A use-case for behavioral programming: An architecture in JavaScript and Blockly for interactive applications with cross-cutting scenarios [1]

Adiel Ashrov[a], Assaf Marron[b], Gera Weiss[a], Guy Wiener[c]

[a] *Ben-Gurion University, Beer-Sheva, Israel*
[b] *Weizmann Institute of Science, Rehovot, Israel*
[c] *HP Labs, Haifa, Israel*

Abstract

We combine visual programming using Google Blockly with a single-threaded implementation of behavioral programming (BP) in JavaScript, and propose design patterns for developing reactive systems such as client-side Web applications and smartphone customization applications as collections of independent cross-cutting scenarios that are interwoven at run time. We show that BP principles can be instrumental in addressing common software engineering issues such as separation of graphical representation from logic and the handling of inter-object scenarios. We also show that a BP infrastructure can be implemented with limited run-time resources in a single-threaded environment using coroutines. In addition to expanding the availability of BP capabilities, we hope that this work will contribute to the evolving directions of technologies and design patterns in developing interactive applications.

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

1. Introduction

The behavioral programming (BP) approach is an extension and generalization of scenario-based programming, which was introduced in [8, 15] and extended in [20]. In behavioral programming, individual requirements are programmed in a decentralized manner as independent modules which are interwoven at run time. Each module describes a behavior

[1] A preliminary version of this paper appeared as A. Marron, G. Weiss, G. Wiener,"A Decentralized Approach for Programming Interactive Applications with JavaScript and Blockly", AGERE! workshop at SPLASH 2012. The additions in this journal version include a section on BP for Android smartphone, new and enhanced examples with more detailed descriptions of interfaces and incremental development, more details about the JavaScript and Blockly implementations, and extended comparisons and positioning of BP relative to other languages and development approaches.

Email addresses: ashrov@cs.bgu.ac.il (Adiel Ashrov), assaf.marron@weizmann.ac.il (Assaf Marron), geraw@cs.bgu.ac.il (Gera Weiss), guy.wiener@hp.com (Guy Wiener)

scenario that may be cutting across multiple objects or "behaving entities" in the system. Advantages of the approach include facilitation of natural and incremental development and facilitation of early detection of conflicting requirements [21, 13]. A review of research and tool development in BP to date appears in [18]. While BP 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 develop BP infrastructure for a specific reactive-systems application domain, propose design patterns for using this technology and show that BP principles can be instrumental in addressing common software engineering issues. Specifically, we demonstrate the implementation of behavioral programming in JavaScript and in Google Blockly (`www.code.google.com/p/blockly/`) for interactive technologies such as the client side of Web applications and smartphone customization.

We propose that scenario-based programming techniques complement and offer advantages over standard programming in these specific domains. Consider, for example, a Web 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 or a smartphone application that has to react to a particular sequence of telephone- or location-related events. Using a non-behavioral style with, e.g., JavaScript or Java, the programmer would handle each button-click or phone event separately, and introduce special code to manage state for recognizing desired sequences of events. We argue that with BP such requirements can be coded in a single sequential script with state management being implicit and natural rather than explicit. Specifically in the context of JavaScript we show how this can be done with coroutines rather than with standard event handlers. Further, multiple cross-cutting requirements of this form can be coded separately, in a loosely coupled manner and then be interwoven at run time, while communicating via a form of publish-subscribe protocol . Thus, for example, programmers can use the BP design pattern to separate the handling of common cases from exceptions, and to supplement optimistic coding with separately coded error handling.

We hope that this paper will help incorporation of scenario-based and behavioral programming principles into a wide variety of new and existing environments and will help add them into the basic set of design patterns that are understandable by and useful for novice and expert programmers alike.

The paper is structured as follows. In Section 2 we provide a brief introduction to BP. In Sections 3, 4, and 5 we describe BP infrastructure implementation in JavaScript and in Blockly respectively, including infrastructure design choices, integration with HTML, and proposed application design patterns. In Sections 6, 7, 8 and 9 we provide detailed examples for applying the design patterns, and in Section 10 we discuss the benefits emerging from the proposed designs and techniques and contexts in which they emerge. In Section 11 we compare BP to other techniques. The code for the infrastructure and examples is available online at `www.b-prog.org`.

A note 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, and 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

For completeness and self-sufficiency of this paper, we provide below a brief introduction to BP. For expanded introduction see, e.g., [18] and references therein.

A preliminary assumption is that we are concerned with reactive systems or applications that are 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 into effects on the world outside the application, which may be a physical world or other systems.

In a behavioral program, event sequences that constitute the integrated system behavior are generated by independent behavior modules that are interwoven at run time in a protocol that is an enhanced combination of publish-subscribe and aspect orientation. Each module (called *b-thread*) represents an aspect of the system behavior by indicating at all times events which from its own point of view must, may, or must not occur next. Section 11 presents comparison and positioning of BP relative to publish-subscribe and aspect orientation as well as other programming approaches and languages.

Specifically, each b-thread is a procedure that repeatedly specifies sets of events to be considered for triggering (called *requested events*) and events whose triggering is forbidden (called *blocked events*).The specification is provided by the b-thread by calling a designated method that registers the specified sets of events and then suspends the b-thread. When all b-threads provide the information and are suspended we say that the system has reached a synchronization point. An event selection mechanism then triggers one event that is requested and not blocked by notifying and resuming all b-threads that requested the event. A b-thread can also declare events that it *waits for* even without requesting them, and it too is notified and resumed when the waited-for events are triggered. All resumed b-threads can perform arbitrary processing, modify their declarations of requested, blocked and waited-for events, and proceed to the next synchronization point. The specifications of the b-threads

that were not resumed are retained for consideration at the next synchronization point. When all resumed b-threads reach their next synchronization point and are suspended, the process of event selection with consultation of all b-threads repeats. The mechanism does not require direct communication between b-threads or other forms of explicit b-thread-to-b-thread coupling.

This design facilitates incremental non-intrusive development. Consider for example a system for controlling water level in a tank with hot and cold water sources. Assume that we start with a b-thread that repeatedly waits for the event `WaterLevelLow` and then requests five times the event `AddHot`. Then, when adding five water quantities for every sensor reading proves to be insufficient, a new requirement is introduced to also add five quantities of cold water, and the developer adds a b-thread which similarly waits for `WaterLevelLow` and requests five times the event `AddCold`. After observing a run in which the five `AddHot` events occurred before the first `AddCold` event, a new requirement is introduced, to the effect that water temperature should be kept relatively stable. The developer then adds a b-thread that interleaves `AddHot` and `AddCold` events using the event-blocking idiom by repeatedly performing the steps of first waiting for `AddHot` while blocking `AddCold` and then waiting for `AddCold` while blocking `AddHot`. In this example, with each new requirement, a new b-thread was added without changing existing b-threads and without direct interaction with them.

In behavioral programming, an important step in developing a system is to determine a common set of events that are relevant to multiple scenarios. While this requires contemplation, it is often easier to identify these events than to determine objects and associated methods, especially when the events emerge directly from the text of the specification independently of object design or system structure. By default, events are opaque and carry nothing but their name, but they can be extended with rich data and functionality. In addition, the incremental traits of BP and the smallness of b-threads (see Section 10) facilitate subsequent adding and changing of event choices.

When multiple events are requested and not blocked, event selection may be subject to different policies. In the BPJ package, in Erlang, and in the JavaScript and Blockly implementation presented in this paper, events are selected according to a priority order induced by a priority order between b-threads and within each requested-events set. Other policies include random selection, lookahead in order to meet desired goals [19, 16], and applying run-time learning [9].

As the BP approach raises questions on conflict detection and resolution, comprehension, and synchronization, tools and architectures have been developed for program verification and synthesis [21, 17, 30], automated program repair [23] visualization and debugging [10], and combining synchronous and asynchronous communications in behavioral applications [22].

It should be noted that BP complements and coexists with existing development platforms and methodologies, and it is the stakeholder's decision where and how much to use scenario composition and where to rely on existing capabilities of the underlying language.

3. Coordinating Behaviors Written in JavaScript

The contribution of this paper is in providing two layers for implementation of BP in the context of web development. In this section we begin with a description of the first layer which is a library for using the BP design pattern in the JavaScript programming language. In the next section we will describe a second layer that builds upon the JavaScript library and provides programmers with a visual language based on the Blockly library.

3.1. b-threads as coroutines

In principle, the concepts of BP are language independent and indeed they have been implemented in a variety of languages using different techniques. For example, in the BP package for Java (BPJ) [20], b-threads are executed as ordinary Java threads. Synchronization and declaration of requested, blocked and waited-for events is done by the b-threads by calling a method named `bSync`, passing it the three sets of events as parameters. In addition to receiving arrays of concrete event objects, the `bSync` method supports, for the waited-for and blocked events arguments, receiving a function as a parameter. When a function parameter is passed, say, for the blocked event set, the event when a requested event is considered for triggering it is passed to the function to determine if it is blocked or not. Function parameters are used in the waited-for events parameter, in a similar manner, to determine which b-threads should be resumed when an event is triggered. This support for a function instead of a concrete array of objects, provides support for blocking and waiting for very large and possibly infinite event sets.

The event selection mechanism is provided by the BPJ library and is invoked internally by the `bSync` method. The Erlang implementation uses Erlang processes for the b-threads. In LSC, the developer draws charts of scenarios using a visual interface, and a control mechanism coordinates the advancement of all charts along their locations. In each implementation, certain language facilities are needed in order to control the execution, synchronization and resumption of the simultaneous behaviors. In Java and Erlang the mechanism coordinates independent threads or processes using language constructs such as `wait` and `notify`. The PlayGo LSC implementation uses aspect oriented programming with AspectJ to synchronize internally between Java modules generated for the different charts.

When considering a BP implementation in the context of a web browser, since the application is typically executed as a single thread, the coordination technique needs to be different than that used for Java and Erlang. Programming literature discusses the use of *coroutines* for implementing orthogonal control flows within a single thread. As described by Knuth in [29], coroutines exchange control symmetrically by *yielding* values one to another, contrary to the asymmetrical way where *returning* a value terminates a subroutine. One can view the explicit control exchange of coroutines as replacing a thread scheduler. Therefore, coroutines are sometimes referred to as a form of *non-preemptive multi-threading*.

The `yield` command was introduced recently in JavaScript 1.7 [38]. JavaScript 1.7 is a non-standard dialect of the ECMAScript standard. This extension is currently supported only in the Firefox browser, however it is scheduled to become part of the next version of the standard, namely ECMAScript 6. The `yield` command returns a value to the caller,

but unlike `return`, subsequent calls (using the `next` or `send` methods) continue from the following command, with the same state of the callee. JavaScript functions that explicitly use `yield` are called *generators*, as they generate a series of values. The use of generators allows for implementing the basic coroutines protocol. Some JavaScript libraries, such as `task.js` (`www.taskjs.org`) extend it to a richer multitasking interface.

The advantages of using coroutines for b-threads as compared with the use of processes or threads is twofold. First, coroutines consume less resources, and therefore can be found in embedded scripting languages that aim at minimizing resource allocation, such as JavaScript and Lua. Second, the strictly sequential processing of coroutines ensures that the ordering of internal events for a given sequence of external events is always the same, and avoids low level race conditions. It also helps guarantee that each external event is fully processed before the processing of the next event begins. As a result, debugging and verification are also simplified. It should be noted that a behavioral program where b-threads do not share data has the desired property that arbitrary delays between any two synchronization points in any b-thread do not change the sequence of events generated, if the external events arrive at the same super-steps. When variables are nevertheless shared, their access must be disciplined, and the usage of coroutines further supports this purpose.

