DOI:10.1145/2209249.2209270

A novel paradigm for programming reactive systems centered on naturally specified modular behavior.

BY DAVID HAREL, ASSAF MARRON, AND GERA WEISS

Behavioral Programming

SPELLING OUT THE requirements for a software system under development is not an easy task, and translating captured requirements into correct operational software can be even harder. Many technologies (languages, modeling tools, programming paradigms) and methodologies (agile, test-driven, model-driven) were designed, among other things, to help address these challenges. One widely accepted practice is to formalize requirements in the form of use cases and scenarios. Our work extends this approach into using scenarios for actual programming. Specifically, we propose scenario-coding techniques and design approaches for constructing reactive systems[28] incrementally from their expected behaviors.

The work on behavioral programming began with scenario-based programming, a way to create executable specifications of reactive systems, introduced through the language of *live sequence charts* (LSC) and its *Play-Engine* implementation.[11,21] The initial purpose was to enable testing and refining specifications and prototypes, and it was later extended toward building actual systems. To this end, the underlying behavioral principles have also been implemented in Java via the BPJ package[25] and in additional environments,[26,34,42,43] adding a programming point of view to that of requirement specification.

To illustrate the naturalness of constructing systems by composing behaviors, consider how children may be taught, step-by-step, to play strategy games (See Gordon et al.[14]). For example, in teaching the game of Tic-Tac-Toe, we first describe rules of the game, such as:

`EnforceTurns`: To play, one player marks a square in a 3 by 3 grid with `X`, then the other player marks a square with `O`, then `X` plays again, and so on;

`SquareTaken`: Once a square is marked, it cannot be marked again;

`DetectXWin/DetectOWin`: When a player places three of his or her marks in a horizontal, vertical, or diagonal line, the player wins;

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

`AddThirdO`: After placing two `O`s in a line, the `O` player should try to mark the third square (to win the game);

`PreventThirdX`: After the `X` player marks two squares in a line, the `O` player should try to mark the third square (to foil the attack); and

`DefaultOMoves`: When other tactics are not applicable, player `O` should prefer the center square, then the cor-

» key insights

- **Behavioral programming is a novel, language-independent paradigm for programming reactive systems, centered on natural and incremental specification of behavior, and implemented in the visual formalism of live sequence charts (LSC), and in the BPJ Java package.**
- **The approach allows coding applications as multi-modal scenarios, each corresponding to an individual requirement, specifying what can, must, or may not happen following certain sequences of events.**
- **To facilitate full behavioral modularity via the independent coding of separate facets of behavior, all scenarios run simultaneously, and all are consulted at every decision point during execution.**
- **The paradigm is supported by tools for debugging, execution planning, learning-based adaptivity, and model-checking for early detection of conflicting and incomplete requirements.**

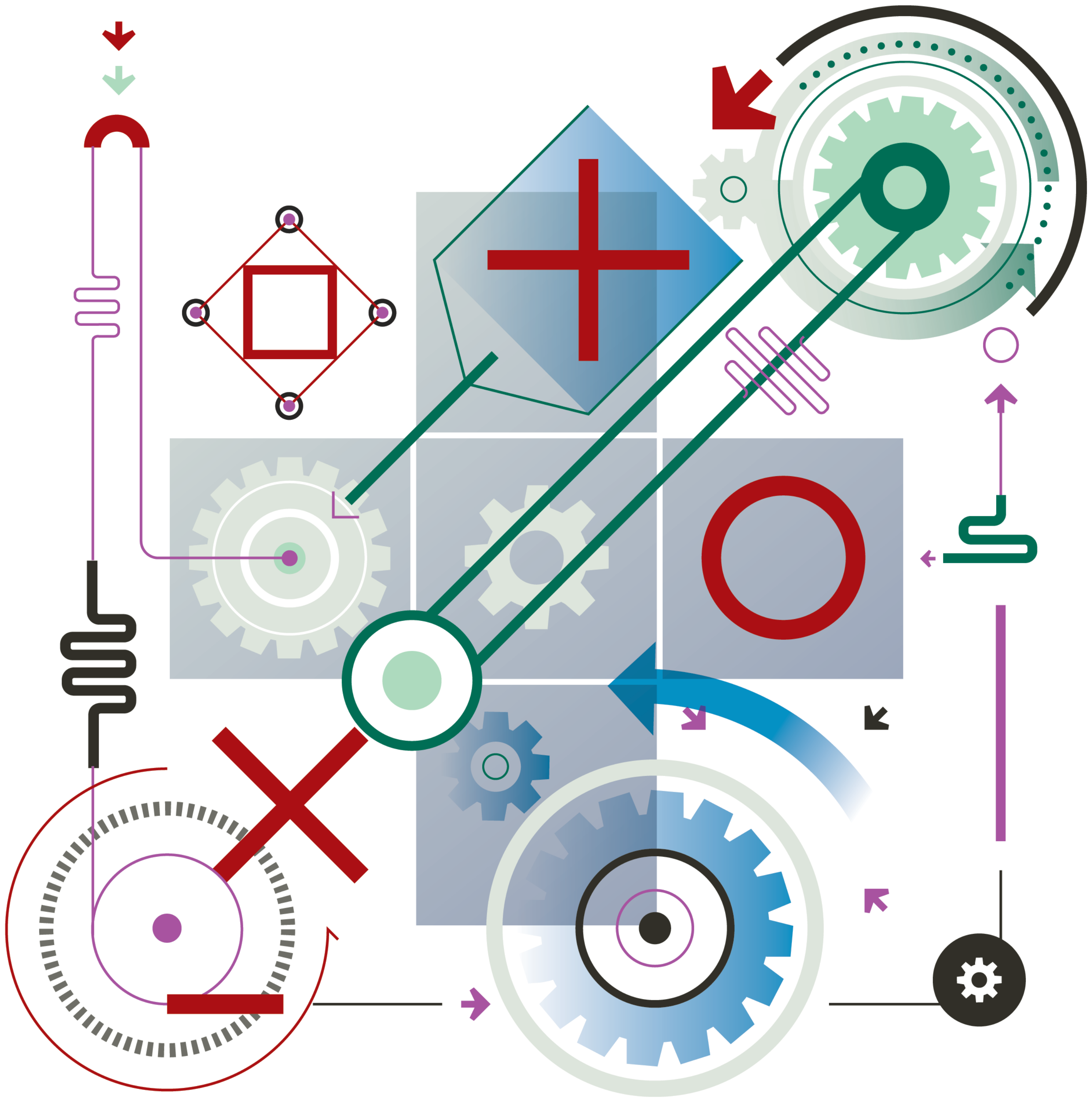


ILLUSTRATION BY LEANDRO CASTELAO

ners, and mark an edge square only when there is no other choice.

