Online Search in Behavioral Programming Models

Orel Moshe Weinstock

Department of Computer Science, Ben-Gurion University of the Negev

Abstract—We present a model based approach to Search Based Software Engineering (SBSE). The approach is based on the Behavioral Programming (BP) paradigm where independent aspects of behavior are woven at run time using a simple interaction protocol. We propose to extend the behavioral programming execution mechanism with on-line heuristic search in program state space that allows programmers to develop non-deterministic programs while relying on a "smart" event selection mechanism to resolve non-determinism in a way that maximizes a specified heuristic function. The paper presents a new library that we have developed in Java and in JavaScript, using Rhino, to facilitate the proposed modeling approach and programming style. We give examples, in the context of a StarCraft game bot built with the library, that demonstrate how the proposed programming idioms can simplify the code and help build robust reactive systems.

I. Motivation and Background

Search Based Software Engineering (**SBSE**) is an emerging field of research which aims to cope with the increased demand for functionality, scalability, and robustness of computer programs (and of reactive robotic systems in particular) using heuristic search mechanisms [1]. SBSE consists of automatic resolution (using search algorithms) of complex decisions that programmers model as optimization problems. There are many published papers in this area that describe various approaches within the software engineering research community [2], [3]. There are also reports that describe how SBSE has been successfully applied to solve problems in nearly all software development life cycle phases [4], [5]. The main challenge in SBSE is, of course, finding a good modeling technique that facilitates the search [6].

Despite the research activity in the area, search methods are practically used only in specific domains. Harman [2] reports, for example, that 54% of SBSE tools are used for testing purposes, an additional 11% for maintenance, and another 10% for project management. It seems that the main barrier that delays further adaptation of the technique is shortage in models for online search [6].

The goal of this this paper is to explore how SBSE can be made accessible to modelers and programmers of reactive systems, such as robotic applications and interactive game bots, as **idioms** that integrate with standard constructs **in common modeling and programming languages**. This allows for natural, powerful derivation from modeling languages (such as LSC [7], [8]) to code. Specifically, we aim at tools that facilitate the following software development methodology:

1) Code and/or derive from models the high-level specification of the system's behavior, using non-determinism to specify free choices in execution.
2) Run the system using an engine that resolves non determinism heuristically or synthesize deterministic code.

3) If unsatisfied with the execution's choices, extend the model by formalizing more refined requirements.
4) Repeat steps 2 and 3 until the behavior is satisfactory.

The behavioral programming (BP) paradigm that we focus on in this paper is described in detail in Section II. BP extends and generalizes scenario-based programming which was introduced with the language of *live sequence charts* (LSC) [7], [8]. In addition to the refinement idioms that already exist in BP, which allow programmers to incrementally shape their software by adding modules that can both widen and narrow the set of possible behaviors of the system [9], we propose in this paper to allow BP based models to also contain specification of fitness criteria for the heuristic search function that can also be refined along the above development process.

The idea of "smart" execution of scenario based specifications started in [10] and in [11] with proposals to apply, respectively, model-checking and planning algorithms for running a single super-step (the part of the run that spans between two consecutive external events) in LSC. We apply a similar mechanism in the context of a behavioral programming library embedded in an imperative programming language. Beyond running in a different setting, the main addition of our library, when compared to these earlier contributions, is that it runs the "smart" event selection mechanism at run-time, on real program code rather than on a model or specification, and that it can consider a horizon beyond a single super-step.

II. Behavioral Programming Principles

As presented in [12], a behavioral model consists of a set of independent behavior threads (b-threads for short). Each b-thread is a specification of a reactive machine that can be modeled, e.g., as a procedure in an imperative programming language. Together, the b-threads control the behavior of flow of the application via a synchronization protocol, as follows. When a b-thread reaches a point that requires synchronization, it waits until all other b-threads reach such synchronization points in their own flow. At synchronization points, each b-thread specifies three sets of events: (1) requested events - the thread proposes that these events be considered for triggering, and asks to be notified when any of them occurs; (2) waited-for events - the thread does not request these events, but only asks to be notified when any of them is triggered. The platform will not consider these events for triggering, unless given as external input to the system; and (3) blocked events - the thread currently forbids triggering of these events.

As shown in Figure 1, when all b-threads are at a synchronization point, a legal event (an event that is requested by at least one b-thread and is not blocked by any b-thread) is chosen. This chosen event is then triggered by resuming all the

b-threads that either requested or waited 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 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.