Thus, in our BP for JavaScript infrastructure, b-threads are implemented as JavaScript generators. The generator of each b-thread determines and returns, using the `yield` command, this b-thread's sets of requested, blocked and waited-for events. The generator function can use the full power of JavaScript for any processing and for preparing the three sets of events.

JavaScript requires that the `yield` command be visible in the source of a function to be executed as a generator. Hence, we chose in the present implementation not to hide the `yield` command within a method dedicated to behavioral synchronization and declarations, such as the `bSync` method in the Java implementation. The `bSync` wrapper is provided in the Blockly interface.

For example, the code for the application logic of the water-tap example discussed above is shown in Figure 1. The function `bp.addBThread` is used to add the b-threads. The parameters to `addBthread` are a description of the b-thread, a priority designation (explained in Section 3.2), and the application logic of the b-thread as a generator function definition.

```
var x;
var y;

bp.addBThread('Add hot five times', priority++, function() {
    yield({ wait: ['waterLevelLow']});
    for (x = 1; x <= 5; x++) {
        yield({request: ['addHot'] });
    }
});
bp.addBThread('Add cold five times', priority++, function() {
    yield({wait: ['waterLevelLow']});
    for (y = 1; y <= 5; y++) {
        yield({ request: ['addCold'] });
    }
});
bp.addBThread('Interleave', priority++, function() {
    while (true) {
        yield({wait: ['addHot'], block: ['addCold']});
        yield({wait: ['addCold'], block: ['addHot']});
    }
});
```

Figure 1: Coding the b-threads for the water-tap example in JavaScript. The main script adds the three b-threads — one that requests the event `addHot` five times, one that requests the event `addCold` five times and one which causes the event triggering to be interleaved by repeatedly waiting for `addHot` while blocking `addCold` and vice versa. Once added, b-threads are automatically started and participate in the next synchronization point (see Section 3.2).

3.2. Execution cycle

The BP for JavaScript infrastructure executes in cycles. Each new b-thread is started by the infrastructure and executes until its first synchronization point. Whenever a b-thread arrives at a synchronization point the infrastructure collects its declaration of requested, waited-for and blocked events. Whenever all running b-threads are suspended at a synchronization point, the infrastructure selects for triggering an event that is requested and not blocked. The candidate events are examined according to the order in which the originating b-threads were first added, and within the requests of each b-thread according to the order in the requested-events array. Each candidate requested event is checked against the blocking declarations of all b-threads — either for presence in the passed array of events, or by passing the event to the the specified function. The infrastructure then resumes all b-threads that requested or waited-for the event by calling them (and only them), using the `send()` method, one by one. The determination of which b-thread to resume is again done by examining all requested-event arrays as well as waited-for events specifications — either as event arrays or as a function provided by the b-thread which determines if a given event is waited for by this b-thread. When each called b-thread reaches its next synchronization point, the next one is called. The order of b-thread invocation is according to the priority assigned when the b-thread was added. When each of the resumed b-threads is suspended again (or has terminated), a new cycle begins. The main algorithm of the BP execution infrastructure which provides repeated synchronization and event selection is described in Figure 2.

```
running ← ∅
pending ← ∅
lastEvent ← undefined
procedure ADDBTHREAD(prio, func)
    queue running, {priority ↦ prio,
                    bthread ↦ create a coroutine instance from func
                    request, wait, block ↦ undefined}
procedure RUN
    bids ← ∅
    while running ≠ ∅ do
        bid ← unqueue running
        bt ← bid.bthread
        newbid ← send bt, lastEvent      ▷ newbid contains request, wait and block values
        queue bids, newbid
        if bt is not terminated then
            newbid.bthread ← bt                          ▷ bt is the updated coroutine object
            newbid.priority ← bid.priority
            queue pending, newbid
    lastEvent ← e | ∃p ∈ bids, e ∈ p.request,      ▷ Select a requested event e such that
                    ∄q ∈ bids, e ∈ q.block,                ▷ e is not blocked, and
                    ∄r ∈ bids, r.priority > p.priority     ▷ e has the highest priority
    if lastEvent ≠ undefined then
        running ← {p | p ∈ pending, lastEvent ∈ p.request ∪ p.wait}
        pending ← {q | q ∈ pending, lastEvent ∉ q.request ∪ q.wait}
        RUN()
```

Figure 2: The synchronization and event selection algorithm of BP using coroutines. Queues of b-threads and their bids are maintained. A b-thread is added by pushing it to the running queue and instantiating a coroutine for its function. The function `run` is called to begin a superstep when a an environment-generated event is triggered. It invokes the coroutine instances sequentially, collects the bids, selects the next event, prepares the queue of b-threads that have requested or are waiting for this event (and hence should be resumed) and calls itself recursively to resume these b-threads. The `pending` queue maintains the b-threads that are not resumed following this event. In the b-threads, `yield` passes the control flow back to the caller together with its bid parameters — requested, waited-for and blocked events. When the coroutine instance is called again, it resumes at its previous state and the `yield` expression evaluates to the value sent by the caller, namely `lastEvent`. The first `send` after coroutine instantiation starts the execution of the function from its beginning. This algorithm is implemented in JavaScript using generators. When starting an application, the BP infrastructure calls `run` in order to begin the cyclic process of event selection and b-thread execution.

When there are no requested events that are not blocked at a given synchronization point, no event is triggered, and the system is considered to have completed a *superstep*. The next behavioral event, if any, must come from an external module (referred to as *a sensor*) reporting some external environment event. As will be explained in Section 3.3, the sensor-generated event initiates a new superstep which then continues until, again, there are no events to be selected.

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. As discussed in detail in [22], the user should consider the following assumptions and implementation choices as ways to simplify the application, when applicable:

- No two events occur in the same instant
- As in Logical Execution Time [25], a superstep always consists of one external, environment-generated event followed by system-generated events.
- A superstep takes (practically) zero time.

Note that the third assumption is common, e.g., in real-time interrupt handling and in user interface programming, where event handlers must respond quickly.

3.3. Input and Output - Sensors and Actuators

Inputs from the external world and from other applications are processed by modules, referred to as *sensors*, which use domain-specific interfaces to sense the external event (e.g. `onclick` callback listener in an HTML button, or a repeated invocation of some probing method). The sensor module then calls the JavaScript BP infrastructure method `bp.event` with a behavioral event that represents the external event. The method `bp.event` internally creates a new b-thread that requests the new behavioral event and terminates, adds this b-thread to the system, and invokes the `run` method of Figure 2 to start another cycle of coroutine execution and event selection:

```
BProgram.prototype.event = function(e) {
    var name = 'request ' + e;
    var bt = function() {
        yield({request: [e], wait: [function(x) { return true; }]});
    };
    this.addBThread(name, 1, bt);
    this.run(); // Initiate a super-step
};
```

Outputs and effects on the environment can be generated by standard JavaScript code in the b-threads. A b-thread can call any method and perform any necessary actions to cause the desired effects between synchronization points. For separation of application logic from output interfaces, we propose a design pattern in which the actuation of effects on the environment is designed as b-threads that are dedicated to that purpose, i.e., they repeatedly wait for behavioral events, and, in response to each one, execute the necessary

domain-specific code. See Section 4, and Sections 6 through 9 for examples of sensors and actuators in contexts such as HTML and Android smartphone.

We believe that the design of a reactive behavioral application should start with analysis and determination of the sensor and actuator interfaces to the environment, and the associated events. For example, Table 1 in Section 7 shows such a list of sensors, actuators and associated events for a computer-game 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.

It should be noted that when a b-thread simulates the occurrence of an external event by requesting the corresponding behavioral event, it is the programmer's responsibility to make sure that the event is triggered after the completion of the current superstep. In the present implementation this can be done, for example, by not calling `bp.event` directly, but instead, a different function called `trigger` which queues the event, with other external events, such that each of them is triggered in its own consecutive superstep. This ensures all system-generated events that can become enabled as part of handling a given external event are processed before the next sensor-generated event is triggered. As stated above, this also maintains a deterministic execution order and helps avoid low level race conditions, reetrancy issues and looping event cascades. The present implementation of `trigger()` uses the `setTimeout` function of JavaScript specifying a delay of zero seconds. The JavaScript single-threaded non-preemptive scheduling will run the time-delayed code only after the current function ends, i.e., after the end of the current superstep.

3.4. Alternatives to Current Implementation

In basing the execution protocol directly on coroutines we have taken several decisions that lead to the current implementation of BP in a web browser environment. In this section we review several technical alternatives.

Our first decision was to prefer the native language support of generators over simulating coroutines using Continuation-Passing Style (CPS). These features are not considered exotic by the JavaScript community, rather the opposite — one of the more commonly-used JavaScript frameworks, Node.js (`www.nodejs.org`), makes extensive use of function values to represent its event-driven control flow. Similarly, an alternative implementation of `bSync` could have taken a fourth argument, `cont`, that specifies what to do once the b-thread is awakened:

```
bSync({request: R, wait: W, block: B, cont: function(e) { // Handle e... }});
```

A key advantage of the above approach is that it can be encoded in any browser implementing the current ECMAScript standard [1], not just JavaScript 1.7 — practically, any modern web browser. We nevertheless ruled against it, preferring that the scenarios be described in a plain sequential manner, to make them easier to understand. For example, consider the code listings in Figure 3. While all the code snippets are equivalent, the use

```
bSync({
 request: R1, wait: W1, block: B1,
 cont: function(e1) {
  bSync({
   request: R2, wait: W2, block: B2,
   cont: function(e2) {
    bSync({
     request: R3, wait: W3, block: B3,
     cont: function(e3) {
      // ...
     }
    })
   }
  })
 }
});
```

(a) Inlined CPS

```
function f1() {
  bSync({request: R1, wait: W1, block: B1,
         cont: f2});
}

function f2() {
  bSync({request: R2, wait: W2, block: B2,
         cont: f3});
}

function f3() {
  bSync({request: R3, wait: W3, block: B3,
         cont: ...});
}
```

(b) Phased CPS

```
yield({request: R1, wait: W1, block: B1});
yield({request: R2, wait: W2, block: B2});
yield({request: R3, wait: W3, block: B3});
```

(c) Coroutines

Figure 3: A comparison between two implementations of `bSync` using continuation passing style and one based on coroutines

of CPS forces the code in Figure 3a into a cumbersome diagonal shape, and in Figure 3b the scenario is split into several functions. Another possible alternative (not shown) is using *promises* (see, e.g., the Q library in `https://github.com/kriskowal/q`). The code would appear similar to Figure 3b, except that the separate functions do not have to be named. We preferred the simple version in 3c, despite the currently limited browser support.

The problem described above could have been solved by the use of CPS preprocessors, such as NarrativeJS [36], StratifiedJS [39], and others. Our second design choice was not to rely on these auxuliaries, despite their promise to bring sequential event-handling to JavaScript. We did not want to add a preprocessing phase to the development process when using BP, as this might not be in line with the users' development routine. This, however, was an administrative decision rather than a technical one. Implementing the BP algorithm described in Figure 2 using the CPS transformation should be simple, and the result should be equivalent to using `yield`.

