is part of the behavioral infrastructure in JavaScript, and it creates the behavioral event passed as a parameter. Details about event triggering appear in Section 4.3

To transform application decisions into effects on the environment, an HTML entity can activate JavaScript code using another part of the behavioral infrastructure we added in Blockly/HTML/JavaScript, the verb `when_`*eventName*, as follows:

```
<input
    value    = "Hello, world!"
    type     = "button"
    when_button1Click  = "value='Good Morning'"
    when_button2Click  = "value='Good Evening'"
/>
```

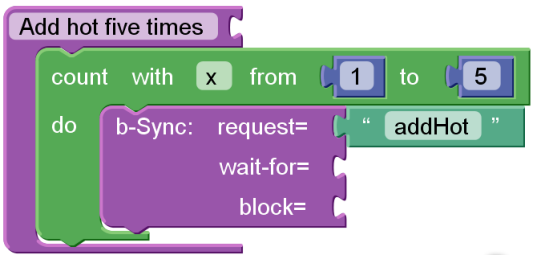
In this simple example, there are no application-logic b-threads and the actuator is driven directly by the behavioral event generated by the sensor.

In the present implementation, events are simply character strings. The semantics of triggering a behavioral event in Blockly is notifying (or resuming) any b-thread or HTML entity that registered interest in the specified event, using the `b-Sync` block or the `when_`*eventName* idiom, respectively.

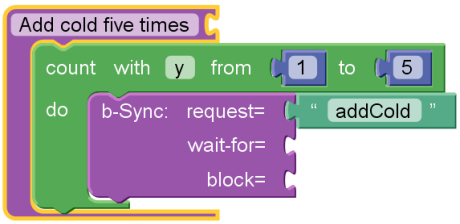
We believe that the design of a reactive behavioral application should start with analysis and determination of the sensors and actuators and the associated events. For example, Figure 3 in Section 4.3 shows such an event list for a richer example. The behaviors can then be added incrementally, as requirements are analyzed. Of course, as needed, sensors and actuators can be modified or replaced. In this approach the role of GUI design can be separated from that of application logic programming, and deciding about sensors and actuators can be seen as a development stage in which negotiation and agreement between individuals acting in these capacities take place.

4.2 Application-logic b-threads

We now move to a slightly richer example - a water-tap application similar to the one described in Section 2. This application's logic is coded in the following b-threads. One b-thread adds five quantities of hot water and terminates:



Another adds five quantities of cold water and terminates:



And, the third b-thread which interleaves the events is the same as the one shown in Section 3.2. The result is of course the interleaved run of ten events alternating repeatedly between `addHot` and `addCold`.

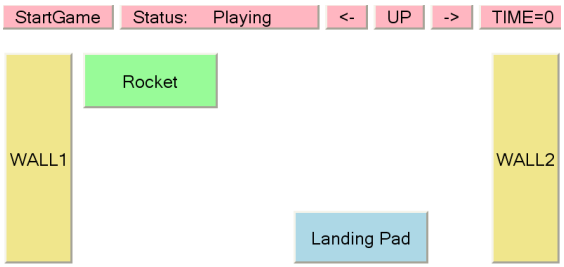
Each b-thread is contained within the Blockly block of `b-thread`. The b-thread can use any Blockly block for implementing the desired logic. To synchronize with other b-threads the `b-Sync` block is used, with the three parameters of requested, waited-for, and blocked events.

In contrast with the BPJ Java implementation, where b-thread priorities were assigned explicitly, in the Blockly implementation, b-thread priority is implied by its physical location on the development canvas, with the topmost b-thread being called first, and the lowest b-thread being called last in every execution cycle. When a new b-thread is added some dragging may be needed in order to insert it at the desired priority.

When starting an application, the Blockly infrastructure also triggers a behavioral event called `start` to activate the JavaScript behavioral programming infrastructure and execution of all b-threads.

4.3 Execution Cycle Details

To show finer points about the interweaving of b-threads in the Blockly/JavaScript environment, we examine an application for a computer game where the player attempts to land a rocket on a landing pad on the surface of a planet, or perhaps a space shuttle on a space station. The rocket moves at a fixed speed in the vertical direction. Using GUI buttons, the player can move the rocket right and left to be positioned directly above the landing pad. The player can also press the `Up` button to suspend the rocket and prevent it from going down in the next time unit. A small challenge is introduced as the landing pad keeps moving right and left either randomly or subject to an unknown plan. Two walls mark the sides of the playing areas, and the rocket cannot move past them (but does not crash when it touches them).