Such required behaviors can be coded in executable software modules using behavioral programming idioms and infrastructure, as we will detail. Full behavioral implementations of the game exist in Java[25] and Erlang.[43] In Harel et al.[20] we show how model-checking technologies allow discovery of unhandled scenarios, enabling the user to incrementally develop behaviors for new tactics (and forgotten rules) until a software system is achieved that plays legally and assures the computer never loses.

This example already suggests the following advantages of behavioral programming. First, we were able to code the application incrementally in modules that are aligned with the requirements (game rules and tactics), as perceived by users and programmers. Second, we added new tactics and rules (and still more can be added) without changing, or even looking at, existing code. Third, the resulting product is modular, in that tactics and rules can be flexibly added and removed to create versions with different functionalities, for example, to play at different expertise levels.

Naturally, composing behaviors that were programmed without direct consideration of mutual dependencies raises questions about conflicting requirements, underspecification, and synchronization. We deal with these issues by using composition operators that allow both adding and forbidding behaviors, analysis tools such as model checkers, and architectures for large-scale applications.

In this article, we present the principles of behavioral programming and illustrate how to program behavioral applications in Java. We detail visual behavioral programming with the LSC language and elaborate on how one deals with conflicting behaviors, underspecification, and a large number of simultaneous behaviors. We conclude with a comparison to other development approaches, applications, and future research.

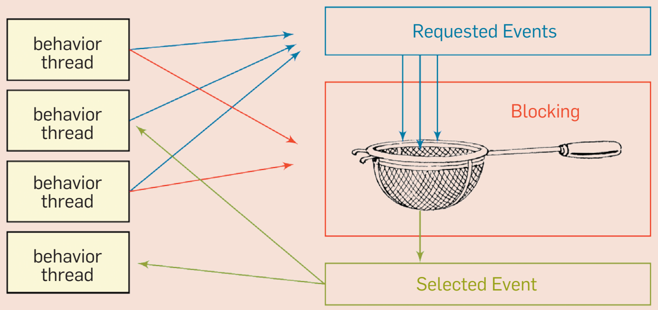
Basic Behavioral Idioms

We propose the term *behavioral application* for software consisting of independent components (called *b-threads*) that generate a flow of events via an enhanced publish/subscribe protocol, as follows (see Figure 1). Each b-thread is a procedure that runs in parallel to the other b-threads. When a b-thread reaches a point that requires synchronization, it waits until all other b-threads reach synchronization points in their own flow. At synchronization points, each b-thread specifies three sets of events: *requested events*: the thread proposes that these be considered for triggering, and asks to be notified when any of them occurs; *waited-for events*: the thread does not request these, but asks to be notified when any of them is triggered; and *blocked events*: the thread currently forbids triggering any of these events.

When all b-threads are at a synchronization point, an event is chosen, that is requested by at least one b-thread and is not blocked by any b-thread. The selected event is then triggered by resuming all the b-threads that either requested it or are waiting for it. Each of these resumed b-threads then proceeds with its execution, all the way to its next synchronization point, where it again presents new sets of requested, waited-for and blocked events. The other b-threads remain at their last synchronization points, oblivious to the triggered event, until an event is selected that they have requested or are waiting for. When all b-threads are again at a synchronization point, the event selection process repeats. For a formal definition of this process see Harel et al.[25,26]

When more than one event is requested and not blocked, the semantics of event selection may vary. For example, the selection may be arbitrary or random, as in the default (a.k.a. *naïve*) semantics of the LSC Play-Engine;[21]

Figure 1. A schematic view of the execution cycle of behavior threads using an enhanced publish/subscribe protocol.



1. All behavior threads synchronize and place their "bids":
   - **Requesting an event**: proposing that the event be considered for triggering, and asking to be notified when it is triggered;
   - **Waiting for an event**: without proposing its triggering, asking to be notified when the event is triggered;
   - **Blocking an event**: forbidding the triggering of the event, vetoing requests of other behavior threads.
2. An event that is requested and not blocked is selected;
3. The behavior threads that requested or wait for the selected event are notified;
4. The notified behavior threads progress to their next states, where they place new bids.

Figure 2. B-threads for increasing water flow. The first two b-threads request the events `addHot` and `addCold` three times, respectively. The third b-thread, `Interleave`, repeatedly waits for `addHot` while blocking `addCold` and vice versa, forcing alternation of these events. Without `Interleave`, the run would be three `addHot` followed by three `addCold`, due to b-thread priorities.

```
class AddHotThreeTimes extends BThread {
    public void runBThread() {
        for (int i = 1; i <= 3; i++) {
            bp.bSync( addHot, none, none );
        }
    }
}

class AddColdThreeTimes extends BThread {
    public void runBThread() {
        for (int i = 1; i <= 3; i++) {
            bp.bSync( addCold, none, none );
        }
    }
}

class Interleave extends BThread {
    public void runBThread() {
        while (true) {
            bp.bSync( none, addHot, addCold );
            bp.bSync( none, addCold, addHot );
        }
    }
}
```

Event log of the coordinated run

```
addHot
  addCold
addHot
  addCold
addHot
  addCold
```

choices may depend on some priority order, as in standard BPJ execution; the mechanism may use look-ahead subject to desired properties of the resulting event sequence, as in *smart play-out*[19,29] in LSC; it may vary over time, based on learning;[13] or, as in Kugler et al.,[34] the entire execution may diverge into multiple concurrent paths.

The programming idioms of *request*, *wait for*, *block* thus express *multimodality*. Reminiscent of modal verbs in a natural language (such as *shall*, *can* or *mustn't*), they state not only what *must* be done (and how) as in standard programming, but also what may be done, and, more uniquely to behavioral programming, what is *forbidden* and therefore *must* not be done.

Behavioral programming principles can be readily implemented as part of different languages and programming approaches, with possible variations of idioms. In addition to Java with the BPJ package[25] (discussed later in more detail) we have implemented them in the functional language Erlang[26,43] and Shimony et al.[42] applied them in the PicOS environment using C. Implementations in visual contexts beyond the original Play-Engine include PlayGo[23] and SBT by Kugler et al.[34]

In behavioral programming, all one must do in order to start developing and experimenting with scenarios that will later constitute the final system, is to determine the common set of events that are relevant to these scenarios. While this still requires contemplation, it is often easier to answer the question "what are the events?" than "which are the objects/functions, etc.?" By default, events are opaque entities carrying nothing but their name, but they may be extended with rich data and functionality.

Programming Behaviors In Java

Our implementation of behavioral programming in Java uses the **BPJ package**.[25] With BPJ, each behavior thread is an instance of the class `BThread`. Events are instances of the class `Event` or classes that extend it (mainly for adding data to events). The logic of each behavior is coded as a method supplied by the programmer, which in turn invokes the method `bSync` to synchronize with other behaviors, and to specify its requested, waited-for and blocked events as follows:

```
bSync(requestedEvents,
      waitedForEvents,
      blockedEvents);
```

By calling `bSync` the b-thread suspends itself until all other b-threads are at a synchronization point and is resumed when an event that it requested or waited for is selected, as described below.