4. Integration with HTML

The integration of the JavaScript application with the external world can be done in several ways. For example, the `waterLevelLow` event can be simulated as the clicking of a GUI button by the user which causes the invocation of the `bp.event` method which generates the behavioral event. The corresponding HTML code that the developer provides is:

```
<p>
   <input
```

```
                value="Report Water Level Low"
                type="button"
                onclick="bp.event('waterLevelLow');"
                style="position:relative;background-color:LightPink;"
                />
</p>
```

In the above code, the `onclick` construct captures the HTML event and invokes `bp.event`.

Actuators can be implemented by direct usage of JavaScript methods in the JavaScript code, coded in Blockly blocks designed for this purpose. In addition, we have created a mechanism by which arbitrary HTML code can be invoked in response to behavioral events. That HTML code can then perform HTML functions, or invoke JavaScript methods. For example, assume that the actual opening of the water taps is simulated by displaying the event names on the screen, where the desired output from running the application is the sequence `addHot, addCold, addHot, addCold,`..., etc. The HTML code for this is:

```
<center style="font-size: x-large"
   when_addHot  = 'innerHTML += "<span style=\"color:black\">addHot</span><br>"'
   when_addCold = 'innerHTML += "<span style=\"padding-left:100px\">addCold</span><br>"'
>
```

The verb `when_`*eventName*, is an HTML entity that activates JavaScript code. The `when_`*eventName* specifications can be entered on any HTML object, and multiple listeners can be coded for a given behavioral event. To implement the `when_`*eventName* construct the infrastructure contains a b-thread that always waits for all events and, when an event is triggered, uses `jQuery` (`www.jquery.com`) to scan the HTML page for `when_`*eventName* specifications and then invokes the found scripts.

It should be noted relative to this and other code examples, that the shown code structures are most basic. The developer is free to use HTML and JavaScript capabilities as needed, including using a JavaScript function to manipulate the DOM, extract style information into CSS, and use JavaScript functions to encapsulate functionalities common to different scenarios, to create well-structured applications.

5. Programming Behaviorally in Blockly

5.1. Background and Rationale for the BP implementation in Blockly

The Google Blockly environment is built along principles similar to those of the popular Scratch language [40]. Other languages and environments in this family include, among others, BYOB/SNAP! [37], MIT App Inventor [2], and Waterbear [11]. 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 subclauses 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 than standard programming languages, and perhaps even more than other visual languages.

While Scratch and BYOB are interpreted (respectively, in SmallTalk and in Flash), Blockly and Waterbear diagrams are translated into code (we use JavaScript) which can later be manipulated and executed natively in any development environment.

Thus, our implementation of BP in Blockly is in fact a visual layer above the JavaScript implementation of BP. Nevertheless, towards the end of this section we discuss the synergies between BP and visual environments such as Blockly and the value of such implementations. The Blockly-based BP infrastructure generates an HTML page which contains three distinct elements. The first element contains the JavaScript code generated from the Blockly blocks specified by the application developer. The second element consists of the HTML entities specified by the developer. These enable external interfaces such as the application GUI, as explained in more detail in Section 4. Last, the HTML page contains JavaScript scripts and methods that provide the runtime infrastructure of the execution cycle of bidding, event selection, and b-thread notification and resumption. The programmer-supplied HTML entities and JavaScript scripts are entered in HTML tabs added in the Blockly editor.

5.2. BP in Blockly: Implementation Details

Our implementation includes three new (types of) blocks:

In the `b-thread` block the programmer provides the b-thread logic. The b-thread body can use any Blockly block for implementing the desired processing. The string `b-thread` in the template header can be replaced by the programmer with the b-thread's name or description. The parameter passed to the `b-thread` block is a set (i.e., a list) of values, and enables coding of symbolic b-threads. The infrastructure generates multiple instances of the given b-thread logic, and passes to each instance one of the values in the list, to be used during instance execution. The `b-Sync` block is used inside a b-thread for synchronization with other b-threads and for specifying requested, waited-for, and blocked events (it will later be translated to the `yield` command in JavaScript). The `lastEvent` block represents a variable containing the event that was triggered in the most recent synchronization point, and can be accessed by b-threads that were resumed after waiting for a number of events to determine which of these events was actually triggered.

For illustration, the application logic b-threads of the water-tap example are coded in Blockly as shown in Figure 4. With our extension to the Blockly infrastructure, these blocks are automatically compiled into the JavaScript code shown in Figure 1.

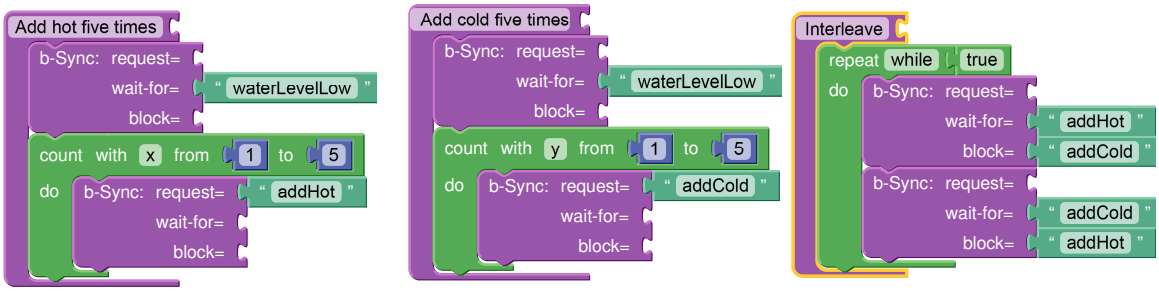


Figure 4: Blockly b-threads code for the water-tap example. The leftmost b-thread waits for `waterLevelLow` event and then requests the event `addHot` five times, the one in the center also waits for `waterLevelLow` and then requests the event `addCold` five times, and the one on the right causes the event triggering to be interleaved by repeatedly waiting for `addHot` while blocking `addCold` and vice versa.

In the Blockly implementation b-threads priorities are implied by their physical location on the development canvas: the higher the location, the higher the priority. When two b-threads are placed side-by-side and are vertically aligned, their priority order is left-to-right. This order makes a difference when there is more than one event that is requested and not blocked at a synchronization point. In this case, the event selection mechanism will select the event requested by the b-thread with the highest priority. When a b-thread requests multiple events that are not blocked, the priority is according to the order in the parameter to the `b-Sync` block which specifies the list of requested events.

In addition to the general advantages discussed in Section 10, the breaking of an application into small independent behavior modules, facilitated by the BP design pattern, is especially instrumental for visual programming languages like Blockly where the size of the screen limits the amount of visible information. Because long and complex code cannot be presented on one screen, programmers need to be able to break their applications into small independent pieces that can be understood and maintained in isolation.

The changes to the Blockly environment involved only standard Blockly customization techniques. The new blocks perform only syntactic translation from Blockly strings and structures to JavaScript. 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 BP.

5.3. Discussion of BP in Blockly

Clearly, in addition to the combined Blockly-and-JavaScript implementation shown here, BP can be used with JavaScript without Blockly, or with Blockly with translation to another language, such as Java. In this regard, Blockly is a layer above our JavaScript implementation, which can simplify development and facilitate experimenting with a variety of programming idioms. However, in addition to the desire to bring BP to yet another environment for reaching additional audiences, and to gain from the visualization afforded by Blockly for

scenario-based development with scenarios, there are synergies between Blockly (and similar visual environments) and BP. First, visual artifacts which can float on the development canvas quite independently of each other seem to be nicely aligned with the BP concepts of scenarios and events as tangible entities that provide much information about the system behavior even when standing alone. Second, the limitations imposed by screen size, which is a common issue when developing software for large systems, are partly overcome by the fact that b-threads are independent of each other. This allows the developer to have a full view of an artifact that represents and entire aspect of system behavior. Finally, we believe that the pleasing nature of visual environments, combined with the tendencies of humans to describe behavior in scenarios, both visually and in natural language, can make visual environments an attractive platform to explore and promote BP concepts.

The BP and Blockly combination is not without limitation and requires more work. For example, the limited computer screen area, while it can accommodate nicely each b-thread in its entirety, seems to be less conducive for visualizing and comprehending a large collection of independent b-thread. Developing debug tools that show the coordinated progress of all b-threads through their synchronization points should be quite straightforward. As was done in the Java library in [10], this involves mainly logging and displaying the data about all b-threads at each synchronization point. As in the Java case, such a tool can help understand desired or undesired interactions between b-threads due to blocking or priorities and explain unexpected system decisions to trigger or not trigger certain events. A more difficult issue is that of efficient state-space exploration as part of direct model checking of the executable program. Basically what is needed is a tool for copying, saving and restoring a generator at a particular state that will thus enable backtracking. In developing the model checker for behavioral Java programs this service was provided by the `Javaflow` package. Presently we are not aware of such tool for JavaScript.

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 in future research as a test-bed for investigating the naturalness of new programming idioms. For example, nesting blocks that, instead of using `bSync`, 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 `bSync`. 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. Another direction for idiom development is that of standard scenario templates, e.g., wait for event `e1` and then block event `e2` until event `e3` is triggered, or wait for events `e1` and `e2` in any order and then request event `e3`. Yet another avenue in this direction is possibly making every free-floating code snippet in Blockly into a self-proclaimed b-thread, without requiring the programmer to label it as such. Idioms of this form may make the coding more accessible to individuals who are not professional programmers.

6. Demonstration: Incremental Development

In this and in the subsequent three sections we present demonstrations of small applications that use the above Blockly and JavaScript infrastructure, each time focusing on different features and design choices. In this section we detail the development of an application for the well-known game of Tic-Tac-Toe, highlighting the incremental nature of the development, and demonstrating how the system can be built from independent modules dedicated to individual game rules or strategies. The programmer develops the application one rule or one strategy at a time, using Blockly. The development steps are similar to those in [21] where a model-checker generates the counter-examples driving the incremental development.

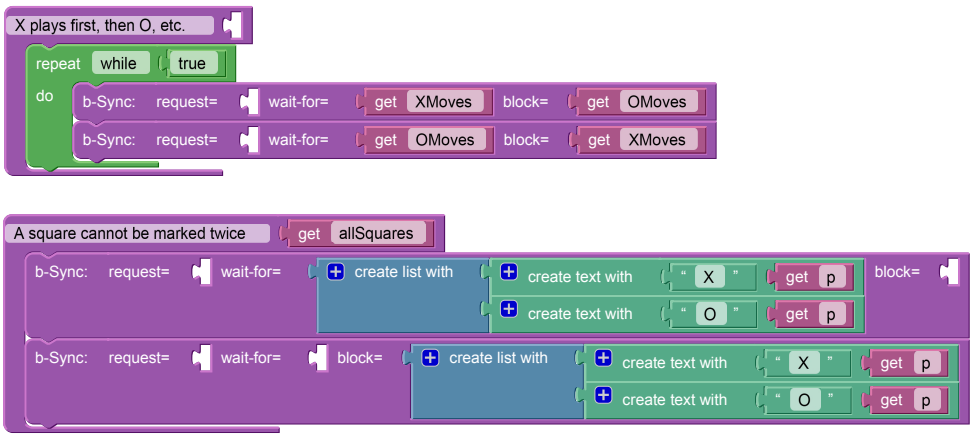
In the game of Tic-Tac-Toe, two players, X and O, alternately mark squares on a 3×3 grid whose squares are identified by $\langle row, column \rangle$ pairs: $\langle 1,1 \rangle$, $\langle 1,2 \rangle, \ldots, \langle 3,3 \rangle$. The winner is the player who manages to form a full horizontal, vertical or diagonal line with three of his/her marks. If the entire grid becomes marked but no player has formed a line, the result is a tie.