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

We will now give the semantics of a set of b-threads in terms of runs of a transition system. For this, we model each behavior thread as a transition system with an association of requested and blocked events to each state:

Definition 1. behavior thread [12]: A behavior thread (abbr. b-thread) is a tuple $\langle S, E, \rightarrow, init, R, B \rangle$, where $\langle S, E, \rightarrow, init \rangle$ forms a deterministic total labeled transition system, $R \colon S \to 2^E$ is a function that associates each state with the set of events *requested* by the b-thread when in that state, and $B \colon S \to 2^E$ is a function that associates each state with the set of events *blocked* by the b-thread when in that state.

We define a composition operator on the set of b-threads and the resulting set of runs of the composite transition system as follows:

Definition 2. Runs of a set of b-threads [12]: We define the runs of a set of b-threads $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i \rangle\}_{i=1}^n$ as the runs of the labeled transition system $\langle S, E, \rightarrow, init \rangle$, where $S = S_1 \times \cdots \times S_n$, $E = \bigcup_{i=1}^n E_i$, $init = \langle init_1, \ldots, init_n \rangle$, and $\rightarrow$ includes a transition $\langle s_1, \ldots, s_n \rangle \xrightarrow{e} \langle s_1', \ldots, s_n' \rangle$ if and only if

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

and

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

When multiple events are requested and not blocked, the semantics of event selection may vary. The process of choosing the next event, the focus of this research, is called *Arbitration*. Various arbiters have been suggested in other works:

- A naïve arbiter would select a legal event at random, as in the LSC Play Engine [8].
- Choosing a minimal event according to b-thread, event, and request order [12], [13].

- Look-ahead subject to desired properties of the resulting event sequence, as in smart play-out [10].
- Planning algorithms [11], called planned play-out in LSC.
- Reinforcement learning where events that have shown to produce better expected value are selected [14].
- Allow concurrent events or split execution into parallel concurrent executions, as in [15].
- Synthesizing specifications into a deterministic automaton (e.g. [16]).

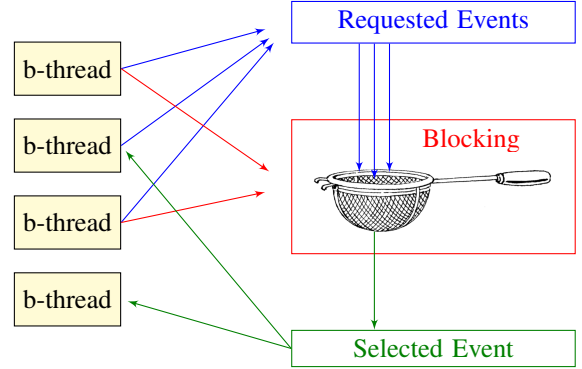In this research we adopt and extend the mechanism proposed in [11] as elaborated in Section IV below.

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

III. BEHAVIORAL PROGRAMMING IN JAVASCRIPT

The intent is for programmers to use the principles introduced in the previous section in an imperative programming language. To illustrate this coding technique, consider a b-thread that increases water flow in a hot water tap by requesting five times the event AddHot which stands for turning the tap anticlockwise for some small fixed amount. Another b-thread performs a similar action, with the event AddCold, on the cold water tap. To increase the water flow in both taps in parallel, as may be desired for keeping the temperature stable, one may activate the above b-threads alongside a third one, which forces the interleaving of events in the two scenarios. The third b-thread, for example, can be coded as "repeatedly: block AddCold until AddHot; block AddHot until AddCold". This programming style was proposed in [12] in Java and is extended to Javascript in our work. A Javascript program for the water tap application is shown in Figure 2.

As seen in Figure 2, defining b-threads is easy and concise - you simply register a function with the application object, which is embedded in the Javascript interpreter from the underlying Java. Synchronization is induced by calling a method called bsync, passing to it three sets of events: requested, waited-for, and blocked events (in this order). The return value of bsync is the triggered event that resumed the b-thread. All multi-threading and concurrency issues are handled by the BP-Javascript engine.

```
bpjs.registerBThread("HotBt", function () {
    while(true){
        bsync(addHot, none, none);
    }
});
bpjs.registerBThread("ColdBt", function () {
    while(true){
        bsync(addCold, none, none);
    }
});
bpjs.registerBThread("AlternatorBt", function () {
    while(true){
        bsync(none, addHot, addCold);
        bsync(none, addCold, addHot);
    }
});
bpjs.registerBThread("Display", function () {
    while(true){
        bsync(none, all, none);
        bplog("Event: " + lastEvent());
    }
});
bpjs.start();
```

Figure 2. An example of using BP-Javascript.