To enforce predictable and repeatable execution, we require that the event selected at each synchronization point be uniquely defined. To this end, the programmer assigns a unique priority to each b-thread, and places the requested events of each b-thread in an ordered set. The event selection mechanism in BPJ then uses this ordering to choose the first event that is requested and not blocked.

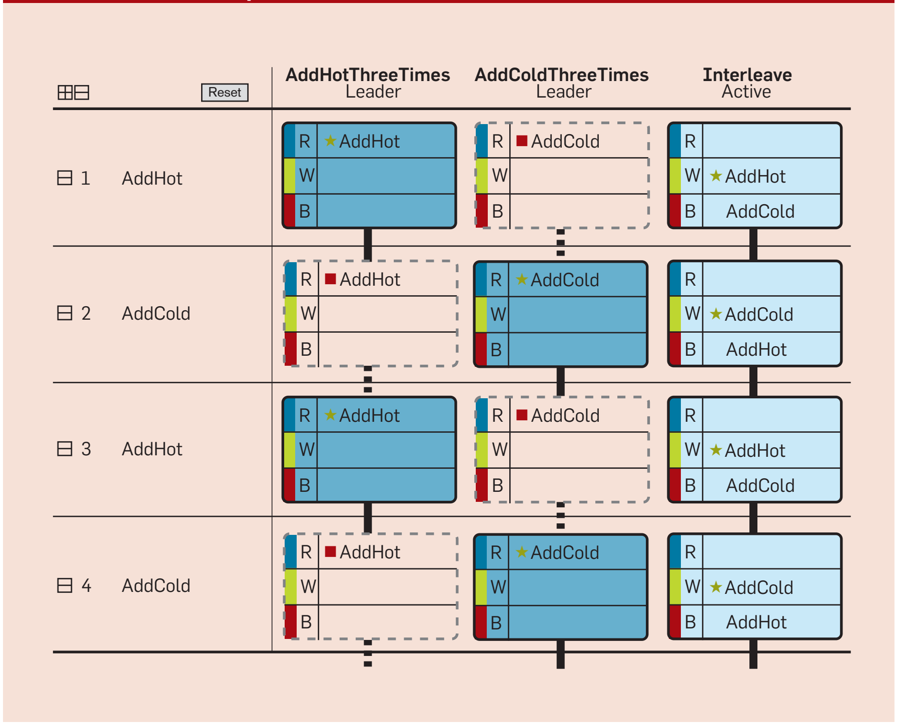
The source code package of BPJ is available online at http://www.b-prog.org with examples and video demonstrations.

Example: Water flow control. To illustrate how these constructs can be used to allow new behaviors to non-intrusively affect existing ones, consider scenarios that are part of a system that controls hot and cold water taps, whose output flows are mixed.

As shown in Figure 2, let `AddHotThreeTimes` be a b-thread that requests three times the event of opening the hot water-tap some small amount (`addHot`), and then stops. The b-thread `AddColdThreeTimes` performs a similar action on the cold water tap (with the event `addCold`). To increase water flow in both taps more or less at the same time, as may be desired for keeping the temperature stable, we activate the above b-threads alongside a third one, `Interleave`, which forces the alternation of their events. `Interleave` repeatedly waits for `addHot` while blocking `addCold`, followed by waiting for `addCold` while blocking `addHot`. Later, we illustrate a similar program written in the visual LSC language.

Behavioral execution can be further analyzed with table-like visuals, as in Figure 3, which was generated by the TraceVis trace-comprehension and debugging tool.[12] Briefly, b-threads are depicted in columns ordered by priority, and successive synchronization points and associated triggered events appear in rows intersecting the b-thread columns. Each table cell describes a b-thread's state at a given synchronization point. The sets of

Figure 3. Visualizing an execution of the water-tap application with TraceVis. Selected events are marked with a green star; blocked events are marked with a red square; cells marked R/W/B show requested, waited for, and blocked events.

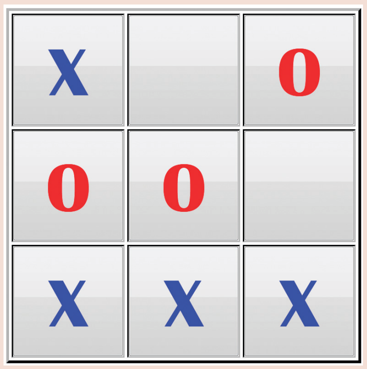


requested, waited-for, and blocked events are shown in sub-cells marked R, W, and B respectively. In each row, all appearances of the selected event are marked with a green star, and requested events that are blocked are marked by red squares, providing insight into the rationale of event selection and b-thread progression. The cell containing the request that drove the event triggering is emphasized with a bold border, and cells of b-threads that did not advance are marked by a dashed border.

Example: Strategies for Tic-Tac-Toe. Behavioral programming supports incremental development, where new behaviors may be added non-intrusively, that is, with little or no change to existing code. We demonstrate this trait with an application for playing the game of Tic-Tac-Toe, described in detail in Harel et al.[25] As outlined in the introduction, players `X` (a human) and `O` (the computer) alternately mark squares on a grid of 3 rows by 3 columns, each attempting to place three of her marks in a full horizontal, vertical or diagonal line. Each marking of a square labeled ⟨`row`, `col`⟩ is represented by a move event, `X`⟨`row`,`col`⟩ or `O`⟨`row`,`col`⟩. The events `XWin`, `OWin` and `Draw` mark possible conclusions of the game.

A play of the game can be described as a sequence of events. For example, the sequence `X`⟨0,0⟩, `O`⟨1,1⟩, `X`⟨2,1⟩, `O`⟨0,2⟩, `X`⟨2,0⟩, `O`⟨1,0⟩, `X`⟨2,2⟩, `XWin`, describes a play in which `X` wins, and whose final configuration is depicted in Figure 4.

Figure 4. The Tic-Tac-Toe gameboard configuration following the move events `X`⟨0,0⟩, `O`⟨1,1⟩, `X`⟨2,1⟩, `O`⟨0,2⟩, `X`⟨2,0⟩, `O`⟨1,0⟩, `X`⟨2,2⟩.



We describe the incremental development of all the b-thread classes needed for the rules and tactics in Harel et al.[25] Here, we describe the flow of some of the b-threads to illustrate how the natural language descriptions in the beginning of the article, can be translated to code which includes calls to `bSync`. The b-thread for the game-rule `SquareTaken`, for example, first calls `bSync` to wait for any `X` or `O` event and then calls `bSync` again to block all events in the newly marked square. As another example, the b-thread `DefaultOMoves` uses a Java loop to repeatedly request (by calling `bSync`) the set of all nine possible `O` moves ordered with center first, then corners, and then edge squares. An example of a longer scenario is `AddThirdO`, which waits for an `O` event, then waits for another `O` event in the same line, and then requests an `O` event marking the third square in the line.