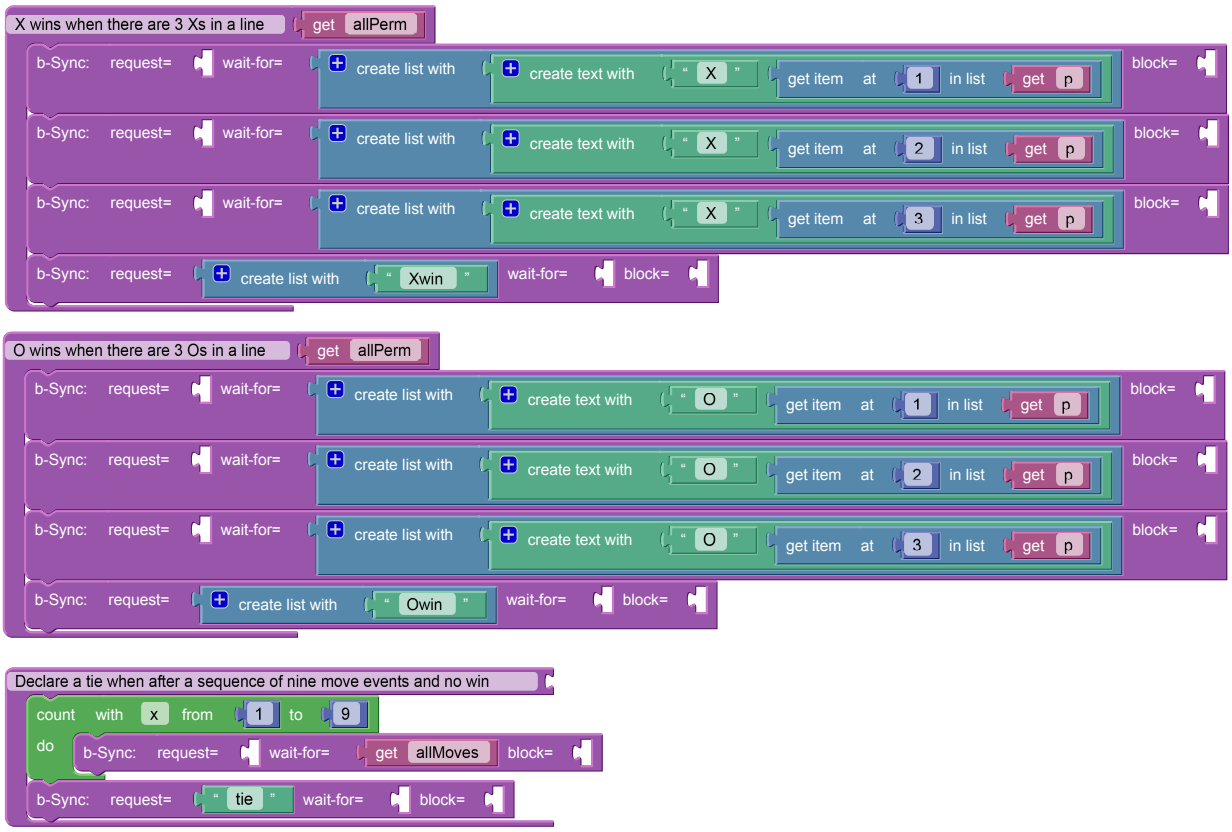
In our example, player X is played by a human user and player O is played by the application. The moves (marking of a square by a player) are represented by the events, `X11`, `X12`, ..., `X33`, for the X player and `O11`, `O12`, ..., `O33` for the O player. Three additional events, `Xwin`, `Owin`, and `tie`, represent the respective victories and a tie.

A play of the game may be described as a sequence of events. E.g., the sequence `X11`, `O22`, `X33`, `O13`, `X31`, `O21`, `X32`, `Xwin` describes a play in which X wins, and whose final configuration is:

X		O
O	O	
X	X	X

Suppose that the developer first creates the b-threads shown and explained below to enforce the rules of the game:





In this code the event sets `XMoves` and `OMoves` represent the sets of all X and all O moves respectively. The first b-thread thus enforces the rule that consecutive X moves or O moves are not allowed. The second b-thread enforces the rule that a square may only be marked once in a game. It uses the variable `allSquares` whose value is a list of length nine containing the coordinates of the squares in the game. Then, it uses the option to plug a parameter to the b-thread construct whose effect is to instantiate a b-thread for each element of the list. The variable `p` in the body of the b-thread refers to the instantiation parameter[2]. We are effectively generating nine b-threads where each b-thread waits for either an X or an O in the square `p` and then blocks further Xs and Os in the same square. The next two b-thread blocks use the variable `allPerms` containing all 48 permutations of all the eight lines (rows, columns, and diagonals) in the game to instantiate a b-thread that identifies when the X and when the O players win, respectively. This implementation of win detection with 96 b-threads is of course just a design choice, where each b-thread simply waits for three very specific events, in a particular order, and has no computation and no conditions. Alternative design choices include having only a single b-thread listen to all events and maintain a local data structure to track the game configuration, or having 8 b-threads, one for each, row, column and diagonal, each waiting for the respective three

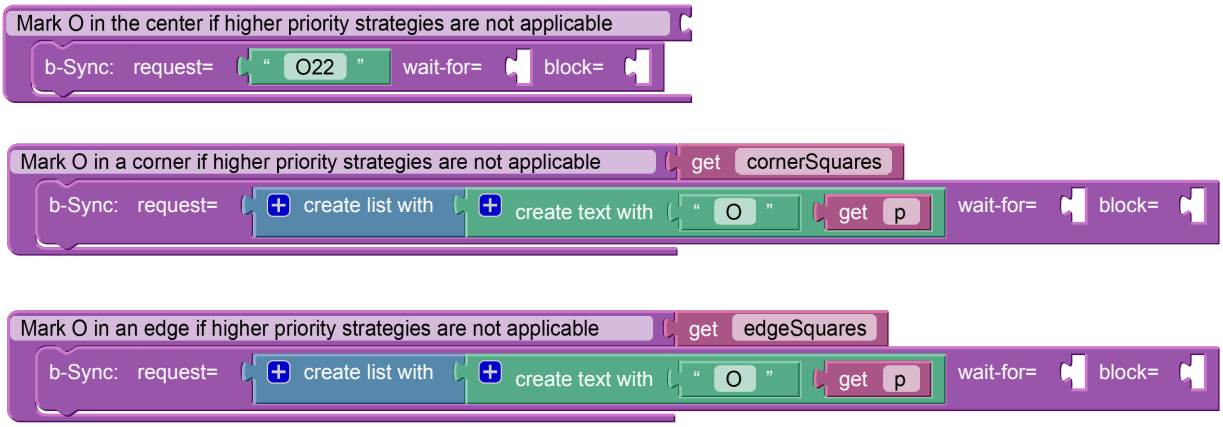
[2]In the shown version this is a standard place holder in all b-threads. In the next version of BP for Blockly presently in development a b-thread can have multiple, named instantiation parameters.

events, in any order. There are advantages to each approach. For example, our choice here highlights the fact that a b-thread can be quite unaware of the application's domain and goals, and be responsible for a very narrow task, while other choices show how a b-thread can use the full capabilities of the underlying language.

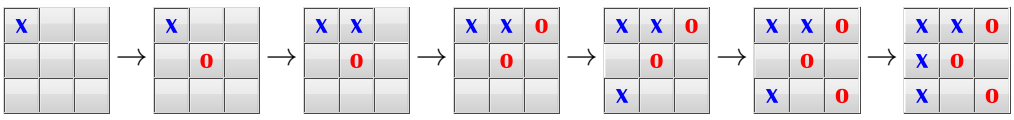
The last b-thread block instantiates a single b-thread that counts nine moves and then requests the event `tie`. It uses the variable `allMoves` whose value is an event-set containing all the 18 move events.

Note that the b-thread that detects X's win and the b-thread that detects O's win need to be given higher priority (be higher on the Blockly canvas) than the b-thread that detects a tie. Otherwise, if X wins in the ninth move, `tie` will be triggered instead of `Xwin`.

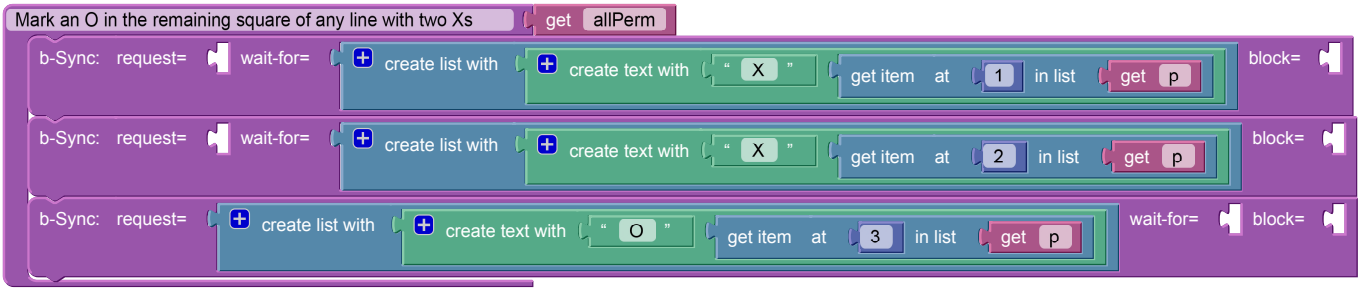
A program including only the above b-threads cannot trigger any move event since none of the b-threads ever requests any; they only wait for and/or block such events. To enable the application to really play, the developer now adds two program components that request move events: (1) a GUI component that translates each user-click on a selected square into a corresponding X move event. This component also displays the game-board to reflect the X and O move events. (2) Three b-threads that drive the default behavior of the O player by repeatedly requesting all nine possible O moves in the following priority order of center, corners and edges:



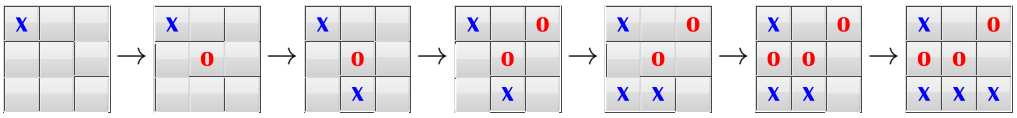
Now, we wish to gradually and incrementally enhance the program until it never loses. Suppose we experiment with the program and find out that it loses, for example, in the following 'bad' scenario:



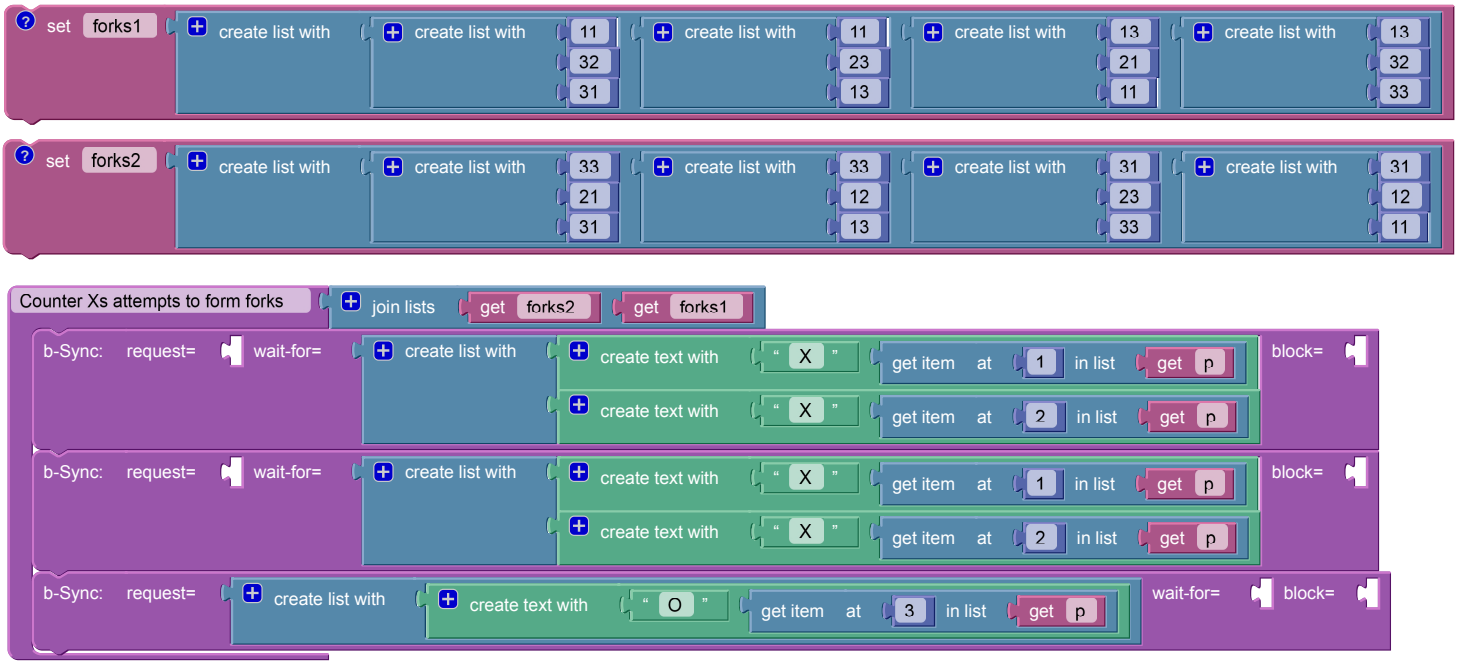
We, of course, realize that the victory of X could have been easily avoided if the application had played `O21` in its last turn, preventing the completion of three Xs in a line. An obvious resolution, therefore, is to add the following b-thread to serve as a basic tactic:



Running the game again after adding this b-thread, we get the same trace. A closer look reveals the cause: the priority assigned to the new b-thread is lower than that of the default moves (because we added it at the bottom). The default moves b-threads then prevail with the request to play 033. To overcome this problem, we move the last b-thread upwards and test again. This time we get the trace:

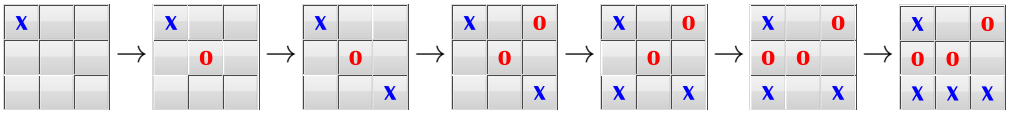


Here, the source of the problem is that once X plays his/her third move at square $\langle 3,1\rangle$, a 'fork' is created (with $\langle 1,1\rangle$ on one hand and $\langle 3,2\rangle$ on the other). If this situation is not prevented, a victory for X is inevitable. To handle the situation, we add the following variables containing the fork configurations and the corresponding b-threads:

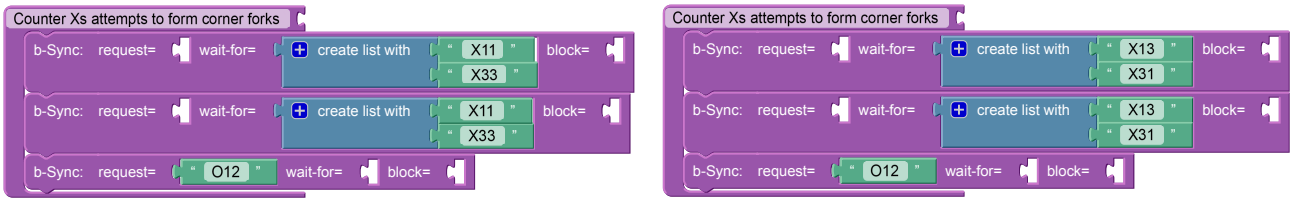


With these b-threads, when two Xs are noticed, in all configurations symmetric to the one shown in the counterexample, an O is marked in the intersection corner of the potential

fork, thus preventing its creation. This b-thread is given a priority higher than that of the default moves, but lower than that of the b-threads that puts O in a line with two Xs, as it seems to be more important to prevent an immediate loss. When we run the application again, we may find the trace:



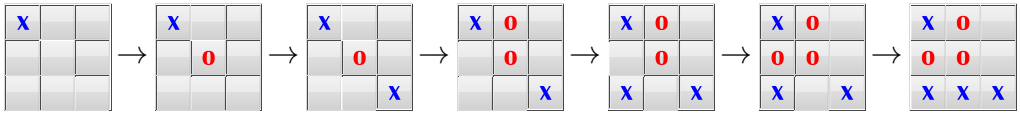
Apparently, there is another kind of a fork that should be prevented — one that consists of three corners. We thus add the b-thread:



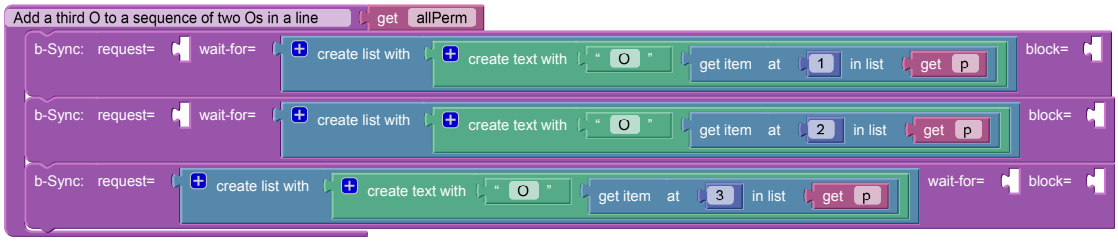
With these b-threads, when the first two Xs are marked in two opposite corners and the first O is marked at the center, O12 is requested. In the spirit of "the best defense is a good offense", this move creates an attack that forces X to play X32 and seems to avoid the immediate fork threat.

It may appear that our code includes an assumption that this strategy is needed only at the beginning of the game, and hence does not check that squares $\langle 0,1\rangle$ and $\langle 2,1\rangle$ are empty. Further testing shows that in the final program this assumption is indeed correct.

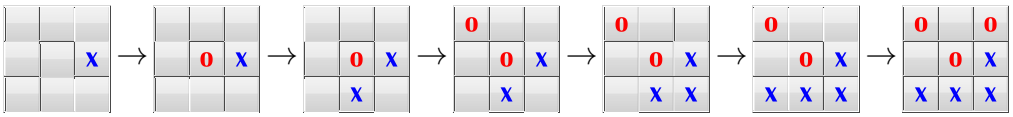
However, we still have a trace of a game where O loses:



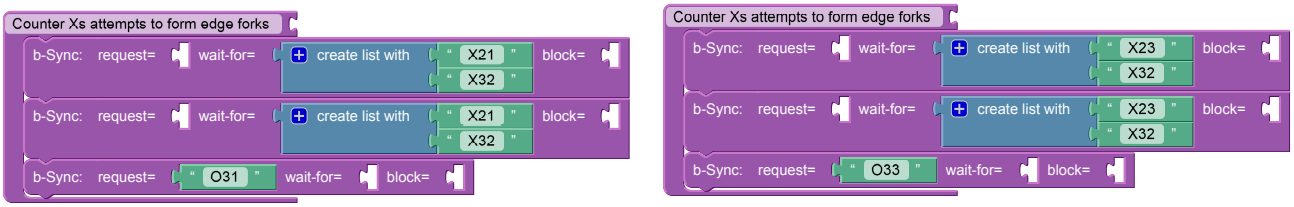
This trace reminds us that the goal of the game is to win (rather than not to lose...). It seems that while we are busy with defense, we miss the opportunity to win. Thus, we add the b-threads:



Clearly, these b-threads should get the highest priority, since winning the game is always the best move. After adding it, we may hope we are done, but a tester (or an automatic model-checker) may come up with another counterexample:



We are surprised to find that there is yet another kind of a fork to be prevented; this time, one that consists of a corner and its two adjacent edge squares. In order to prevent it, we add the b-threads:



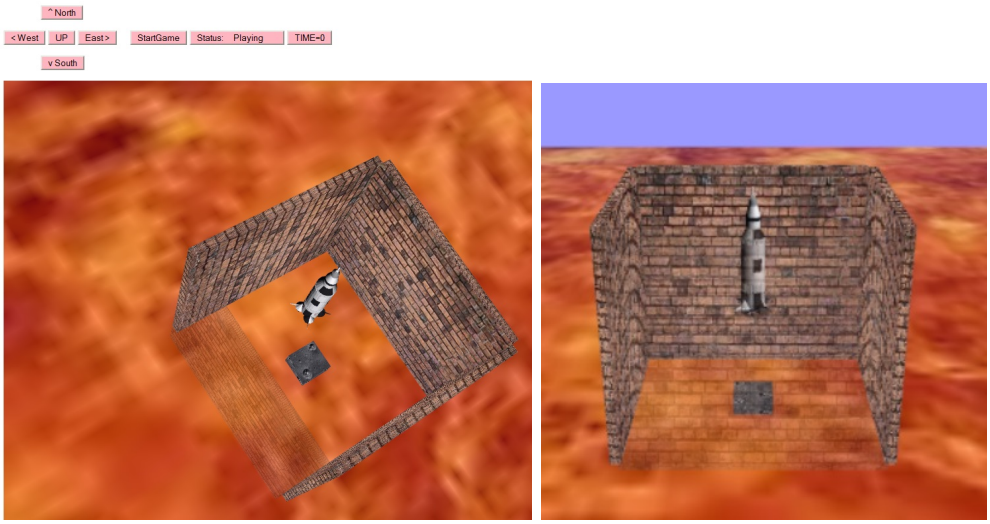
With these b-threads, when two X's are noticed in two edge squares that are adjacent to a common corner, an O mark in that corner is requested. Note that we only need to handle two such cases as we found out, by testing, that the other two cases are already taken care of.

Now, finally, a thorough testing or automatic model-checking confirms that we are done.

This Tic-Tac-Toe example demonstrates how behavioral program can be developed incrementally. When a bug is identified a b-thread is added to counter the cause of the problem. We demonstrated that a counterexample supplied by a tester or by an automatic model-checker may often be directly used in improving the solution, by treating the counterexample as a scenario, and preventing its occurrence by creating a corresponding "anti-scenario". The anti-scenario waits for all but the last system-driven event in the counterexample, and then either requests a different system event at a higher priority, or blocks the last event choice allowing other b-threads to take care of requesting the correct move.

7. Demonstration: Integration with a 3D-Graphics JavaScript library

The next example further illustrates the above concepts and also shows how a behavioral application can interface with the real world or with other applications in a way that is separated from the application logic. We examine an application for a small three dimensional computer game where the player attempts to land a rocket on a landing pad on the surface of a planet, as in the figure below.



