Model-Checking Behavioral Programs

David Harel, Robby Lampert, Assaf Marron*
Weizmann Institute of Science
Rehovot 76100, Israel
firstname.lastname@weizmann.ac.il

Gera Weiss†
Ben Gurion University of the Negev
Beer Sheva 84105, Israel
geraw@cs.bgu.ac.il

ABSTRACT

System specifications are often structured as collections of scenarios and use-cases that describe desired and forbidden sequences of events. A recently proposed *behavioral programming* approach, which evolved from the visual language of live sequence charts (LSCs), calls for coding software modules in alignment with such scenarios. We present a methodology and a supporting model-checking tool for verifying behavioral Java programs, without having to first translate them into a specific input language for the model checker. Our method facilitates early discovery of conflicting or under-specified scenarios, which can often be resolved by adding new scenarios rather than by changing existing code. Also, counterexamples provided by the tool are themselves event sequences that can serve directly for refinements and corrections. Our tool reduces the size of the execution state-space using an abstraction that focuses on behaviorally interesting states and treats transitions between them as atomic.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program verification—*Model Checking*; D.1 [**Programming Techniques**]: Miscellaneous; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

*The research of the first three authors was supported 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 from the European Research Council (ERC) under the European Community's Seventh Framework Programme (FP7/2007-2013).

†The research of the fourth author 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.

General Terms

Verification

Keywords

Behavioral Programming, Java

1. INTRODUCTION

Behavioral programming, originating in the language of *live sequence charts* (LSCs) [7, 16], is a recently-proposed approach for scenario-based development of reactive systems [18]. It calls for constructing systems from threads of behavior, each of which independently represents (a part of) an allowed, desired, or forbidden operating scenario of the final system. The collective execution of behavior threads is coordinated in a way that generates a combined sequence of events, which constitutes cohesive system behavior. The approach is described and illustrated as part of Section 2. In [18], general advantages of behavioral programming, such as incrementality in development and alignment with how people often describe behaviors, are demonstrated using a Java package called BPJ.

The current paper describes a methodology for model-checking-assisted development of behavioral programs. To support the methodology, we present a model checker for behavioral programs that are written in Java with the BPJ package. The main motivation for developing such a model-checker is to find conflicts and under-specification as follows.

Since specifications for reactive systems are often modeled with scenarios (e.g. in the form of Message Sequence Charts), using behavior threads to encode them can create a software system whose components are aligned with the requirements, and reflect them directly. An obvious limitation of this approach is that requirements sometime conflict, or are not detailed enough, and composing them automatically without consideration may yield a composition that produces undesired joint behavior, the resolution of which may be complicated and error-prone.

We first observe that behavioral programming suggests an interesting way to resolve conflicts and underspecification: refinements can be ed as new requirements, and be coded as additional behavior threads or scenarios. E.g., when two behaviors require that opposing actions take place following a given event sequence, a conflict-resolving refinement may provide the conditions under which each of the two behaviors should prevail.

Nevertheless, the above leaves open the question of early discovery of conflicts and underspecification, and in this paper we demonstrate a proof-of-concept model-checker specifically designed for this purpose.

In addition to the basic model-checking functionality, we also show synergies between behavioral programming and the proposed model-checking tool. Since the tool can verify behavioral Java programs directly, without translating them first into a model-checker-specific language, behavioral software components can serve both as elements of a final executable system, and as elements of an abstract system model to be subjected to verification later on. In addition, the full power of Java can be used to flexibly and conveniently specify the properties to be verified. Lastly, the tool's construction is simplified by leveraging the inherent abstraction in behavioral programs, where program execution between event occurrences is treated as atomic.

The rest of the paper is structured as follows. In Section 2 we review the principles of behavioral programming and illustrate, through a detailed example, how model-checking can be used in the development of a behavioral application; in Section 3 we review the design of the model-checking tool, present summaries of initial experience and performance in the context of well known problems, and explore possibilities for leveraging the behavioral programming capabilities as part of a more general methodology; and, in Section 4 we review related approaches to development, synthesis, program repair, verification and execution that include model-checking.

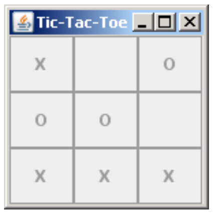
2. EXAMPLE: MODEL-CHECKING AN EVOLVING APPLICATION

In this section we describe the usage of our model-checker in the development of a behavioral application for playing Tic-Tac-Toe, with an emphasis on incremental development. Along the way, we briefly review the basics of behavioral programming. The way incrementality works is that following an initial setup of the application, each program refinement, developed in response to a counterexample provided by the model checker, is programmed either as a new behavior thread (abbreviated b-thread) or by changing configuration parameters such as relative priorities of b-threads. This is in contrast to the more common approach, in which discovery of conflicts and underspecification often results in changes to previously developed modules. Moreover, as counterexamples represent scenarios (that should be avoided) they can constitute the basis for the added b-threads. The example is borrowed, with modifications, from [17, 18]. In our description of the application we omit some details, focusing on those that are relevant in the context of model checking.