To demonstrate incremental development, consider how when we learn that our defense behaviors are insufficient against a corner-center-corner attack (for example, `X`⟨0,0⟩, `O`⟨1,1⟩, `X`⟨2,2⟩) for which the only defense is a counterattack, we can add a b-thread as follows. To foil `X`'s plan, the new b-thread waits for this sequence of events (and equivalent ones), and attacks back by requesting the move `O`⟨0,1⟩. Later, we discuss how this development approach can be enhanced using a model-checking tool.

B-threads may autonomously watch out for very specific sequences of events embedded in larger traces, with expressiveness that goes beyond responding to a single event or to a combination of conditions, as is common in basic rule engines. Moreover, in our experience, a given "world configuration" or a complete event sequence may be assigned different meanings by different behaviors as they individually work toward different goals. For example, `DetectXWin` and `PreventThirdX` can independently observe the same two `X` moves in the same line, but while the former then waits for another `X` move toward announcing a win, the latter proceeds to make an `O` move in the third square to prevent a loss. In fact, most of our Tic-Tac-Toe b-threads do not check the game configuration; for example, a b-thread `DetectDraw` counts any nine moves and declares the end of the game with no winner, and `PreventThirdX` ignores `O` moves before requesting its own desired move.

Focusing on a narrow facet of a behavior can simplify the b-thread and can be accomplished by instantiating copies of it with different parameters. For example, we implemented `SquareTaken` with an instance for each square, and `DetectXWin` with an instance for each permutation of three `X` events in each line.

The autonomy afforded by a narrow world view is facilitated also by the fact that all b-threads that request a given event at a particular synchronization point are notified when it occurs, and are unaware of whether the selected request was theirs or came from another b-thread. For example, a single marking of an `O` in a particular square could result from simultaneous requests by the `AddThirdO`, `PreventThirdX`, and `DefaultOMoves` b-threads. Using blocking and priorities, autonomous b-threads can "carve out" undesired behaviors of other b-threads, as, say, with coding `DefaultOmoves` to repeatedly ask for the same set of events without checking which one was triggered, and then adding the b-thread `SquareTaken`.

Example: Real-time aircraft stabilization. Given the principles described so far, one may ask how behavioral programs deal with external events, such as physical ones originating in the environment, or user actions. This section briefly introduces elements that can serve in a layer above the behavioral programming infrastructure for development of real-time systems. See Harel et al.[26] for more details.

Behavioral applications can detect external events at any time, using all the features available in the host language, and can introduce them as behavioral events in the next synchronization point. For the integration of behavioral and non-behavioral parts of an application, we adopt the following scheme, based on the concept of *super-steps*, which is similar to the timing semantics of Statecharts.[27]

The first super-step begins when the system starts. Then, internal b-thread-driven events are triggered until there are no more such events to trigger. At this point the behavioral system halts, all b-threads are inside a bSync method call, and the system is waiting for an external event. When an external event occurs and introduced as a behavioral one, it marks the beginning of a new super-step, which then continues until there are no events to trigger, and so on. We propose a convention, whereby external events are not introduced as behavioral events as long as there are other internal events to trigger. In LSC, this is enforced by the Play-Engine and PlayGo tools. In BPJ, the programmer can assign to a b-thread that introduces external events a priority lower than that of any b-thread that may request other (internal) events at the same time. One may view the super-step as an ordered sequence of events, which ideally takes "zero time," as in Berry's synchrony hypothesis[4] and in Statecharts,[27] and similar to hybrid time sets and logical execution time (LET) design.[31]

We now outline parts of the software for controlling a quadrotor, an aircraft lifted and propelled by four fixed rotors, as detailed in Harel et al.[26] One of the challenges in stabilizing a quadrotor is using a fixed set of controls, namely the rotors' speed (RPM), to balance competing goals like desired forces and moments along different axes: flight direction, roll (side-to-side), pitch (raising and lowering the front), and yaw (rotation of the entire quadrotor). These goals compete with each other as changes in any rotor speed may affect multiple forces. For example, changing the back rotor's RPM affects the thrust, the pitch and the yaw. Behavioral programming allows decomposing the application into b-threads, each of which takes care of only one force. For example, "when thrust is too low, request the increase of at least one of the rotors' RPM and block the decrease of all rotors' RPM" or "when pitch angle is too high, request the increase of the back rotor's RPM or the decrease of the front rotor's RPM while blocking the increase of the back rotor's RPM and the decrease of the front rotor's RPM." Note that the event selection mechanism will weave these two behavior threads, in such a way that when the thrust is too low and the pitch is too high, only the back rotor's RPM will increase, addressing both deviations. To fix deviations of different sizes, many small RPM-change events occur before new input of desired forces is obtained in the next super-step. The actual b-threads are more involved than those shown here, but they maintain their naturalness and independence. For a fuzzy-logic based approach to implementing control with behavioral programming, see Harel et al.[24]

Live Sequence Charts

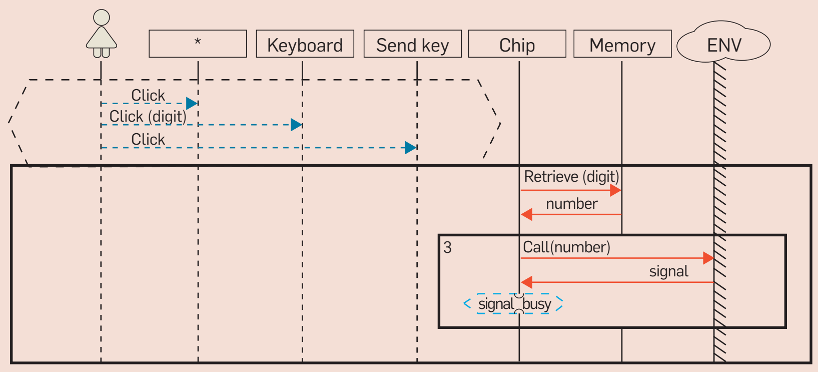
The visual language of *live sequence charts* (LSC) introduced scenario-based programming, and implicitly also the basic concepts of behavioral programming; see Damm and Harel.[11] One continuation of that work was the invention of the *play-in* and *play-out* techniques for constructing and executing LSCs, which were implemented in the *Play-Engine* tool.[21] A more recent tool, *PlayGo*, has been developed, and is currently being extended and strengthened.[23] The LSC approach also inspired the SBT tool.[34] While the current status of these tools does not yet enable broad usage in real-world applications, the versatility of the LSC language has been demonstrated in various application domains, including hardware, telecommunication, production control, tactical simulators, and biological modeling.[2,10,41]

LSC adds liveness and execution semantics to behaviors described using *message sequence charts* (MSC) by extending MSC with modalities, symbolic instances, and more. An MSC depicts behavior using vertical lifelines to represent objects and horizontal arrows for messages passed between them, with time flowing from top to bottom. This yields a partial order for occurrences of the events in a chart. However, the expressive power of MSC is limited,[11] as these charts describe possible scenarios and cannot specify, for example, what is mandated or what is not allowed. In fact, given a set of objects and events, a system that generates all possible sequences of events would satisfy any MSC.