The rocket moves downward at a fixed speed in the vertical direction. Using GUI buttons or keyboard keys, the player can move the rocket north, south, east and west with the goal of positioning it directly above the landing pad. The player can also press an `Up` button to create an exhaust burst that suspends the rocket and prevents it from going down in the next time unit. The landing pad keeps moving on the ground either randomly or subject to an unknown plan. Four walls mark the sides of the playing area, and the rocket cannot move past them (but does not crash when it touches them). The game is won when the rocket lands exactly on the landing pad, and is lost when it touches the ground without being fully on the pad. The rocket movement is in three dimensions and the view of the entire game scene can be manipulated (tilt, pan, etc.) in 3D.

In the proposed design pattern the development work can be divided into two distinct tasks performed by different individuals with different roles. A graphic designer decides on the library to be used for the 3D effects (in our case, the `three.js` library at `www.three.js`) and selects the shapes and sizes of images of the rocket, landing pad, planet surface, and walls. The second role is that of the application-logic programmer, who plans the behavior scripts, as described in the game rules (in our case they were borrowed almost unchanged from a previous, two-dimensional version of the application). As suggested in Section 2, the graphic designer and the programmer first agree on the events, or sensors and actuators, that will interface between the behavioral application and the graphics. Table 1 lists the events chosen in the present example.

Table 1: The sensors and actuators used for interfacing with the external world in the 3D rocket game application. Sensors and actuators associated with rocket at or away from north or south wall, and pad movement north and south are omitted for brevity.

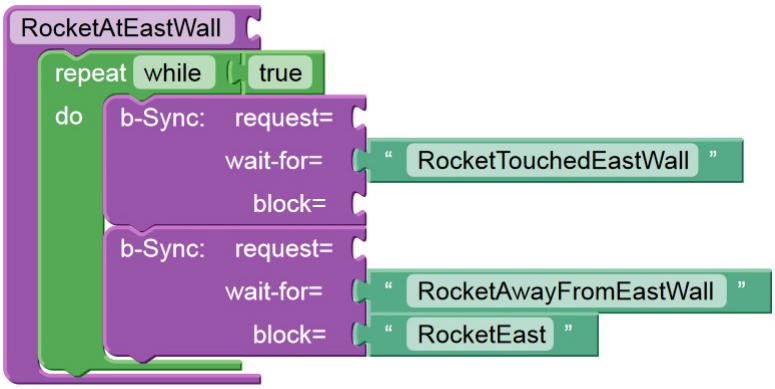
Sensor / Actuator	Event	Event Meaning (Description)
Sensor	BtnEast	User clicked `East`
Sensor	BtnWest	User clicked `West`
Sensor	BtnNorth	User clicked `North`
Sensor	BtnSouth	User clicked `South`
Sensor	BtnUp	User clicked `Up`
Sensor	TimeTick	A unit of time passed
Sensor	RocketTouchedEastWall	Rocket arrived at east wall
Sensor	RocketAwayFromEastWall	Rocket departed from east wall
Sensor	RocketTouchedWestWall	Rocket arrived at west wall
Sensor	RocketAwayFromWestWall	Rocket departed from west 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	RocketWest	Request to redraw rocket 10 pixels further to the west
Actuator	RocketEast	Request to redraw rocket 10 pixels further to the east
Actuator	RocketDown	Request to redraw rocket 10 pixels down
Actuator	PadWest	The application wishes the pad to move 10 pixels further to the west
Actuator	PadEast	The application wishes the pad to move 10 pixels further to the east
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
. . .	. . .	. . .

The graphic designer then programs the sensors and actuators. Figure 5 illustrates an actuator for the `rocketEast` event, which updates the rocket's coordinates for the graphics library interfaces (note that this event is distinct from the `BtnEast` event indicating the user's pressing of `East` button). The same script also checks if the rocket reached the east wall, and if it just departed from the west wall, and serves as the sensor that generates the associated environment events `RocketTouchedEastWall`, and `RocketAwayFromWestWall` respectively. The graphic designer could, of course, implement the sensors and actuators in many other ways. The power of the proposed separation is that these choices are transparent to the application-logic programmer.

```
//rocketEastActuator
   function rocketEastActuator()
   {
       while (true){
           yield({ wait: ['RocketEast'] });

           //Away
           if (rocketX-rocketWidth/2 <= westWall+wallThickness/2){
            bp.event('RocketAwayFromWestWall');
           }

           rocket.position.x=(rocketX+=step);

           //Touch
           if ((rocketX+rocketWidth/2) >= (eastWall-wallThickness/2)){
            bp.event('RocketTouchedEastWall');
           }
       }
   }
```

Figure 5: JavaScript code for an actuator that handles the `RocketEast` event, and also serves as a sensor that generates the events indicating if when the rocket reaches the east wall or departs from the west wall

An example of an application logic b-thread is:



Once the rocket reaches the wall as indicated by the event `RocketAtEastWall`, the b-thread blocks all `rocketEast` events (which would otherwise cause the rocket to move further east) until the rocket moves west away from the wall.

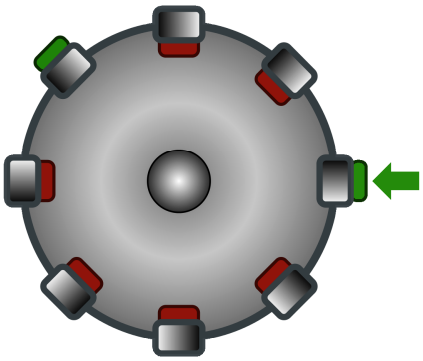
In the present implementation, the invocation of the interfaces with the 3D library can be set up as JavaScript scripts in the HTML section of the Blockly application as shown in Figure 6.

```
<script src="lib/Three/threeRev54.js"></script>
. . .
   var mytime=0;
   var tickdelay = 2000;
   var rocketX=0,          rocketY=0,     rocketZ=110;
   var rocketWidth=40,        rocketHeight=10;
. . .
   init();           // initialization
. . .
   function init()
   {
      scene = new THREE.Scene();
      var SCREEN_WIDTH = window.innerWidth, SCREEN_HEIGHT = window.innerHeight;
      var VIEW_ANGLE = 45, ASPECT = SCREEN_WIDTH / SCREEN_HEIGHT, NEAR = 0.1, FAR = 20000;
      camera = new THREE.PerspectiveCamera( VIEW_ANGLE, ASPECT, NEAR, FAR);
      scene.add(camera);
      camera.position.set(0,-800,400)
. . .
```

Figure 6: Interfacing with the three.js library from the blockly application. The code is entered in the HTML tab which include the JavaScript code needed to activate the external library and establish the necessary functions, callbacks and variables that will be used by the rest of the application.

8. Demonstration: Orchestrating Animations

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. Consider for example, the problem of where the graphic design involves animation segments which have to be orchestrated both in sequence and in parallel. We demonstrate handling this issue in the context of a puzzle application based on a combinatorial game [42] where the human player and a computerized adversary push switches in and out on a rotating wheel with a goal of reaching (or avoiding) a particular configuration.



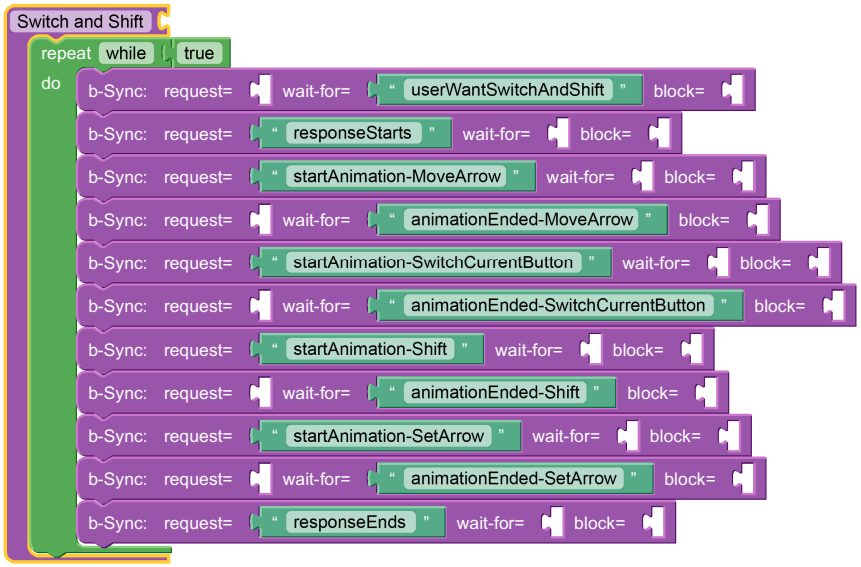
A game move consists of optionally pressing the switch (invoked when the `Switch` event is triggered), and then rotating the wheel to the next switch position (invoked when the `Shift` event is triggered).

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 [4], where the integration with the external library details follow the same design as in the rocket game example in Section 7. 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 using `yield`, 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:

```
UserWantsSwitchAndShift = function() {
   state = "SwitchandShift0";
   ResponseStart();
}

ResponseStart = function() {
   if( state = "SwitchAndShift0") {
      state = "SwitchAndShift1";
      startAnimation_MoveArrow();
   } else {
      ...
   }
}

AnimationEnded_MoveArrow = function() {
   if( state = "SwitchAndShift1") {
      state = "SwitchAndShift2";
      startAnimation_SwitchCurrentButton();
   } else {
      ...
   }
}

...
```

In our implementation this sequence is handled naturally in a b-thread as consecutive instructions:



Thus, BP facilitates waiting for events in-line and not only via callbacks. As mentioned before, several JavaScript preprocessors 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 event blocking or multiple b-threads.

9. Demonstration: An Infrastructure for Smartphone Customization

In this section we describe additional infrastructure for running behavioral programs coded in Blockly and JavaScript on Android smartphones, and demonstrate basic scenarios as well as sensors and actuators interfaces to smartphone functions. The infrastructure, shown in Figure 7, consists of two main parts: (a) a Web site where users can create b-threads using Blockly, translate them to JavaScript and send them to a smartphone for execution, and (b) an Android application that encapsulates an execution environment for user-provided b-threads.