The application object is not the only object available to the Javascript programmer. Other objects from the Java layer, like the predefined event set objects none and all, and addCold and addHot can also be embedded in the interpreter and, thus, can be accessed by the Javascript code. The ability to embed Java operations and objects within the interpreter allows us to create DSLs for sharing common functionality between b-threads and a much easier way to extend the application or even embed it in another BP-Javascript application.

The b-threads can combine the full power of the Javascript language with synchronized behavioral execution and can also dynamically integrate with applications containing non-behavioral components. BP-Javascript also enjoys the flexibility and conciseness of Javascript without sacrificing any of the semantics of the Java BP back-end.

A comparison to [12], which has no backtracking, and [17], which uses external tooling for backtracking, is in order. Our implementation adapts [17] for online search by using Javascript backtracking over Rhino with a modified BPJ library without exteral tooling, see V.

IV. BP-JAVASCRIPT WITH SEARCH

While BP, as presented in the preceding sections, is useful in allowing for relatively independent code components, the model is, by definition, a non-deterministic system that leaves a choice when there is more than one event that is requested and not blocked. To get a standard, deterministic, implementation, programmers can add b-threads that refine the specification or, as we propose in this paper, use a "smart" execution mechanism. Specifically, we now demonstrate how an extension of BP-Javascript with an application agnostic search-based event selection mechanism allows for a cleaner and more robust program. The examples from now on are based on a library we have developed to support this approach. We demonstrate how it can be used in the context of a StartCraft [18] bot. The bot itself is only in an early development phase, the code here is only an outline, not part of a full, working implementation.

```
function findAndHarvestMinerals(move) {
    while(true){
        var minerals = bsync(new FindMineral(this), none, none
            );
        var othersHarvesting = new OthersHarvesting(this,
            minerals.getLocation());
        while(notFullyLoaded){
            var harvest =
            bsync(new Harvest(minerals.getLocation()),
            none, othersHarvesting);
            updateLoaded(harvest.getAmount());
        }
        bsync(new ReturnToBase(this), none, none);
    }}
```

Figure 3. A b-thread for a worker assignment to mineral fields. FindMineral is an event list of all orders to harvest a visible mineral field, one for each such field. Harvest is a predefined event class for harvesting a given mineral field. OthersHarvesting is an event set which includes all Harvest event by other workers.

```
var totalMinerals = 0;
function h(bpstate) {
    var le = bpstate.getLastEvent();
    if(isMineralsCollectedEvent(le))
        totalMinerals = totalMinerals + le.getValue();
    return totalMinerals;
}
```

Figure 4. The heuristic function summing total collected minerals.

Figure 3 shows an example of using BP-Javascript with search for optimal worker assignment to mineral fields in the game of StarCraft. The example is accompanied by the screenshot shown in Figure 5. The function described is registered as a b-thread for each worker. To see how the search mechanism works in this example, let us examine the arbitration process for a behavioral program with these b-threads bsync by bsync, assuming the heuristic function's value is the total amount of minerals harvested as in Figure 4. Each time a bsync is executed (lines 3,8,12), BP will apply the heuristic function to the game state.

Line 3: The worker asks for orders to mine any visible mineral field - it requested such an order for each visible field. The worker will then start harvesting the location it was sent to at line 8. At this point, the BP system has to choose which mineral field harvest order to trigger. The system will search

Figure 5. Zerg hatchery (red square) with Drones (worker units, green squares) harvesting minerals (blue crystals). Drones collect and bring minerals to the hatchery, adding to the player's credit. Optimal resource collection allows the player to pay for combat units, more workers and new buildings.

through the different branches of the program space, each starting with a different harvest order. A sub-optimal choice will get a lower heuristic value than the optimal choice. **Line 8:** The worker harvests the field he received in the preceding `bsync`. While this worker is harvesting the mineral field (receiving `Harvest` events), because of the blocked event set given to `bsync`, no other b-thread (worker) can receive a `Harvest` event on the same field (i.e. can harvest it). While searching through program space at this point, any branch in which a worker will wait on an already taken mineral field will get a lower heuristic value than one in which there are no workers who are waiting for others to finish with the field. Therefore, the heuristic drives the search away from unnecessary waits by workers. **Line 12:** The worker asks for orders to head back to base with the collected minerals.