The game is won when the rocket lands safely on the landing pad, and is lost if the rocket either lands on the landing-pad when it is not aligned with it, or if it misses the landing-pad altogether.

As suggested in Section 4.1 we first agree on the events, and the associated sensors and actuators in the game. They are listed in Figure 3.

As described in the infrastructure section, at every synchronization point, the declarations of all b-threads are consulted, an event is selected for triggering, and b-threads that requested or waited for that event are resumed. Events that are generated by sensors are handled as follows. The function `bp.event` dynamically creates a b-thread which requests the event at the next synchronization point, and terminates once the event is triggered.

When an execution cycle ends and no event is triggered, the system is considered to have completed a *superstep*. The next behavioral event, if any, must come from a sensor reporting some external environment event. The sensor-generated event initiates a new superstep which then continues until, again, there are no events to be selected. To force a sensor-generated event to be triggered only at a beginning of a new future superstep, the sensor code should not call the event-triggering function directly, but should set it as a timer-driven JavaScript event. Due to the JavaScript single-threaded non-preemptive events mechanism the code will run as soon as the current function (the superstep) ends. This is shown below where a `RocketLeft` actuator uses a `when_` clause and the function `trigger` to serve as a `RocketTouchedLeftWall` sensor.

```
<script>
...
function trigger(exEvent) {
   setTimeout("bp.event('"+exEvent+"')",0);
}
...
</script>

<input
   value = "WALL1"
   type  = "button"
```

Sensor / Actuator	Event	Event Meaning (Description)
Sensor	BtnLeft	User clicked <-
Sensor	BtnRight	User clicked ->
Sensor	BtnUp	User clicked Up
Sensor	TimeTick	A unit of time passed
Sensor	RocketAtLeftWall	Rocket started touching left wall
Sensor	RocketAwayFromLeftWall	Rocket stopped touching left wall
Sensor	RocketAtRightWall	Rocket started touching right wall
Sensor	RocketAwayFromLeftWall	Rocket stopped touching right wall
Sensor	TouchDown	Rocket touched launch pad and is aligned with it
Sensor	Missed	Rocket reached or passed launch pad without being aligned with it
Actuator	RocketLeft	Request to redraw rocket 10 pixels to the left
Actuator	RocketRight	Request to redraw rocket 10 pixels to the right
Actuator	RocketDown	Request to redraw rocket 10 pixels down
Actuator	DisplayWin	The application determined that the player won
Actuator	DisplayLose	The application determined that the player lost
Actuator	GameOver	The application determined that the game should be stopped
Actuator	PadLeft	The application wishes the pad to move 10 pixels to the left
Actuator	PadRight	The application wishes the pad to move 10 pixels to the right

Figure 3. The external world in the rocket-landing game is represented to the behavioral application via sensors and actuators.

```
   style = "position:absolute;left:10;top:
            150;width:52;height:500"
   when_RocketLeft =
      "if(rocketX<=(leftWall+1)){
         trigger('RocketTouchedLeftWall');
       }"
/>
```

As shown here, to avoid unnecessary delays, the specified time can be zero.

Note that a separate `RocketLeft` listener is responsible for moving the rocket on the screen, and that multiple listeners can be coded for a given behavioral event. The relevant `when_` clauses may be coded under a wide selection of HTML objects — the approach does not require that the programmer chooses "correctly" HTML entities associated with a given sensor or actuator.

Two of the central questions in real-time system design is whether two events can occur exactly at the same instant, and how much time is required for the processing of all system-generated events that follow a single sensor-generated event. The user should consider the following assumptions and implementation choices as ways to simplify the application, when applicable:

- Always trigger sensor-generated events in a new superstep (using the time-out technique above)
- As in Logical Execution Time [21], assume that a superstep which consists of one sensor-generated event followed by system-generated events takes (practically) zero time.