To address this, in a live sequence chart one can distinguish what must happen (called *hot* in LSC terminology, and colored red) from what may happen (termed *cold*, and colored blue), and can also express what is not allowed to happen (*forbidden*). Moreover, event specifications that are to be executed in a proactive manner can be distinguished from ones that specify monitoring; that is, merely tracking the event. LSC also distinguishes between universal charts, which depict executions that are to apply to all runs, and existential charts—which serve as "examples" and are required to apply only to at least one run. A universal LSC consists of a prechart and a main chart, as in Figure 5. The semantics is that if and when the behavior described by the prechart occurs, the behavior described by the main chart must occur too.

Using a designated chart area, one can forbid occurrence of events at

Figure 5. A universal LSC. Whenever a telephone user presses the sequence of a star, a digit, and send (see hexagonal prechart), the chip must retrieve the corresponding number from memory and call it by sending a message to the environment. If a busy signal is returned, the call must be tried up to three times. The events in the main chart may occur only in the order specified.



certain times. There are other ways to forbid things from occurring; one of which is done by indicating that events in the main chart must occur only in the specified (partial) order. For example, one can specify that when the main chart is active, events that appear in the chart but are not presently enabled cannot be triggered at that point in time by other charts.

A modest view of LSC considers it to be a requirements and specification language for making assertions about sequences of events. In this view, a system satisfies an LSC specification if all its runs satisfy all the universal charts in the specification, and for each existential chart, there is at least one run that satisfies it.

However, the play-out technique facilitates the execution of an LSC specification, that is, a collection of charts, just like any computer program. Play-out does this by tracking events that may be selected next in all lifelines in all charts, selecting and triggering events subject to the must/may/forbidden modalities, and advancing affected charts accordingly.[21] As described in more detail below, play-out may be viewed as interpreting charts with modal events as threads of behavior with their requested, waited-for, and blocked events.

A dialect of LSC has been designed to be compliant with UML 2.0,[22] and can be defined as a profile therein. Instead of precharts it uses solid and dashed arrows to indicate whether an event is to be executed or is only monitored, while the red and blue color retain their respective modalities of must and may. The PlayGo tool[18,23] is currently based on this version of the language. Figure 6 depicts a PlayGo example similar to the water-tap application discussed earlier, with the addition of the user pressing a start button to activate all scenarios.

Internally, the LSC play-out mechanism uses the *request / wait / block* idioms for collective execution, as follows. Initially, the next enabled event for each lifeline in each chart is the topmost event in the lifeline. All enabled events on all lifelines are considered waited-for. All enabled events that are also to be executed (and not just monitored) are considered also as requested. All events that are forbidden, either explicitly or implicitly, are considered blocked. An event that is requested and not blocked is then triggered. When no event can be triggered, the system waits for an event from the user or the environment. When an event is triggered, a rich *unification* algorithm determines which event specifications in different charts refer to that event, and all lifelines in which it is enabled are advanced to their next state. Whenever this advancing causes a prechart to be completed, the main chart portion of the chart is activated. When a forbidden event nevertheless occurs, for example, as driven by the environment or the user, a violation occurs and the execution terminates.

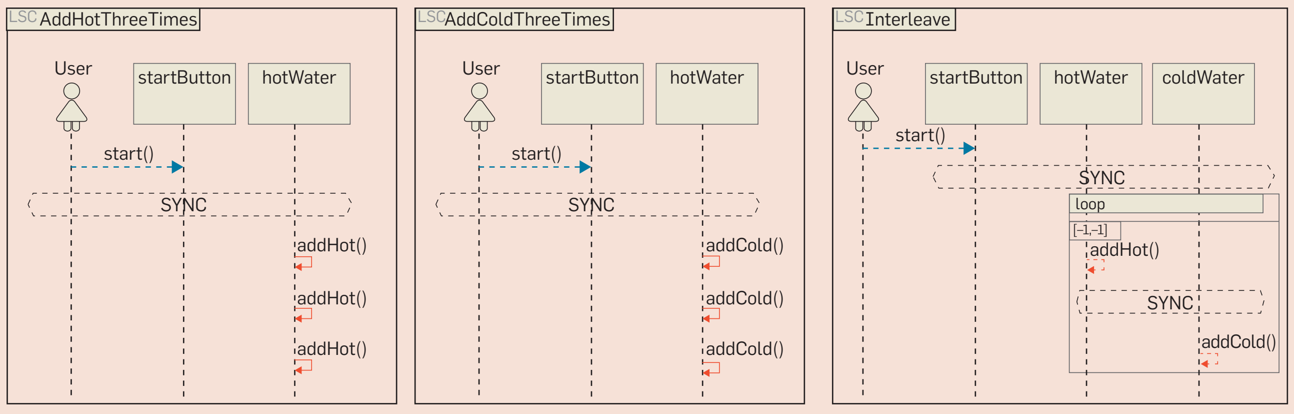
This process is often referred to as *naïve play-out*. In a more advanced mechanism, called *smart play-out*[19,29] the Play-Engine uses either model-checking or AI planning algorithms to look ahead, in an attempt to select events in ways that do not eventually lead to violation of the specification or deadlock.

In addition to the interpreter-like approach of play-out, a compiler for LSC has been developed, which produces executable code by compiling the specification into Java and weaving the results with AspectJ.[38]

One notable difference between LSC and the BPJ package is that BPJ benefits from the power of the Java host language. By contrast, the LSC language provides its own constructs for objects and properties, flow of control, exceptions, variables, symbolic objects and messages, a notion of time, sub-chart scope, access to functions in other languages, and external communication.[21]

Play-in. An essential element of programming is the process by which programmers perform actual coding. In behavioral programming, it seems only natural to allow this activity to include walking through a scenario, generating events and sequences thereof, and using them in specifying what we want done or forbidden. Toward that purpose, the LSC language allows a new way of coding, called *play-in*,[15,21] which captures scenarios as follows: whenever possible, the developer actually performs the event—

Figure 6. UML-compliant LSCs. Each chart begins with the user pressing the start button. Two charts request tap-turning events, and the third causes their interleaving by alternately waiting for these events. Events can occur only when enabled for triggering in all charts in which they appear. The SYNC construct forces order between events in different lifelines.



for example, by pressing "send" on a telephone—and the tool captures the event and includes it as part of the gradually generated LSC. The reader is referred to Harel and Marelly[21] and the Web site http://www.wisdom.weizmann.ac.il/~playbook for more details.