This example shows how the non-determinism created by the existence of multiple requested yet unblocked events is resolved by the search engine, using an appropriate heuristic function. This can be utilized to write concise programs that would have been longer on a regular imperative platform, even when employing search techniques [19], [20]. As a rough quantitative comparison, assume only 10% of the code managing the build order (resource collection, unit training and building construction sequence) in the bot coded in [20] deals with harvesting minerals. That is approximately $40\mathrm{KB}$ of source code, significantly longer than the code in Figure 3.

V. UNDER THE HOOD

Defining the program-state correctly has great influence on search results. At every synchronization point where the arbiter has to choose the next event, we run the program with our choices while controlling its inputs and outputs to determine the heuristic value of states reached in the run and choose the event leading to the highest scored state. The model in this case is the actual program state - all its variables, stack frames, memory space etc.

The technique of running the program in a controlled environment is called *sandboxing*, and is often used for testing [21]. Although using this method requires writing an environment simulator (to generate inputs), in addition to the desired application itself, this is not a significant penalty to development as a simulator is a required in order to test the application. While running the program in a sandbox with a simulator in its entirety may be difficult for big general purpose applications (due to the large codebase), it is viable for control/reactive systems - the focus of this research. This type of systems contains higher level code that concentrates on logical flow, which runs faster than general code. Note that the simulation of the environment does not have to be complete, a good search mechanism can make much use even of an abstract description of the environment.

We will now discuss the implementation of the program-state object. Programs in BP are comprised of b-threads, so their states is an aggregate of the independent state of each b-thread. Anything happening inside a b-thread between `bsync` calls - that is, between synchronization points - is by definition internal to the b-thread, and so can be considered atomic to the program as a whole. Therefore, we can ignore these internal workings and focus on `bsyncs`, as only the states at these synchronization points define the integrated system behavior.

The only programmatic construct that captures program execution in an immutable, re-entrant object, as required by search algorithms, is a *continuation* [22], [23]. Continuations are representations of the program at a given point in execution, which are available to the programmer, rather than hidden by the runtime environment. They are used here to traverse the state space by ordinary program execution, as they facilitate backtracking and resuming of execution from desired points where they were captured. We can now formally define the state space for the search:

Definition 3. A BT-state represents the state of a specific b-thread in a `bsync` call during a run of the program. It is composed of the b-thread and its captured continuation.

```
public BEvent bsync(RequestInterface requested,
    EventSet waited, EventSet blocked) {
    _request = requested;
    _wait = waited;
    _block = blocked;
    ...
    Context cx = ContextFactory.getGlobal().enterContext();
    _cont = cx.captureContinuation();
    ...
}
```

Figure 6. Code snapshot of the implementation of the `bsync` function in the underlying Java b-thread object.

As shown in Figure 6, a BT-state (`_cont`) is captured at every call to `bsync` by the underlying Java b-thread object. This ensures that once all b-threads have reached a synchronization point, their continuation object representing that state is updated, so that a complete state of the BP system can be captured:

Definition 4. A BP-state represents the state of the whole program. It is composed of the BT-states of all b-threads in the program, captured at a `bsync`.

When the BP infrastructure is required to make a choice between multiple events to trigger, it creates a BP-state as a root for the search. Expanding search nodes is done by triggering legal events and capturing the new BP-states created by the triggering. This is done by executing the code as in [17], and not offline or by running on an abstract model of the code as in [24]. This ensures that there are no discrepancies between the code itself and the search results. The BT-state and BP-state are the abstractions used by the search algorithm directly such that all BP specific code is encapsulated within those objects and is completely transparent to the search algorithm.

With the state space defined, we can delve into the search mechanism itself - how we run behavioral programs in a sandbox. The sandbox is composed of the environment simulator (input generator), a search algorithm and a heuristic function.