Note that the second assumption is common, e.g., in real-time interrupt handling and in user interface programming, where event handlers must respond quickly. Thus, the BP semantics is well defined, and when b-threads communicate only through behavioral events also does not allow race conditions. SImplicity emerges partly from the fact that each b-thread can declare the events which affect it at a given state, and then handle the effects of their triggering, and completely ignore the existence of events that should not affect its state.

5. Key Scenarios where BP Benefits Emerge

Below we outline and exemplify some of the advantages and desirable capabilities of behavioral programming techniques , and the software development scenarios in which they appear. For additional comparison of BP with standard programming techniques as well as other publish/subscribe and rule-based environments see [14, 17]

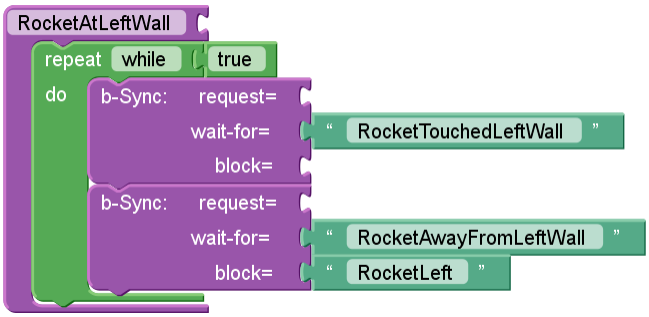
5.1 Incrementality and alignment with the requirements

The first and foremost benefit of programming behaviorally is the ability to structure application modules such that they are aligned with the requirements. As discussed at length in [14, 17], modules can be written to reflect individual requirements with little or no change to existing code. Further, as requirements are added, refined, or merely taken sequentially from a requirements document, the corresponding b-thread code can be developed and added to the application incrementally.

In the rocket-landing game, for example, assume that one first codes the following b-threads without any requirement for walls — hence without the wall-related b-threads and the four wall-related events. The coded b-threads are thus:

- Attempting to move the rocket down in response to the passing of time,
- Attempting to move the rocket right or left in response to a corresponding user click,
- Blocking the rocket's down move in response to clicking `Up`,
- Moving the landing pad right and left, and
- Detecting and announcing user winning or losing.

Then, the developer or the customer realize that walls may be required and describe the desired behavior: no advancement past the wall, but hitting a wall does not mean a crash. The appropriate sensors and b-threads to block rocket movement like

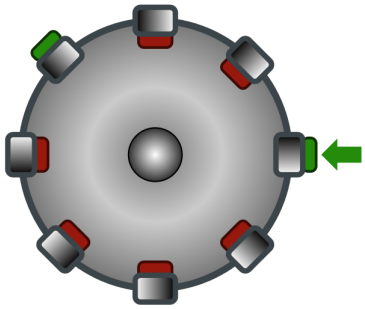


can then be added incrementally. Other capabilities which can be added with no change to the existing b-threads include additional obstacles (with behavior similar to that of the walls), increasingly hard-to-track movements of the landing pad, and a facility for advanced human players to automate their own play by programming strategy b-threads that request prescribed sequences of button-clicking events (this last example is, of course, not needed for this simple game, but is desirable and common in more advanced games).

The independence of behavior threads is also manifested in that scripts and scenarios do not have to communicate with each other directly. In the native Blockly or Scratch, broadcasting and publish/subscribe techniques can suffice for rich processing, relying only on local variables and avoiding global or shared variables. With the addition of behavioral synchronization and event-blocking (i.e., forbidding) the integrated runs are enriched, without adding a burden of peer-to-peer communication. Specifically, any event or condition that a b-thread blocks may be generated by any existing or yet-to-be-developed b-thread or sensor.

5.2 Easy state management for long scenarios

Scenarios are, of course, central to behavioral programming, and go substantially beyond the rule-based capability of waiting for an event and then triggering another event based on the system's state. For example, consider the *nullification game* application [31] — a combinatorial game where the player attempts to push in an entire array of switches placed on a rotating wheel, and where an adversary attempts to reverse the switch settings.



A game move consists of optionally pressing the switch (*Switch*), and then rotating the wheel to the next switch position (*Shift*).

In this example the animated drawing of the rotation of the wheel and the changes in switch position are performed by the GUI-processing JavaScript application package Raphael [2]. Following a game move , multiple animations have to occur, including the moving of an arrow indicating the pressing of a switch, the

movement of the switch itself, wheel rotation, and the flyover of the arrow from the human player side to the adversary side and vice versa, in an indication of whose turn it is.