Play-in is similar to programming by example[37] in that both try to make programming easier for humans using visualization and physical actions, and the approaches can certainly gain from one another. The main difference is that programming by example is a way to avoid writing code in small programs, for educational purposes, where play-in is intended for use as part of programming modal scenarios to be executed collectively as a complex system.

Play-in is similar to programming by example in that both try to make programming easier for humans using visualization and physical actions, and the approaches can certainly gain from one another.

Can It Work In The Real World?

In way of trying to tackle such questions as "can the approach deal with conflicts and underspecification?" or "can one coordinate thousands of simultaneous behaviors?" we outline some relevant research results.

One concern associated with aligning application scenarios with requirements is that individually valid requirements may conflict. Thus, coding them independently of each other and composing them without consideration may yield undesired joint behavior.

We first observe that, as described, our approach suggests resolving conflicts using new b-threads and priorities. For example, in Tic-Tac-Toe, the conflict (which may emerge very early in development) concerning both players requesting a move at the same time, is resolved by a b-thread that enforces turn alternation. Similarly, a conflict between a defensive move and a move that yields an immediate win is resolved by prioritizing the latter.

Further, we present a methodology and a supporting model-checking tool (called BPmc) for verifying behavioral programs without having to first translate them into a specific input language for the model checker.[20] Our method facilitates early discovery of conflicting or underspecified scenarios. For example, when model-checking a behavioral Tic-Tac-Toe application, the counterexample `X⟨0,0⟩`, `O⟨1,1⟩`, `X⟨0,1⟩`, `O⟨0,2⟩`, `X⟨2,0⟩`, `O⟨2,2⟩`, `X⟨1,0⟩` suggests (as described earlier) that the victory of `X` could have been avoided had the application played `O⟨1,0⟩` in its last turn, preventing the completion of three `X` marks in a line, instead of its default preference to mark corners. Note that in coding refinements and corrections, counterexamples provided by the tool can be used directly since they are sequences of events. See Harel et al.[17] for an application of this property to the automatic local repair of behavioral programs.

From the model-checking perspective, the BPmc tool (which currently applies to our Java implementation of behavioral programming) reduces the size of the state-space of a Java program using an abstraction that focuses on the behaviorally interesting states and treats transitions between them as atomic. To the existing standard execution control, which consists of deterministic progression along a single path in the behavioral program state graph, we add two model-checking execution modes: safety and liveness. The safety mode explores the different paths in the graph in search of a state that violates the given safety property, while the liveness mode seeks cycles that violate the given liveness property. The graph traversal in BPmc is carried out with established model checking algorithms and uses the Apache `javaflow` package to save and restore continuations—objects that hold the states of participating threads—for the required backtracking.

Synthesis techniques have also been applied to LSC, in order to check for conflicts and, when possible, to generate a program that correctly implements a system complying with the specification.[30,35]

Model checking and planning algorithms are used when running LSCs to help avoid conflicts when these can be resolved via "smart" event selection using look-ahead within a super-step.[19,29] Future research directions include applying BPmc to achieve look-ahead in Java execution too, as well as going beyond a single super-step in the smart play-out method in LSC.

As for underspecification and adaptability, it is well accepted in software engineering that a requirements document is never really complete,[16] and that new requirements keep emerging as developers and users learn about and experiment with the developed system.

Similarly to the case of conflicts, new requirements in behavioral programming can often be coded as new behaviors. For example, while developing the quadrotor application we realized that rotor speed (RPM) cannot be negative. We solved this by adding a b-thread that blocks speed reduction events when the speed is too low.

Obviously, both model checking and the look-ahead mentioned here may help detect and deal with such underspecification. The problem can also be handled by making the program learn and adapt as part of its development. For example, extending the semantics of behavioral programming with reinforcements allows applications that also specify, in addition to what should be done or not done at every step, broader goals.[13] Reinforcements are captured by b-threads, each one contributing a narrow assessment of the current situation relative to a longer-term goal. Leveraging the unique structure of behavioral programs, an application-agnostic learning mechanism translates the reinforcements into event-selection decisions that improve over time. This ability to learn and adapt allows removal of the need for a total order on b-threads and event requests, thus simplifying development. For example, a salad-making robot is specified in Eitan and Harel,[13] with scenarios for picking up vegetables, and washing, cutting, and serving them in designated locations. With the help of reinforcements, the robot learns to perform these tasks in the correct order, while overcoming obstacles in the kitchen and dealing with refueling tasks.

Another concern around behavioral programming execution is that if one divergent b-thread (a runaway) fails to synchronize, the entire application stops. The problem is of course aggravated when many behaviors are involved.

As described in Harel et al.[26] and following the work in Barak et al.,[3] we expect that in large behavioral applications not all behaviors will be required to synchronize with each other. Instead, we anticipate that synchronization requirements will be reduced by dividing large numbers of naturally specified behaviors into nodes; each of which is fully synchronized internally, and where the communication between nodes is carried out by external events. The resulting system will still be incremental, in that new functionality can be implemented by adding scenarios to different behavior nodes to generate and interpret (new) external events, with little or no modification to existing ones.

Consider a manager-employee relation in a corporation. Each of the two is constantly driven by a multitude of (personal) behaviors, but without participation in each other‘s decisions. The overhead of a communication protocol, the delays in reacting to messages while continuing autonomous work, and indeed, the occasional correctable misunderstanding, are tolerable, and are balanced with the efficiency and efficacy afforded by autonomy.

The assignment of b-threads to nodes should allow for discovering and dealing with synchronization issues in a local manner, using both model checking and standard development and testing techniques. The division into behavior nodes also simplifies priority assignment, in that one needs to consider priorities only within a node. Different behavior nodes may be associated with different time-scales, reducing synchronization delays.[26] For example, a behavior node for handling a multistop travel itinerary of a quadrotor may synchronize at a much slower pace than the one responsible for stabilizing the aircraft at all times. This division may help also in the runtime detection of runaway b-threads, by using node-specific timers.

The concept of behavior nodes that communicate only via events also makes it easier to avoid race conditions. In this context it should be noted that race conditions are completely avoided in behavioral programming if behaviors communicate only through events, and do not use host language facilities to share data.[26]

Related Work and Future Directions

In some languages (for example, workflow engines or simulation specifications) scenarios and behaviors may be encoded quite directly and visibly. In others (for example, procedural and object oriented programming, functional programming and logic programming) different modularization may cause scenario encodings to be more subtle; rendering them visible only at runtime. One of the main contributions of behavioral programming is the ability to program multimodal scenarios incrementally using modules that are aligned with requirements.[16]

Relative to object-oriented programming, behavioral modules and events may involve objects, but they are not necessarily anchored to a single one. When programming behaviorally, one focuses on system behavior, and less on identifying actors. Often, behavior threads represent inter-object scenarios that are not directly visible when the software is implemented as methods of individual objects. The states of such scenarios often conveniently replace or complement data in standard objects.