First, let us describe the (classical) game of Tic-Tac-Toe, and the events that represent the expected behaviors. Two players, X and O, alternately mark squares on a 3×3 grid whose squares are identified by $\langle row, column\rangle$ pairs: $\langle 0,0\rangle$, $\langle 0,1\rangle, \ldots, \langle 2,2\rangle$. 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 draw.

In our example, player X should be played by a human user, and player O is played by the application. Each move (marking of a square by a player) is represented by an event, $\mathsf{X}_{\langle row,col\rangle}$ or $\mathsf{O}_{\langle row,col\rangle}$. The events XWin, OWin, and draw represent the respective victories and a draw.

A play of the game may be described as a sequence of events. E.g., the sequence $\mathsf{X}_{\langle 0,0\rangle}$, $\mathsf{O}_{\langle 1,1\rangle}$, $\mathsf{X}_{\langle 2,1\rangle}$, $\mathsf{O}_{\langle 0,2\rangle}$, $\mathsf{X}_{\langle 2,0\rangle}$, $\mathsf{O}_{\langle 1,0\rangle}$, $\mathsf{X}_{\langle 2,2\rangle}$, XWin describes a play in which X wins, and whose final configuration is:



Before proceeding with the development of the example application, we briefly review the BPJ principles and operation described in [18]. Each b-thread is coded as a Java thread. The code of each b-thread uses API method calls to induce synchronization with all other b-threads; i.e., whenever a b-thread reaches a synchronization API call, it waits for all other b-threads to reach such points in their own code. At each synchronization point, 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) *watched* or *waited-for* events — the thread does not request these events, but asks to be notified when any of them is triggered; and (3) *blocked* events — the thread currently forbids these events.

When all b-threads are at a synchronization point, a central mechanism uses the specified sets to determine the next triggered event, as follows. It iterates over all b-threads in the order of their priorities, and for every b-thread it iterates over the ordered set of its requested events. Each event is checked as to whether it is blocked (i.e., belongs to the blocked-events set of some b-thread). If indeed it is blocked, the event-selection mechanism moves on to the next event. Otherwise, the event is triggered by resuming all the b-threads that either requested or waited for this event. The resumed b-threads proceed with their execution to their next synchronization point, while each of the other b-threads remains at its last synchronization point, oblivious to the triggered event, until an event it requested or is waiting for, is selected. When all b-threads are again at a synchronization point the process repeats. When there are no requested events that are not blocked, no event is triggered and the program waits indefinitely. BPJ allows the dynamic addition of b-threads (by non-behavioral components), and these, in turn, may request events that cause the behavioral program to resume its operation. When it is known that no such b-threads will appear, the program is considered to be in a deadlock.

Note that the total order between the b-threads and between the requested events of each b-thread implies that the standard execution of a behavioral program by BPJ is deterministic. That is, the specified event sets of all b-threads uniquely determine the triggered event (if any are possible). As detailed in Section 3 below, in developing the model-checking tool for behavioral programming, we replaced the standard BPJ event-selection mechanism with one that is tailored for model-checking runs, in that it allows b-threads to request sets of events *nondeterministically*. This enables model-checking an application with respect to all possible environment behaviors.

We now describe the incremental development of the application with the assistance of our proof-of-concept model-checking tool, referred to here as BPmc (for behavioral programming model checker). Suppose that the developer first creates the b-threads that enforce the rules of the game:

- `SquareTaken`: block further marking of a square already marked by X or O.
- `EnforceTurns`: alternately block O moves while waiting for X moves, and vice versa (we assume that X always plays first).
- `DetectXWin` (resp. `DetectOWin`): wait for placement of three X marks (resp. O marks) in a line and request XWin (resp. OWin).
- `DetectDraw`: wait for nine moves and request draw event.

Due to the ordering considerations mentioned above, the b-threads `DetectXWin` and `DetectOWin` are given higher priority than `DetectDraw`. If this is not done, then if X wins in the ninth move, draw will be triggered instead of XWin. A behavioral 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 to the program components that request these events:

- A GUI component that translates each user-click on a selected square to a corresponding X event (this component also displays the game-board to reflect the X and O move events).
- `DefaultMoves`: a b-thread that repeatedly requests all nine possible O moves in the following order of center, corners and edges: $O_{\langle 1,1\rangle}$,$O_{\langle 0,0\rangle}$,$O_{\langle 0,2\rangle}$,$O_{\langle 2,0\rangle}$,$O_{\langle 2,2\rangle}$, $O_{\langle 0,1\rangle}$,$O_{\langle 1,0\rangle}$,$O_{\langle 1,2\rangle}$,$O_{\langle 2,1\rangle}$.

Now, we wish to use the model checker to gradually and incrementally enhance the program until it never loses.

In order to verify that there is no strategy for X to win against the program, we first replace the user-driven entry of X moves with the b-thread `XAllMoves`, which repeatedly, nondeterministically, requests all nine possible X moves (this will be the only nondeterminism in the program).

We also have to specify that the desired safety property is that O never loses. This is done by modifying the b-thread `DetectXWin`, where it requests the XWin event, to call an API method that declares the next state of the program as bad. This causes BPmc to announce that the verification failed, and to print the relevant event sequence as a counterexample (see Section 3, below, for more details).