In native JavaScript, without coroutines, this sequence of events would have to be programmed with callbacks and/or independent event handlers, and with variables to keep track of the evolving state and would generally look similar to:

```
when_UserWantsSwitchAndShift=
   state =  SwitchandShift0 ;
   trigger( ResponseStart );

when_ResponseStart =
   if( state =  SwitchAndShift0 ) {
      state =  SwitchAndShift1 ;
      trigger( startAnimation-MoveArrow );
   } else {
      ...
   }

when_AnimationEnded-MoveArrow =
   if( state =  SwitchAndShift1 ) {
      state =  SwitchAndShift2
      trigger( startAnimation-SwitchCurrentButtons );
   } else {
      ...
   }
...
```

In our implementation this sequence is handled naturally in a b-thread as consecutive instructions as shown in Figure 4.

Thus, our solution facilitates waiting for events inline and not only by callbacks. It should be noted that several JavaScript pre-compilers, such as NarrativeJS [25], StratifiedJS [29], and others, allow for sequential event handling in JavaScript, similarly to the ability described in this section. These extensions to JavaScript can be viewed as implementing a subset of the complete behavioral protocol, often without events blocking or multiple b-threads. In this context it should be noted that BP is different from rule-based systems is in that blocking in BP is targeted at events, regardless of their originator, as opposed to rule-based system in which blocking is by disabling rules (see, e.g., [7].

5.3 Integration with standard programming

Coding behaviorally does not mean that all calculations and data processing performed by the application must be based on events. A behavioral application can contain substantial pieces that are coded in standard programming languages. In the context of JavaScript and Blockly, JavaScript functions can be called from Blockly blocks, or from the sensors and actuators. In the nullification game example the calculations of the adversary strategy which is based on deBruijn sequences [31], are performed by calling a JavaScript function. Needless to say, the Raphael animations discussed above also demonstrate the power of such integration capabilities.

5.4 Programming with parallel continuous entities with well-defined semantics

As in the LSC language, the basic units of program code (the actors) in the current Blockly and JavaScript environment are scripts that run "all the time". These scripts take desired actions when specific conditions are met, or constantly express their opinion about the global state from a narrow viewpoint based on events that they listen-out for. As observed in [10, 23], this design appears to be "natural" in the sense that it was adopted by children who were not explicitly guided to use it.

In behavioral programming, instantiation, activation and repeated synchronization of such scripts is easy, often "free", i.e., automatic, in comparison to the more elaborate setup commonly needed in other languages and contexts.

A problem in Scratch pointed out by Ben-Ari and discussed in Scratch forum [3] is that the semantics of interweaving scripts depends on intricate properties of the model whose effects on scheduling are sometimes hidden from the programmer. For a less intricate but illustrative example, consider the scripts

The programmed flow is that once the green flag is clicked, the first script broadcasts `mymsg`, waits 1 second and then broadcasts the message again. Whenever this message is received by the second script, the variable X is incremented, and after 2 seconds, the variable Y is incremented. However a result of running and stopping the scripts is