Ideas for using behaviors that are specified as refinements and constraints over other modules are discussed in the context of superimpositions.[7] Behavioral programming offers practical programming mechanisms for implementing implicit, indirect control of one behavior over all other relevant behaviors, without explicit references from a controlling or constraining module to the controlled, base module. Additionally, in behavioral programming all system behaviors are treated equally, without the distinction between base and refinements.

Aspect-oriented programming (AOP)[33] focuses on implementing cross-cutting concerns as separate modules that can modify the behavior of base application modules. AOP's relation to superimposition was pointed out in Katz and Gil.[32] We believe behavioral programming can contribute toward implementing symmetric aspects, complementing the currently prevalent asymmetric approach that distinguishes base code from aspects. In addition, while behavioral programming allows the triggering of behaviors by sequences of events, in present AOP implementations, join-points commonly represent individual events, and triggering behaviors following rich sequences of events requires non-trivial state management in the aspect code.

In robotics and hybrid-control there are behavior-based architectures, including Brooks's subsumption architecture,[9] Branicky's behavioral programming,[8] and leJOS,[37] which construct systems from behaviors (see the review in Arkin[1]). Our behavioral programming approach may serve as a formalism, implementation, or possible extension, of some coordination and arbitration components in these architectures.

The test-driven or behavior-driven development methodologies (for example, JBehave, see http://jbehave.org) emphasize the importance of articulating scenarios of expected overall system behavior early in development. As the formal description of scenarios is shown to be valuable, we propose that with behavioral programming it may be possible to actually use such specifications as part of the developed system.

We feel that a key contribution of behavioral programming to established programming paradigms seems to be the addition of a concise and autonomous way for a process to block, or veto, events that other processes may attempt to trigger. In common publish/subscribe mechanisms, for example, such blocking would require additional inter-process communication. In research to be published separately, we prove that the explicit blocking idiom can make behavioral programs exponentially more succinct (in the number of states) than traditional publish/subscribe idioms.

Clearly, behavioral programming principles can be implemented in other languages and environments. We view the approach as an enrichment of, not an alternative to, current programming approaches. In particular, constructs like semaphores/rendezvous, channels/message queues, and threads/continuations, can be used to implement and to complement the synchronization and blocking of behavioral programming. More specifically, in rich decentralized applications, behavioral programming can coexist with actor-oriented, agent-oriented, and other concepts that enable coordination of concurrent processes (see, for example, the survey in Bordini et al.[6]).

In this context, the main point about behavioral programming is its focus on interweaving independent behaviors to yield a desired run (a sequence of events), and the lesser focus on issues related to parallel execution of independent behaviors and the resulting performance gains. In fact, some implementations of the behavioral execution mechanism are single-threaded. It would be interesting to explore the synergy of behavioral programming with such languages, which could be done, for example, by introducing blocking and synchronization idioms into non-behavioral platforms and using established platforms to connect behaviorally programmed nodes.

We believe that behavioral programming can contribute toward implementing symmetric aspects, complementing the currently prevalent asymmetric approach that distinguishes base code from aspects.

The execution semantics of behavioral programming has similarities to the event-based scheduling of SystemC,[39] which performs co-routine scheduling in three phases, *evaluation*, *update,* and *notification*, as follows: all runnable processes are run, one at a time, up to a synchronization point; queued updates are recorded; and, processes affected by these updates are then made runnable.

The BIP language (behavior, interaction, priority) and the concept of *glue* for assembling components proposed by Sifakis and his colleagues (see, for example, Bliudze and Sifakis[5]) pursue goals similar to ours. Though some of the terminology is similar, the specifics are different. BIP focuses on creating a system that is correct-by-construction with regard to safety properties like freedom from deadlock, while behavioral programming concentrates on programming in a natural way, and turns to other techniques, including model checking, to discover and resolve potential conflicts. A possible research direction involves adding synchronization and blocking as composition idioms of BIP.

Finally, behavioral programming may be suitable for software projects that call for feature-oriented development[40] and product-line packaging. For example, an expert version of a game-playing program could differ from the novice version by simply including behavior threads for additional strategies. In Harel et al.[25] we discuss behavioral programming in relation to additional programming languages and models.

As to application domains, we note how features of behavioral programming contribute to making it useful for particular domains, as follows. Coding inter-object scenarios can be useful, for example, for orchestrating valves,

ILLUSTRATION BY LEANDRO CASTELAO

pumps, and the like in automation and process control. The ability to pack distinct, seemingly unrelated behaviors into a single operating entity seems promising in areas like robotics, self-guided vehicles, and the modeling of biological systems. The combination of reactivity and rich scripts can be applied to information-system management including workflow control, event processing, root-cause analysis, and automated configuration, among others. Finally, the ability to trace events in the context of their respective scenarios, may allow decision-making applications to explain their behavior and facilitate ongoing human validation.

Behavioral programming may also accommodate customization as part of the development cycle, where end-users can enhance, change, or remove functionality of delivered systems (for example, smartphones), by coding or downloading new behaviors without accessing the core product code.

As a general paradigm, behavioral programming is still in its infancy. It has been applied to a relatively small number of projects, and the existing tools are not yet of commercial power. More research and development is needed in expanding implementations in a variety of programming contexts and for larger real-world applications. We should also experiment with the collaboration of multiple development groups, and expand the work on formal verification, synthesis, performance and scalability, automated learning and adaptability, the use of natural language, and enhanced play-in.

We feel that the natural incremental development afforded by behavioral programming could become valuable for novices and seasoned programmers alike. We hope the paradigm, with its current implementations in LSC, Java, and other platforms, contributes to the vision of *liberating programming*,[16] and that this article will encourage debate about the ideas, as well as further research and development.

Acknowledgments

David Harel thanks Werner Damm and Rami Marelly for the wonderful collaborations of 1998–1999 and 2000–2003, respectively, without which the ideas described here would not exist. We are grateful to Shai Arogeti, Yoram Atir, Michael Bar-Sinai, Daniel Barkan, Dan Brownstein, Nir Eitan, Michal Gordon, Amir Kantor, Hillel Kugler, Robby Lampert, Shahar Maoz, Yaarit Natan, Amir Nissim, Yaniv Saar, Avital Sadot, Itai Segall, Nir Svirsky, Smadar Szekely, Moshe Vardi, Moshe Weinstock, Guy Wiener, and Guy Weiss for their valuable comments and contributions throughout the development of this article and the ideas behind it. Thanks to Shmuel Katz for insights on positioning behavioral programming relative to aspect-orientation. We also thank the anonymous reviewers for their valuable suggestions.