An implementation of a look-ahead mechanism, beyond one super-step, requires that the system be able to predict the actions of the environment to some precision. For this, we need to ask programmers to provide the search mechanism with an abstract model of the environment (which can be probabilistic), a simulator, to provide inputs to the behavioral program while in the sandbox.
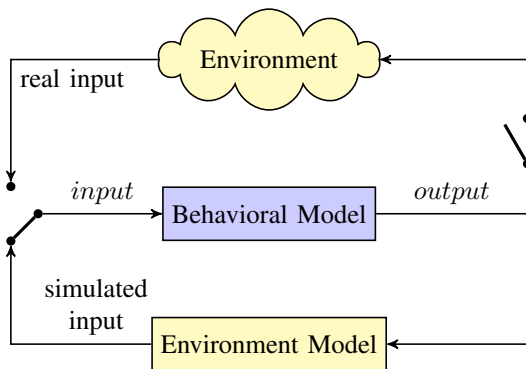
Figure 7. The architecture of the BP search sandboxing mechanism. We developed a switched system that operates in two modes. (1) In normal mode the application b-threads receive input from the environment. (2) In search mode, the application b-threads get input from a simulator and output is fed only to the simulation b-threads, so that the environment is not actuated. The figure shows the state of the switches in search mode. When in normal mode, both switches are in their other mode, i.e., the right switch is closed and the left switch is connecting 'real input' to 'input'.

As shown in Figure 7, we have implemented an environment simulator as b-threads in the program itself. In normal operation, a simulator b-thread's `bsync` is modified such that its requested event sets are added to the waited-for events set, and its requested event set is empty. This ensures the simulation b-thread is made aware of all events relevant to it, so that it maintains a correct state for the next use in simulation mode.

When the BP infrastructure needs to search for an event to trigger, the simulation b-threads are switched to simulation mode, in which they request events normally triggered by the environment as modeled in their code. No manipulation of their event lists is done in simulation mode. This "Eating our own dog food" approach greatly contributes to the robustness of the simulator and program, while also providing a clear and unified interface for programming behavioral programs.

Let us now consider another example, illustrated in Figure 8. In this example, we demonstrate usage of the environment simulator to encode our knowledge that the environment-driven Marines attack the closest enemy. We therefore code the Marine simulator function as in Figure 9 and register it as a simulation b-thread. When in normal operation, BP will receive the Marine's actions from the environment. When in simulation mode, the yellow Marine will request to attack the top Hydralisk, and the red Marine will request to attack the bottom Hydralisk. If we learn more about Marine behavior, we can change the function in Figure 9, or register more b-threads specifying their behavior as simulation b-threads.

An interface for sending inputs to the behavioral program and for examining its outputs is then required. A convenient solution for this is defining the interface to and from the behavioral program to be an event queue (an input queue has been introduced in [25]). This way the environment and the sandbox both enqueue events for triggering within the behavioral program in the input queue and read the behavioral program's output from its output queue. The behavioral program does not directly perform actions on the environment - events from the output queue are fed as player actions back into the environment by an adapter, thus enabling running in sandbox without extra code analysis.

Figure 8. Hydralisks (green) and Marines (yellow, red) accompanied by a Medic (light blue) in combat.

```
1  function killClosest(move) {
2      var closestEnemy = getClosestEnemy();
3      bsync(new AttackCommand(closestEnemy),none,none);
4      while(closestEnemy.isAlive()){
5          bsync(new Shoot(this,Enemy),none,none);
6      }}
```

Figure 9. The Javascript function registered as the Marine's b-thread. The `bsync` in line 3 will request and trigger the AttackCommand event, which is an output event. The `bsync` in line 5 will request the Shoot event, which is an input event, in order to keep track of the enemy's health. When the enemy is killed, we can move to a new target.

Selecting the right search algorithm for a behavioral program can have great impact on the results. The algorithms we have used are depth-limited `A*` and `minimax` [26] textbook implementations [27]. The architecture of our solution is such that it is easy to introduce other search algorithms, as there is no coupling between the algorithm itself and the BP-Javascript engine. The specific choice of best search algorithms for specific application domains is beyond the scope of this work.

Writing a heuristic function is straightforward: the BP-state object passed to the function grants access to the entire program without compromising speed or space. This includes the b-thread's `bsync` event sets and public access methods. The programmer, then, is given full power in the evaluation of program state, independent of the search algorithm used. The programmer can write different heuristic functions that reward desired events and b-thread properties.

VI. Conclusion

In this paper we have shown how a BP engine with an embedded program-state space search mechanism can facilitate development in a new, more natural way using a standard, modern programming language. The BP architecture was enhanced to be more flexible and real-world ready. The game bots we programmed in this manner accomplished their initial goals by fulfilling their function in the game while being relatively short and easy to code. This will allow us to further explore and advance the use of BP, in general and in AI domains particularly, using run-time search to provide the programmer with more freedom.