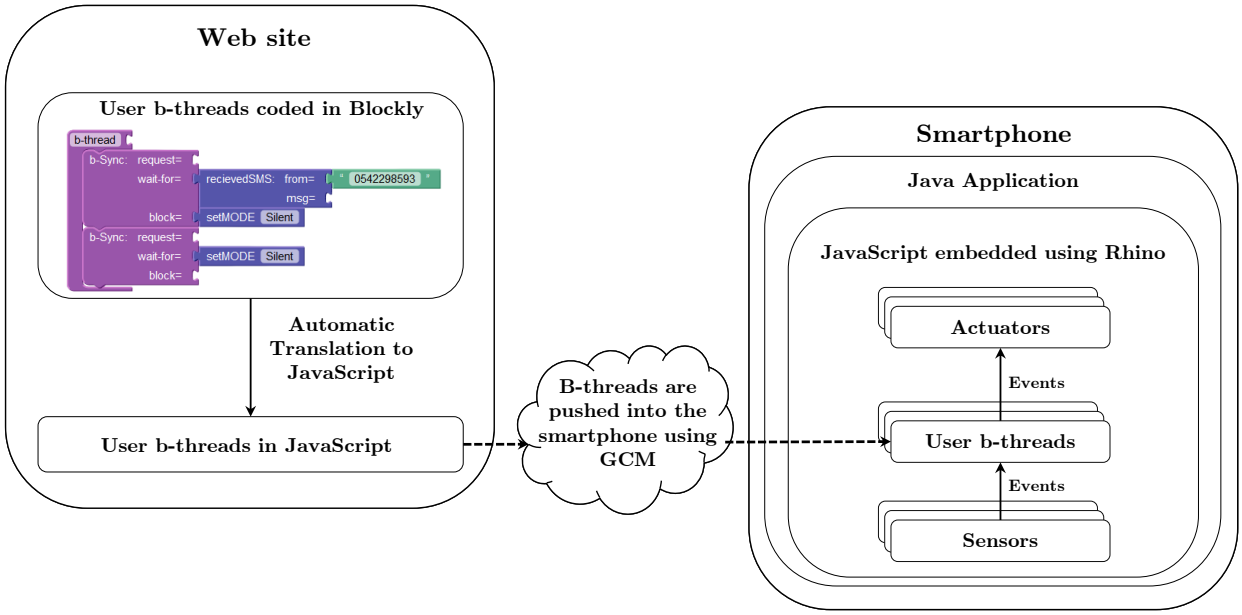


Figure 7: Blockly and JavaScript infrastructure for Android. The User creates b-threads using Blockly on the Web site. The b-threads are translated to JavaScript and the resulting JavaScript code is then pushed to the smartphone using Google Cloud Messaging (GCM). On the phone, the b-threads are executed and interwoven in a JavaScript environment created under a Java application using Rhino. The JavaScript actuators and sensors in turn use Rhino to invoke phone functions and to install callbacks for intercepting phone events using the Android Java API.

To allow the b-threads and BP execution mechanisms written in JavaScript to run in the Android Java environment, and to provide them with direct access to the relevant APIs of the device functionality we use the Rhino package [7]. Specifically, Rhino allows exporting Java objects to JavaScript code and vise versa. Using this facility, one can call Android API functions from JavaScript code and install callbacks written in JavaScript that will be invoked when events of interest happen in the device. For more advanced interfaces and functionality one can also integrate the infrastructure with the PhoneGap library (`www.phonegap.com`).

The sensors and actuators are precoded in the infrastructure as follows. The sensors are implemented as callbacks that are installed to be called when Android events occur. They then inject corresponding behavioral events to the BP infrastructure using `bp.event`. In the next super-step, b-threads that wait for the event will be awakened and will be able to react to it. For example, the following JavaScript code is a sensor for handling the arrival of text messages (SMS):

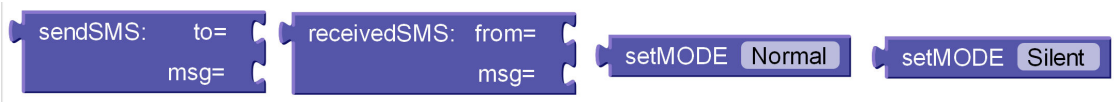
```
/**
* Receive SMS - Sensor
*/
activity.on("ReceiveSMS", function(context, intent) {
```

```
   /* Extract data from the Android intent parameter */
   msgSrc = getMessageSource(intent);
   msgBody = getMessageBody(intent);

   /* Insert the event to the behavioral program */
   bp.event('receivedSMS:from:'+msgSrc+':msg:'+msgBody);
});
```

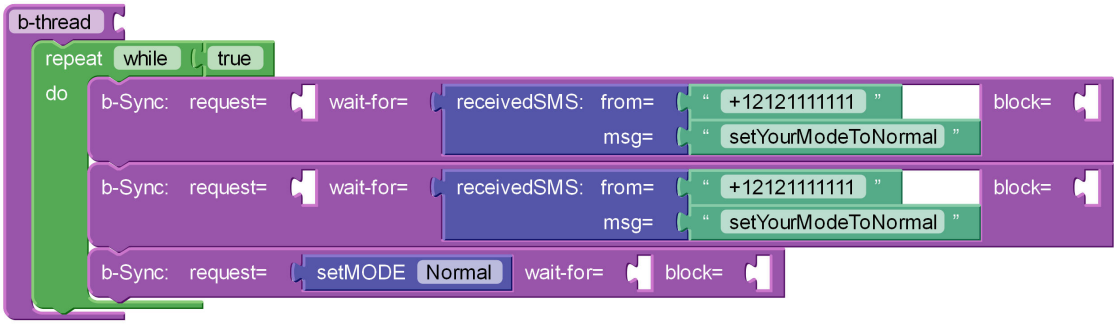
Actuators are implemented as infrastructure-provided b-threads that wait for behavioral events and generate effects on the environment using Android API. For example we define the behavioral event `setMode` to mean a request to change the ringer mode of the phone. The Blockly specification of these events are translated into JavaScript strings with parameters such as `normal`, `silent`, or `vibrate`. The actuator below is a b-thread that waits for all `setMode` events, parses the string and calls an Android-specific interface (`audioDevice`) to translate the occurrence of the behavioral event to the setting of the ringer mode in the physical Android device.

```
/**
* Set Ringer Mode - Actuator
*/
bp.addBThread('setMODE',priority++, function() {
   while(true) {
      /* Wait for all behavioral setMode events */
      yield({ wait:[function (e) {
            /* Return true if the event string starts with 'setMode' */
            return (e.indexOf("setMODE") === 0);
         }]);

      /* Extract the data from the event */
      var mode = bp.lastEvent.split(":")[1].toUpperCase();

      /* Use Android API to create the side-effect */
      audiodevice.SetRingerMode(mode);
   }
});
```

To facilitate the use of the sensors and actuators, we added to the Blockly pallet events such as `receivedSMS`, `sendSMS`, and `setMode` shown below:



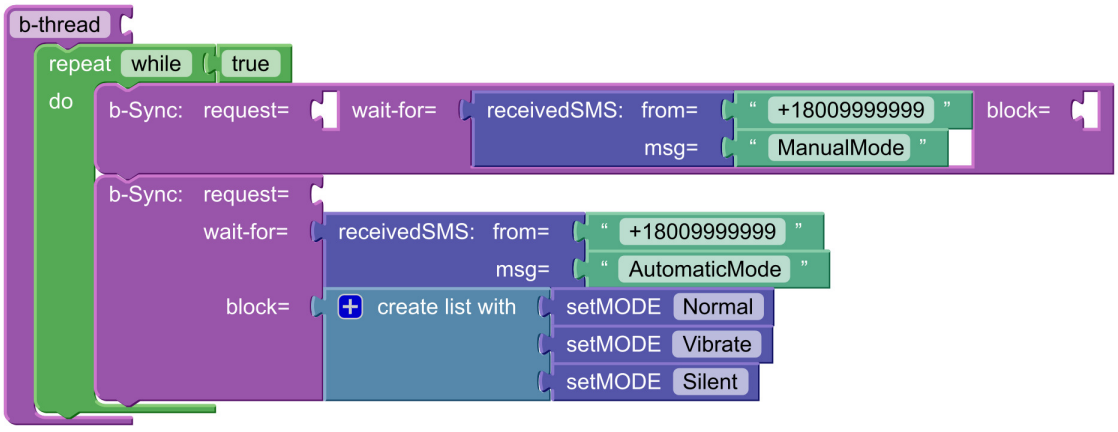
We now turn to a simple example of coding and interweaving scenarios. Consider a user who wishes to change the ringer mode of the smartphone in response to SMS messages. The user first writes a b-thread to automatically change the ringer mode to normal if he or she receives two text messages "setYourModeToNormal" from a given number:



These Blockly blocks are translated to the JavaScript code:

```
bp.addBThread('b-thread', priority++, function() {
   while (true) {
      yield({wait:["receivedSMS:from:+12121111111:msg:setYourModeToNormal"]});

      yield({wait:["receivedSMS:from:+12121111111:msg:setYourModeToNormal"]});

      yield({request:["setMODE:Normal"]});
   }
});
```

After having programmed several such scenarios, the user realizes that in some rare situations these automatic changes are not desired, say, when a message "ManualMode" is received from another number. The user could, of course, change all existing b-threads, but with BP it is possible to just add the following b-thread that waits for a text message `ManualMode` from a designated number and blocks all `setMode` events until an `AutomaticMode` message is received from that number:



These Blockly blocks are translated to:

```
bp.addBThread('b-thread', priority++, function() {
   while (true) {
      yield({wait:["receivedSMS:from:+18009999999:msg:ManualMode"]});

      yield({wait:["receivedSMS:from:+18009999999:msg:AutomaticMode"],
             block:["setMODE:Normal","setMODE:Vibrate","setMODE:Silent"]});
   }
});
```

This brief demonstration shows how multiple scenarios that interact with the smartphone can be developed incrementally and be interwoven at run time.

The approach that we have described in this section is similar in nature to the architecture of on{X} (`www.onx.ms`), a tool by Microsoft that allows users to specify rules via a graphical interface on a website, translate them to JavaScript, and push them to an Android phone where a dedicated application runs them and gives them access to the phone capabilities via a rich API. The popularity that on{X} gained shows that users need such abilities to better tailor their phones to their needs. The small demonstration that we have described above suggests that addition of event blocking and of an ability to describe scenarios, in addition to short if-then rules, may provide good ways for users to interweave multiple needs in an incremental manner where each behavioral aspect is specified separately.

10. Discussion: BP Benefits and Development Contexts where they Emerge

In this section we summarize the above demonstrations by outlining some of the advantages and desirable capabilities of behavioral programming techniques, and software development scenarios in which they appear. We conclude the section with a discussion of some limitations.

10.1. 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 seen in the Tic-Tac-Toe example in Section 6 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. For another example consider the ease with which one would add additional obstacles in the rocket-landing game in Section 7, or the ability to demonstrate to an end-user or a customer the running application at very early stages, e.g., only with rocket movements, or only with landing-pad movement, or with both, but without walls and other obstacles. It will also be easy to allow 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 ones).

The independence of behavior threads is also manifested in that scripts and scenarios do not have to communicate with each other directly. In native Blockly or Scratch, without BP additions, broadcasting and publish-subscribe techniques already allow rich processing relying only on local variables and avoiding global or shared variables. The addition of behavioral synchronization and event-blocking enriches the integrated runs without adding a burden of peer-to-peer communication. Specifically, events that a b-thread blocks or waits for may be generated by any existing or yet-to-be-developed b-thread or sensor, and the system can be run at many intermediate development stages with meaningful results.

10.2. Natural state management for long scenarios

As demonstrated in detail in the nullification-game example in Section 8, programming behaviorally allows for managing state in a natural way by relying on instruction-progression in independent b-threads, as compared, say, to updating and examining state variables in callback routines.

Another illustration of the naturalness of state management in BP is in the Tic-Tac-Toe game application in Section 6. In the design chosen here, the data structure of the board is used only for the graphics display, and b-threads do not examine it at run time to understand the evolving state of the application or to drive the strategy. Instead, each b-thread instance is responsible for a particular sequence of events — simply waiting for certain two events to occur, and then requesting another event. Thus, each b-thread becomes simpler and easier to debug by hand or with automation tools, such as model-checkers.

10.3. 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 BP 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 [31, 13], this design appears to be "natural" in the sense that it was adopted by children who were not explicitly guided to use it. An attractive design pattern which emerges when such capabilities are available, is to have a main process with only basic conditions and branches, and a large collection of smaller exception handlers, dealing "all the time" with special cases and new requirements as they evolve over time.

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 [5] 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 Scratch 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

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 never incremented. We did not find documentation of this semantics of Scratch.