Harel's and Marron's research was supported in part by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant to DH from the European Research Council (ERC) under the European Community's FP7 Programme. Weiss's research was supported by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University and by a reintegration (IRG) grant under the European Community's FP7 Programme. C

References

1. Arkin, R.C. *Behavior-Based Robotics.* MIT Press, 1998.
2. Atir, Y. and Harel, D. Using LSCs for scenario authoring in tactical simulators. In *Summer Computer Simulation Conference.* Soc. for Comp. Simulation Int., 2007.
3. Barak, D., Harel, D. and Marelly, R. Interplay: Horizontal scale-up and transition to design in scenario-based programming. *Lectures on Concurrency and Petri Nets*, (2004), 66–86.
4. Berry, G. and Cosserat, L. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, Springer, 1985, 389–448.
5. Bliudze, S. and Sifakis, J. A notion of glue expressiveness for component-based systems. *CONCUR*, 2008.
6. Bordini, R.H., Dastani, M. Dix, J. and Seghrouchni, A.E.F. *Multi-Agent Programming: Languages, Tools and Applications.* Springer, 2009.
7. Bouge, L. and Francez, N. A compositional approach to superimposition. In *POPL*, 1988.
8. Branicky, M.S. Behavioral programming. In *Working Notes AAAI Spring Symp. on Hybrid Sys. and AI*, 1999.
9. Brooks, R. A robust layered control system for a mobile robot. *IEEE J. of Robotics and Automation 2*, 1 (1986).
10. Bunker, A., Gopalakrishnan, G. and Slind, K. Live sequence charts applied to hardware requirements specification and verification. *Int. J. on Software Tools for Technology Transfer 7*, 4 (2005).
11. Damm, W. and Harel, D. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design 19*, 1 (2001).
12. Eitan, N., Gordon, M., Harel, D., Marron, A. and Weiss, G. On visualization and comprehension of scenario-based programs. *ICPC*, 2011.
13. Eitan, N. and Harel, D. Adaptive behavioral programming. *IEEE Int. Conf. on Tools with Artificial Intelligence*, 2011.
14. Gordon, M., Marron, A., and Meerbaum-Salant, O. Spaghetti for the main course? Observations on naturalness of scenario-based programming. *ITICSE.* To appear July 2012.
15. Harel, D. From play-in scenarios to code: An achievable dream. *IEEE Computer 34*, 1 (2001).
16. Harel, D. Can programming be liberated, period? *IEEE Computer 41*, 1 (2008).
17. Harel, D., Katz, G., Marron, A. and Weiss, G. Non-intrusive repair of reactive programs. *ICECCS.* To appear July 2012.
18. Harel, D., Kleinbort, A. and Maoz, S. S2A: A compiler for multi-modal UML sequence diagrams. *Fundamental Approaches to Software Engineering*, 2007.
19. Harel, D., Kugler, H., Marelly, R. and Pnueli, A. Smart play-out of behavioral requirements. *FMCAD*, 2002.
20. Harel, D., Lampert, R., Marron, A. and Weiss, G. Model-checking behavioral programs. In *EMSOFT*, 2011.
21. Harel, D. and Marelly, R. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer, 2003.
22. Harel, D. and Maoz, S. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling 7*, 2 (2008), 237–252.
23. Harel, D., Maoz, A., Szekely, S. and Barkan, D. PlayGo: Towards a comprehensive tool for scenario based programming. *ASE*, 2010.
24. Harel, D., Marron, A., Nissim, A. and Weiss, G. Combining behavioral programming and fuzziness for hybrid control systems. In Proc. 2012 IEEE International Conference on Fuzzy Systems. To appear June 2012.
25. Harel, D., Marron, A. and Weiss, G. Programming coordinated scenarios in Java. *ECOOP*, 2010.
26. Harel, D., Marron, A., Weiss, G. and Wiener, G. Behavioral programming, decentralized control, and multiple time scales. *AGERE!*, 2011.
27. Harel, D. and Naamad, A. The STATEMATE semantics of statecharts. *TOSEM 5*, 4 (1996).
28. Harel, D. and Pnueli, A. *On the Development of Reactive Systems, in Logics and Models of Concurrent Systems. NATO ASI Series, Vol. F-13.* 1985.
29. Harel, D. and Segall, I. Planned and traversable play-out: A flexible method for executing scenario-based programs. *Tools and Algorithms for the Constr. and Anal. of Systems*, 2007.
30. Harel, D. and Segall, I. Synthesis from live sequence chart specifications. *Computer System Sciences*, 78:3 (2012), 970-980.
31. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A. and Pree, W. From control models to real-time code using Giotto. *IEEE Control Systems Magazine 23*, 1 (2003).
32. Katz, S. and Gil, J.Y. Aspects and superimpositions. *AOP Workshop at ECOOP*, 1999.
33. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M. and Irwin, J. Aspect-oriented programming. *ECOOP*, 1997.
34. Kugler, H., Plock, C. and Roberts, A. Synthesizing biological theories. In *CAV*, 2011.
35. Kugler, H. and Segall, I. Compositional synthesis of reactive systems from live sequence chart specifications. *Tools and Alg. for the Constr. and Anal. of Systems*, 2009.
36. LEJOS. Java for LEGO Mindstorms; http://lejos.sourceforge.net/.
37. Lieberman, H. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann, 2001.
38. Maoz, S. and Harel, D. From multi-modal scenarios to code: Compiling LSCs into AspectJ. In *FSE*, 2006.
39. OSCI. Open SystemC Initiative. IEEE 1666 Language Reference Manual; http://www.systemc.org.
40. Prehofer, C. Feature-oriented programming: A fresh look at objects. *ECOOP*, 1997.
41. Sadot, A., Fisher, J., Barak, D. Admanit, Y. Stern, M.J., Hubbard, E.J.A and Harel, D. Toward verified biological models. *IEEE/ACM Trans. Comput. Biology Bioinform 5*, 2 (2008).
42. Shimony, B., Nikolaidis, I., Gburzynski, P. and Stroulia, E. On coordination tools in the PicOS tuples system. *SESENA*, 2011.
43. Wiener, G., Weiss, G. and Marron, A. Coordinating and visualizing independent behaviors in Erlang. In *9th ACM SIGPLAN Erlang Workshop*, 2010.

David Harel (dharel@weizmann.ac.il) is The William Sussman Professorial Chair in the Department of Computer Science and Applied Mathematics at The Weizmann Institute of Science, Rehovot, Israel.

Assaf Marron (assaf.marron@weizmann.ac.il) is a researcher at The Weizman Institute of Science, Rehovot, Israel.

Gera Weiss (geraw@cs.bgu.ac.il) is an assistant professor in the Department of Computer Science at Ben Gurion University of the Negev, Be'er Sheva, Israel.