REFERENCES

[1] M. Harman and B. F. Jones, "Search-based software engineering", *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[2] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: trends, techniques and applications", *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[3] H. Jiang, Z. Ren, X. Li, and X. Lai, "Transformed search based software engineering: a new paradigm of sbse", in *SSBSE*, 2015.

[4] J. Manuel, C. Trilla, S. Poulding, and C. Runciman, "Weaving parallel threads: searching for useful parallelism in functional programs", in *SSBSE*, 2015.

[5] S. Yoo, "Amortised optimisation of non-functional property in production environment", in *SSBSE*, 2015.

[6] J. Ahluwalia, I. H. Krüger, W. Phillips, and M. Meisinger, "Model-based run-time monitoring of end-to-end deadlines", in *Proceedings of the 5th ACM international conference on Embedded software*, ACM, 2005, pp. 100–109.

[7] W. Damm and D. Harel, "Lscs: breathing life into message sequence charts", *Formal methods in system design*, vol. 19, no. 1, pp. 45–80, 2001.

[8] D. Harel and R. Marelly, *Come, let's play: scenario-based programming using LSCs and the play-engine*. Springer Science & Business Media, 2003, vol. 1.

[9] D. Harel, A. Marron, and G. Weiss, "Behavioral programming", *Communications of the ACM*, vol. 55, no. 7, pp. 90–100, 2012.

[10] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, "Smart play-out of behavioral requirements", in *FM-CAD*, Springer, vol. 2, 2002, pp. 378–398.

[11] D. Harel and I. Segall, "Planned and traversable play-out: a flexible method for executing scenario-based programs", in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2007, pp. 485–499.

[12] D. Harel, A. Marron, and G. Weiss, "Programming coordinated behavior in java", in *ECOOP 2010–Object-Oriented Programming*, Springer, 2010, pp. 250–274.

[13] G. Wiener, G. Weiss, and A. Marron, "Coordinating and visualizing independent behaviors in erlang", in *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, ACM, 2010, pp. 13–22.

[14] N. Eitan and D. Harel, "Adaptive behavioral programming", in *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*, IEEE, 2011, pp. 685–692.

[15] H. Kugler, C. Plock, and A. Roberts, "Synthesizing biological theories", in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, G. Gopalakrishnan and S. Qadeer, Eds., ser. Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 579–584, ISBN: 978-3-642-22109-5. DOI: 10.1007/978-3-642-22110-1_46. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_46.

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

[17] D. Harel, R. Lampert, A. Marron, and G. Weiss, "Model-checking behavioral programs", in *Proceedings of the ninth ACM international conference on Embedded software*, ACM, 2011, pp. 279–288.

[18] (1998). Starcraft: brood war — wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/StarCraft:_Brood_War (visited on 07/11/2015).

[19] D. Churchill and M. Buro, "Build order optimization in starcraft.", in *AIIDE*, 2011.

[20] ——, (2015). UAlbertaBot, Starcraft bot code, [Online]. Available: http://github.com/davechurchill/ualbertabot (visited on 07/11/2015).

[21] A. Fox, E. Brewer, *et al.*, "Harvest, yield, and scalable tolerant systems", in *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, IEEE, 1999, pp. 174–178.

[22] G. D. Plotkin, "Call-by-name, call-by-value and the λ-calculus", *Theoretical computer science*, vol. 1, no. 2, pp. 125–159, 1975.

[23] J. C. Reynolds, "The discoveries of continuations", *Lisp and symbolic computation*, vol. 6, no. 3-4, pp. 233–247, 1993.

[24] G. Katz, "On module-based abstraction and repair of behavioral programs", in *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer, 2013, pp. 518–535.

[25] D. Harel, A. Kantor, G. Katz, A. Marron, G. Weiss, and G. Wiener, "Towards behavioral programming in distributed architectures", *Science of Computer Programming*, vol. 98, pp. 233–267, 2015.

[26] S. Russell and P. Norvig, "AI a Modern Approach", *Learning*, vol. 2, no. 3, p. 4, 2005.

[27] (2015). Java implementation of algorithms from Norvig and Russell's "Artificial Intelligence - A Modern Approach", [Online]. Available: http://github.com/aima-java/aima-java (visited on 07/11/2015).