Our approach, in this paper, is to view scenarios as global entities with well-defined semantics (see Section 2) for scheduling, synchronization, and interweaving. Using our Blockly and JavaScript environment, an application similar to the above example will have to be coded differently. If the BP programmer codes two b-threads, with one instance of each, to perform a function similar to the above event broadcasting and processing, the event associated with the second message causes no effect, as at the synchronization point when it is triggered the processing b-thread will not be waiting for it, but instead will be waiting for a behavioral event signifying the completion of the wait time. If, on the other hand, the application starts another instance of the second b-thread class to catch such messages while other instances wait for the time-delay to pass, the event will be processed. In either case, the semantics will be well defined and the composite behavior will be readily predictable.

Since the above implementation is single-threaded, standard concerns about race conditions between parallel processes are alleviated. However, even in a multi-threaded or multi-process BP environment (e.g. BP in Java or in Erlang) the well-defined synchronization, publish-subscribe and event selection semantics, allows designers of BP applications

to rely solely on behavioral events and avoid variable sharing, and thus further assure the avoidance of such race conditions.

10.4. Priority as a first-class idiom

When multiple simultaneous behaviors are actively trying to affect the progress of a common application and each presents its event declarations independently of the others, a priority scheme that can be specified externally to the behaviors themselves becomes a useful construct. This is seen for example in the Tic-Tac-Toe application where moves that produce an immediate winning for player O are of higher priority than defensive moves, and where certain default moves (such as marking the center or corners squares) are preferred, using priority, over other moves (marking edge squares). In BP, the priority of each b-thread is specified when it is added, which is a natural and commonly-used scheme. In addition, the algorithm in Figure 2 specifies a sequential execution order for b-threads, that refines the behavioral semantics of BP.

10.5. Integration with standard programming

Coding behaviorally does not mean that all calculations and data processing performed by the application must be based on events. This was clearly exemplified in the nullification and rocket-landing game examples where much of the processing was done not behaviorally. This is usually the case with domain specific elements of the application, whose complexity does not come from the interweaving of multiple behaviors. This however does not preclude decomposing even such application-specific processing into independent simultaneous behaviors. For example, in [22], a single intertwined system of linear equations controlling the thrust and balancing of an aerial vehicle was replaced with four independent b-threads where one controls only the thrust, and each of the remaining three independently controls only one of the attitude angles - pitch, roll and yaw.

10.6. Direct description of cross-cutting scenarios

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 [15], in the designer's mind, or at least in their description of behaviors, scenarios are not necessarily anchored on a given object. For example, the scenario of a complete telephone call can be readily described independently as a stand-alone scenario involving two human parties, two telephone devices and multiple infrastructure facilities, devices and controls. The common engineering approach of describing the same scenario as emerging from the independent full behavior specification of each participating entity obviously works, but appears more complex and less natural.

The Blockly/JavaScript environment presented here does not force the programmer to associate desired behaviors with behaving objects. For example, in the Tic-Tac-Toe application, the game rules such as turn alternation or prevention of multiple marks in the same square are not anchored on any of the players and not even on an invisible controller, and

in the nullification game example the complex sequence of switching and wheel rotation, both logically and visually is just that — a scenario, and is not anchored on the wheel, the switches, the players, or a game controller.

In [31] the researchers observed that when forced to associate scripts with sprites, young programmers spread the scripts of the (correct) behavior of one game character across 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.

We thus propose that there is a distinction between objects in general, as in object oriented programming, and the concept of behaving entities that are tangible in the user's eyes. It may be preferable from points of view of system structure or naturalness of development to not require anchoring code on perceived behaving entities and, instead, code scenarios that cut across multiple such entities as standalone modules in their own right. Of course, the scenarios, events, screen objects, etc., may themselves be coded with object-oriented programming.

10.7. Limitations

There can also be cases where designing applications with BP is not attractive and other design approaches may be considered. See for example the discussion in [24] in this journal special issue. For example, one may be concerned that an application developed incrementally with BP may become a loosely coupled collection of scenarios representing base code, bug corrections and handling of exceptions and of other new requirements discovered at different times. In answering this concern we observe that first, indeed, in the maintenance of many legacy applications, such patches are introduced in various techniques, and the straightforward way by which such patching can be done in BP may be useful. Second, we note that it is the role of the developers to properly plan their activities and design the application. When a new requirement or bug are nevertheless introduced, they may need to decide whether to code new scenarios, modify an existing ones, or refactor a whole set of scenarios. In the latter case, one can think of the refactoring as implementing new insights into how to best decompose the systems integrated behavior into loosely coupled scenarios.

Another concern is that the incrementality may not be applicable whenever desired. Examples include cases where an existing b-thread unconditionally blocks an event that a new b-thread wishes to trigger, or where a new b-thread blocks an event requested by an existing b-thread causing the existing b-thread to stall forever, where the real intent was to have it only abandon the request and continue. Idioms for handling these situations can be readily added to BP in the future, but we do not feel compelled to do so yet. Instead, we believe that resolving such issues by rewriting b-threads, while keeping them aligned with the original requirements, may help reach a better understanding of those requirement. Further, retaining full and pure incrementality for every new, refined, or changed requirement, may not always be desired in the first place.

11. Comparison of BP to other techniques

As stated above, certain aspects of BP appear in various forms in other programming techniques and paradigms such as publish-subscribe and aspect orientation. Further, event-

based programming with synchronized simultaneous behaviors and with concise capability for event blocking can be implemented in many ways and with many techniques. Detailed comparisons to other techniques and languages appear in [20, 18]. Briefly, standard publish-subscribe does not include idioms for synchronization of simultaneous behaviors or for event-blocking, and, while standard aspect-oriented programming and rule-based systems do offer synchronization and some form of blocking, state management in the advice or action modules must use variables and cannot rely only on progression of instructions. Regarding rule-based systems which allow blocking and disabling of rules such as in [12], we note that by contrast, in BP blocking is targeted at events, and does not require specification of the entity that generates the event.

BP shares some goals with Functional Reactive Programming (FRP). Of particular interest in the present context is the JavaScript-based Flapjax [32]. In both Flapjax and BP there is a focus on reactive specifications, i.e., that external events trigger internal ones which may in turn trigger other internal events and so on, until an external effect is generated. Also, in both, a key goal is to enable building a system from small modules aligned with the user's perception of overall system behavior while retaining the usage of the full power of a standard programming language (JavaScript). An interesting difference is that Flapjax aims at functional programming while BP is more procedural. Specifically, in BP, b-threads are multistep scenarios that return values in each successive synchronization point while in Flapjax event handling is done by functions that compute and return a single value in each invocation. Hence the progression of inter-object scenarios may not be directly visible in Flapjax code. Flapjax does not offer explicit event blocking. Clearly the two approaches can be combined in a variety of ways. For example, b-threads can process events coming from Flapjax event-streams and can implement new sources for such streams, in support of interweaving of simultaneous scenarios.

The coordination and event selection of BP can be seen as a variation on the blackboard metaphor, and tuple-space model. Indeed, Shimony et al [41] used a tuple-space model in the PicOS environment using the C language, as an underlying infrastructure for implementing BP principles and the idioms of requesting, waiting-for and blocking events.

The paper [20] includes a brief discussion of positioning BP relative to formalisms, languages and environments designed specifically to express concurrency, such as *communicating sequential processes* (CSP) [26], the *calculus of communicating systems* (CCS) [33], the *π-calculus* [34], the programming languages Erlang [3], Esterel [6], Lustre [14], Signal [27], Orc [28], and UNITY [35]. The main difference is that BP idioms and the b-threads entities are meant to describe system behavior, and do not directly deal with interprocess communication and concurrency. As stated in [20], we have not found constructs whose execution semantics can be readily mapped to that of BP's request/wait/block, yet it is clear that BP can be readily implemented in these environments. In fact, the paper [24] in this special issue suggests such an implementation in a distributed environment using message passing protocols, with an example in Erlang.

12. Conclusion 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 BP.

Possible next steps include exploring the scalability of the concepts in applications such as complex robotics and large biological models, and studying BP programming idioms as suggested in Section 5.3.

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 ongoing 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. We thank Guy Weiss for his support in developing the 3D Rocket game. We thank Barak Inbal and Uriel Pinker for the development of the BP infrastructure for Android.

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, by an Advanced Research Grant to David Harel from the European Research Council (ERC) under the European Community's FP7 Programme, and by the Israeli Science Foundation. The research of Gera Weiss and Adiel Ashrov was supported by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University, by a reintegration (IRG) grant under the European Community's FP7 Programme, and by the Israeli Science Foundation.

References

[1] ECMAScript Standard, 2013. http://www.ecmascript.org.

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

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

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

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

[6] G. Berry. The foundations of Esterel. In *Proof, Language, and Interaction*, pages 425–454, 2000.

[7] N. Boyd et al. Rhino: Javascript for java, 2007. URL `http://www.mozilla.org/rhino`.

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

[9] N. Eitan and D. Harel. Adaptive behavioral programming. *IEEE Int. Conf. on Tools with Artificial Intelligence*, 2011.
[10] 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.
[11] D. Elza. Waterbear language web site. URL: http://waterbearlang.com/, accessed Aug. 2012.
[12] 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.
[13] 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.
[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data-Flow Programming Language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, Sep. 1991.
[15] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
[16] 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.
[17] D. Harel and I. Segall. Synthesis from live sequence chart specifications. *Jour. Computer System Sciences*, 78:3:970–980, 2012.
[18] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7): 90–100.
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] D. Harel, A. Kantor, G. Katz, A. Marron, G. Weiss, and G. Wiener. Towards behavioral programming in distributed architectures. *Science of Computer Programming*, 2014.
[25] T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using giotto. *Control Systems Magazine, IEEE*, 2003.
[26] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.
[27] B. Houssais. *The synchronous prog. language SIGNAL, a tutorial*. IRISA, 2002.
[28] D. Kitchin, A. Quark, W. Cook, and J. Misra. The Orc Programming Language. In *FMOODS/FORTE*, pages 1–25, 2009.
[29] D. E. Knuth. *Fundemental Algorithms*, volume 1 of *The Art of Computer Programming*, section 1.4.2, pages 193–200. Addison-Wesley, 3rd edition, 1997.
[30] H. Kugler, C. Plock, and A. Roberts. Synthesizing biological theories. In *CAV*, 2011.
[31] 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.
[32] 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.
[33] R. Milner. *A Calculus of Communicating Systems*. LNCS vol. 92. Springer, 1980. ISBN 3-540-10235-3.
[34] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Inf. Comput.*, 100(1):1–40, 1992.

[35] J. Misra. A foundation of parallel programming. *International Summer School on Constructive Methods in Computer Science*, pages 397–433, 1988.
[36] N. Mix. Narrativejs. URL: http://www.neilmix.com/narrativejs/, accessed Aug. 2012.
[37] J. Moenig and B. Harvey. BYOB Build your own blocks (a/k/a SNAP!). URL: http://byob.berkeley.edu/, accessed Aug. 2012.
[38] Mozilla Foundation. FireFox JavaScript 1.7 -. URL: http://developer.mozilla.org/en/New_in_JavaScript_1.7, accessed Aug. 2012.
[39] Oni Labs. Stratifiedjs. URL: http://onilabs.com/stratifiedjs/, accessed Aug. 2012.
[40] 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.
[41] B. Shimony, I. Nikolaidis, P. Gburzynski, and E. Stroulia. On coordination tools in the PicOS tuples system. *SESENA*, 2011.
[42] 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.