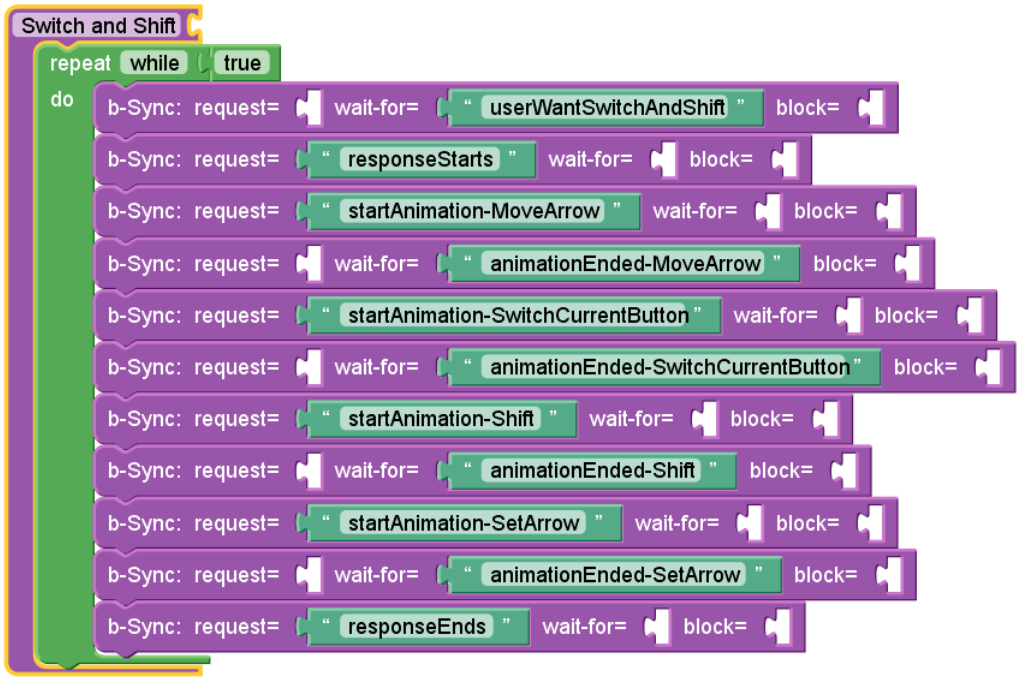


Figure 4. A scenario of consecutive instructions (shown here in Blockly) facilitates natural and implicit state management.

suggesting that when the message is broadcast a second time, the first execution of the second script is interrupted and is never resumed, thus Y is incremented only once. When the delay in the first script is set to 5 seconds instead of 1, the final value of Y is 2. We did not find documentation of this semantics of Scratch.

Our approach, in this paper, is to view scripts as global entities with well-defined scheduling, synchronization protocol, and interweaving semantics. Using our Blockly and JavaScript environment, an application similar to the above example will have to be coded differently. Depending on the programmer's solution, when the behavioral event associated with second message is triggered, this event will either: (a) cause no effect, as at the synchronization point when it is triggered no b-thread will declare it as a requested or waited-for event (instead, the second b-thread is only waiting for time-delay to pass), or (b) it will be processed by another running instance of the second b-thread class, which would be explicitly started by the application to catch such events while other instances wait for the time-delay to pass. In either case, the semantics will be well defined and the composite behavior will be readily predictable.

5.5 Priority as a first-class idiom

When multiple simultaneous behaviors are active and vote with their event declarations as votes' with regard to the progress of an application, priority becomes a useful construct. In our implementation, b-thread priorities are based on their easy to manage order on the canvas, i.e., a b-thread laid higher on the canvas has a higher priority. The priorities of b-threads that are perfectly aligned with each other vertically are ordered based on the b-threads' left-to-right horizontal order. In addition, the single-threaded sequential calling of JavaScript coroutines provides for well-defined semantics of the "simultaneous" part of the behavior, and of the corresponding effects on any variables that are shared between b-threads.

5.6 The secondary role of the behaving objects

In Scratch, scripts are anchored on game characters called *sprites* which are perceived as the behaving entities. On one hand, the sprites can be readily thought of

as agents or actors in their own right, which in turn rely on scripts as their implementation or as another level in their hierarchy. On the other hand, following the detailed discussion of inter-object versus intra-object behavior in [11], behavior scenarios are not necessarily anchored on a given object. For example, in [23] the researchers observed that when forced to associate scripts with sprites, young programmers split the scripts of the (correct) behavior of one game character on multiple sprites, and game rules were associated with arbitrary sprites. This further puts into question the need to focus on "the behaving entity" when observing a behavior. The Blockly/JavaScript environment presented here does not force the programmer to associate desired behaviors with behaving objects.

We propose that there is an important distinction between objects in general, as in object oriented programming, and the concept of behaving entities that are tangible in the user's eyes. For example, in an application with a graphical user interface, it is not always best to anchor the code on the elements on the screen. It may be better, instead, to code scenarios that involve multiple tangible entities as standalone modules. Of course, scenarios, events, screen objects, etc., may be coded with object-oriented programming.

6. Discussion and Future Work

We presented an implementation of behavioral programming principles in JavaScript and in Google Blockly. The result is a proof-of-concept for a programming environment which appears to be natural and intuitive, and highlights interesting traits of behavioral programming. An important next step is to show the scalability of the concepts and their applicability to complex systems. As our understanding of BP develops, it will also be interesting to expand the discussion of comparisons of BP to other paradigms, which appear in [14, 17], to additional platforms such as rule-based (e.g., [7]) and functional reactive programming environments (e.g. [24]).

The ease of creating new language constructs in Blockly and the fact that visual block-based programming seems natural to individuals with little computer training, call for using Blockly as a test-bed for investigating the naturalness of new programming idioms. For example, nesting blocks which state things like "while forbidding events `a,b,c` do", or "exit the present block when event `e` is triggered" have the potential of making behavioral programs simpler and more readable than when written with just basic `b-Sync`. Specifically, they can simplify the management of the sets of requested, waited-for and blocked events, and reduce the need to wait, in a single command, for multiple events and then check which of them was indeed triggered.

The availability of Blockly and JavaScript 1.7 for mobile platforms, such as Android smartphones, opens the way for a wide range of applications, and the single-threaded JavaScript implementation may further facilitate running behavioral applications with many b-threads in environments which do not readily accommodate large numbers of concurrent Java threads.

The combination of implementations of behavioral programming principles in popular languages, with IDEs that are particularly user friendly, and with a growing set of natural programming idioms, may further facilitate programming in a decentralized-control mindset by wider communities of beginners and professionals.

Acknowledgments

We would like to thank David Harel for support and valuable discussions and suggestions in the development of this work. The focus on Scratch-like languages is inspired by our on-going collaboration with Orni Meerbaum-Salant and Michal Gordon and by suggestions by Eran Mizrachi. We thank the anonymous reviewers for their valuable comments and suggestions for improvements, which we tried to incorporate.

The research of Assaf Marron and of Guy Wiener 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 David Harel from the European Research Council (ERC) under the European Community's FP7 Programme. The research of Gera 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.

A. Appendix: Behavioral Programming using Coroutines

This appendix presents a generic algorithm for implementing the behavioral programming event-driven loop using coroutines. The motivation for pursuing coroutines for b-threads is twofold. First, coroutines

consume less resources than threads or processes, and therefore can be found in embedded scripting languages that aim at minimizing resource allocation, such as JavaScript and Lua. Second, this implementation does not require concurrent execution, that is not always desirable, e.g., when debugging or verifying.

A.1 Introduction to Coroutines

Coroutines provide a mechanism for executing two or more control-flows independently, without requiring a thread scheduler. Instead, the control is passed from one coroutine to the other explicitly. Hence, coroutines are also referred to as type of *non-preemptive multi-threading*.

While coroutines are supported by a many programming languages, each with its own syntax for defining and using them. In this section we will use the following notation.

- A coroutine is a function that instead of the standard `return` statement uses the special `yield` statement. Similarly to `return`, `yield` passes the control flow back to the caller together with a given value. Unlike `return`, when the coroutine is called again, it resumes at the state of the previous `yield`, as if it was paused and resumed. When the coroutine is resumed, the `yield` expression evaluates to the value sent to the coroutine by the caller.
- The `create` statement takes the name of a coroutine and creates a coroutine object. This object acts as a unique identifier of the state of the coroutine. Each subsequent call to the same identifier will continue from the state of the last call to `yield`. Notice that `create` does not run the coroutine itself.
- The `send` statement takes a coroutine and a value, and resumes the coroutine with the state of the last call to `yield`, returning the given value. If this is the first call after `create`, the coroutine will start, ignoring the value.
- The `alive` predicate tests if a coroutine object is still running. It returns `true` after the coroutines is created, and as long as the coroutine function is not completed. It returns `false` otherwise.

A.2 The Algorithm

```
// Queues of b-threads and their bids
running = []
pending = []

// ------------------------------------------
// Adding a b-thread translates to pushing it
// to the running queue and creating a coroutine
// for it
// ------------------------------------------
function addBThread(prio, func) {
   queue running, {priority: prio,
                   bthread: create func}
}

// ------------------------------------------
// Run is called to begin a superstep. It invokes
// the coroutines sequentially, collects the bids,
// selects the next event, and calls itself
// recursively
// ------------------------------------------
function run() {
   while (running is not empty) {
      bid = unqueue running
      bt  = bid.bthread
      newbid = send bt, lastEvent
      if (bt has not finished) {
         newbid.bthread = bt
         newbid.priority = bid.priority
         queue pending, newbid
      }
   }

   lastEvent = the first event (w.r.t. priority)
               that some b-thread requested and
               no b-thread blocked

   if (lastEvent is not equal to undefined) {
      temp = []
      while (pending not empty) {
         bid = unqueue pending
            if (bid specifies waiting-for
                  or requesting lastEvent)
               queue running, bid
            else
               queue temp, bid
      }
      pending = temp
      run()
   }
}
```

References

[1] H. Abelson and M. Friedman. MIT App Inventor. URL: http://appinventor.mit.edu, accessed Aug. 2012.

[2] D. Baranovskiy. Raphael. URL: http://raphaeljs.com/, accessed Aug. 2012.

[3] M. Ben-Ari and J. Maloney. Scratch project forum discussion. URL: http://scratch.mit.edu/forums/viewtopic.php?id=8130, accessed Aug. 2012.

[4] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[5] N. Eitan, M. Gordon, D. Harel, A. Marron, and G. Weiss. On Visualization and Comprehension of

Scenario-Based Programs. *Int. Conf. on Program Comprehension (ICPC)*, 2011.

[6] D. Elza. Waterbear language web site. URL: http://waterbearlang.com/, accessed Aug. 2012.

[7] J. Fenton and K. Beck. Playground: an object-oriented simulation system with agent rules for children of all ages. *ACM SIGPLAN Notices*, 24(10):123–137, 1989.

[8] N. Fraser. Google blockly - a visual programming editor. URL: http://code.google.com/p/blockly, accessed Aug. 2012.

[9] M. Gordon and D. Harel. Generating executable scenarios from natural language. *Computational Linguistics and Intelligent Text Processing*, pages 456–467, 2009.

[10] M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the main course? observations on naturalness of scenario-based programming. *17th Annual Conference on Innovation and Technology in Computer Science Education*, 2012.

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

[12] D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 485–499, 2007.

[13] D. Harel and I. Segall. Synthesis from live sequence chart specifications. *Jour. Computer System Sciences*, 78:3:970–980, 2012.

[14] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90–100.

[15] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 378–398, 2002.

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

[17] D. Harel, A. Marron, and G. Weiss. Programming Coordinated Scenarios in Java. In *Proc. 24th European Conf. on Object-Oriented Programming (ECOOP)*, pages 250–274, 2010.

[18] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. In *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, pages 279–288, 2011.

[19] D. Harel, A. Marron, G. Weiss, and G. Wiener. Behavioral programming, decentralized control, and multiple time scales. In *Proc. of the SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, pages 171–182, 2011.

[20] D. Harel, G. Katz, A. Marron, and G. Weiss. Non-intrusive repair of reactive programs. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2012.

[21] T. A. Henzinger, C. M. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, 23(1):50–64, 2003.

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

[23] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of Programming in Scratch. In *Proc. of the 16th Annual Joint Conf. on Innovation and Technology in Computer Science Education (ITICSE)*, pages 168–172. ACM, 2011.

[24] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[25] N. Mix. Narrativejs. URL: http://www.neilmix.com/narrativejs/, accessed Aug. 2012.

[26] J. Moenig and B. Harvey. BYOB Build your own blocks (a/k/a SNAP!). URL: http://byob.berkeley.edu/, accessed Aug. 2012.

[27] Mozilla Foundation. FireFox JavaScript 1.7 -. URL: http://developer.mozilla.org/en/New_in_JavaScript_1.7, accessed Aug. 2012.

[28] E. Naone. HTML 5 could challenge Flash. URL: http://www.technologyreview.com/news/418130/html-5-could-challenge-flash/, accessed Aug. 2012.

[29] Oni Labs. Stratifiedjs. URL: http://onilabs.com/stratifiedjs/, accessed Aug. 2012.

[30] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: Programming for All. *Comm. of the ACM*, 52(11):60–67, 2009.

[31] G. Weiss. A combinatorial game approach to state nullification by hybrid feedback. In *46th IEEE Conference on Decision and Control*, pages 4643–4647. IEEE, 2007.

[32] G. Wiener, G. Weiss, and A. Marron. Coordinating and Visualizing Independent Behaviors in Erlang. In *Proc. 9th ACM SIGPLAN Erlang Workshop*, 2010.