Finally, before verifying an application with BPmc, additional small modifications are sometimes required. In our case, several b-threads were modified not to terminate before the termination of the program.

The program is now ready to be model-checked with BPmc, which is done by running it with appropriate run-time parameters.[1]

Model-checking the behavioral program consisting of the above b-threads immediately shows that the application may lose. The counterexample trace printed by BPmc is:

```
X(0,0), O(1,1), X(0,1), O(0,2), X(2,0), O(2,2), X(1,0).
```

The victory of X could have been easily avoided if the application had played $O_{\langle 1,0\rangle}$ 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:

- `PreventThirdX`: when two Xs are noticed in a line, add an O in that line (and prevent an immediate loss).

Running BPmc again after adding this b-thread, we get the same error trace. A closer look (which can be done, e.g., using the trace visualizer in [10]) reveals the cause: the priority assigned to the new b-thread is lower than that of `DefaultMoves`, which then prevails with its request to play $O_{\langle 2,2\rangle}$. To overcome this problem, we assign `PreventThirdX` a higher priority and model-check again. This time we get the trace:

```
X(0,0), O(1,1), X(2,1), O(0,2), X(2,0), O(1,0), X(2,2).
```

Here, the source of the problem is that once X plays his/her third move at $\langle 2,0\rangle$, a 'fork' is created (with $\langle 0,0\rangle$ on one hand and $\langle 2,1\rangle$ on the other). In this situation, a victory for X is inevitable. To avoid this situation, we add the b-thread:

- `PreventXFork`: when two Xs are noticed, in all configurations symmetric to the one shown in the counterexample, mark an O in the intersection corner of the potential fork, thus preventing its creation.

This b-thread's priority is higher than `DefaultMoves`'s, but lower than `PreventThirdX`'s, as it is more important to prevent an immediate loss.

When we run the tool again, we get the following trace:

```
X(0,0), O(1,1), X(2,2), O(0,2), X(2,0), O(1,0), X(2,1).
```

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

- `PreventAnotherXFork`: when the first two Xs are marked in two opposite corners and the first O is marked at the center, request $O_{\langle 0,1\rangle}$. In the spirit of "the best defense is a good offense", this move creates an attack that forces X to play $X_{\langle 2,1\rangle}$, 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 model checking shows that in the final program this assumption is indeed correct.

When running the model-checker at this stage, we still get a counterexample:

```
X(0,0),O(1,1),X(2,2),O(0,1),X(1,2),

O(0,2),X(2,0),O(1,0),X(2,1).
```

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 (twice in this trace) the opportunity to win. Thus, we add the b-thread:

[1] One can try the process by commenting-out lines of `TicTacToe.java` (in [17]) that add b-threads to the behavioral program, leaving only already-developed b-threads, and then running the model-checker.

- `AddThirdO`: when two Os are located on a single line, add a third O (and win).

Clearly, this b-thread should get the highest priority, since winning the game is always the best move. After adding it, we may hope we are done, but the model checker comes up with another counterexample:

```
X(1,2),O(1,1),X(2,1),O(0,0),X(2,2),O(2,0),X(0,2).
```

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-thread:

- `PreventYetAnotherXFork`: when two Xs are noticed in two edge squares that are adjacent to a common corner, mark an O in that corner.

Now, finally, the model checker confirms that we are done.

This Tic-Tac-Toe example shows how a model checker may be used in developing a behavioral program. Here it was accompanied by some knowledge-based effort by the developer: identify the source of the problem presented in the counterexample, and design a solution. However, a counterexample supplied by the model-checker may often be directly used in improving the solution, as follows. Treat the counterexample as a scenario, and prevent its occurrence by creating a corresponding 'anti-scenario' — a b-thread that waits for all but the last application-driven event in the counterexample, and blocks the last event choice. Other b-threads should then take care of requesting the correct move. This approach, which has been used also in the development of a model for the vulva precursor cells of the *C. elegans* nematode [23], may become useful in partially automated ways to patch programs (c.f. [22]).

3. BPMC: A PROOF-OF-CONCEPT TOOL

This section provides an overview of our proof-of-concept tool, BPmc. The tool can be viewed as enabling a sort of "in vivo" model-checking for behavioral programs, since the Java programs constituting the b-threads continue to execute with the native JVM. BPmc is integrated into the BPJ execution control mechanism described in [18], and the code is posted in [17].

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. Safety mode explores the different paths in the graph to search for a state that violates the given safety property, while liveness mode looks for cycles that violate the given liveness property. Our graph traversal techniques follow the algorithms described, e.g., in [2] and use the Apache `javaflow` package to save and restore continuations — objects that hold the states of participating threads — for the backtracking required during these traversals. All three execution modes (standard, and model checking of safety/liveness properties) rely on transitions defined by the enabled events — events that are requested and not blocked. Further details are provided below, explaining terms used, features and capabilities (as well as limitations) of the tool, and how the application being verified can interact with it.

3.1 Features, interfaces, internals

BT-states: In line with the definitions of b-threads in [18], which are based on transition systems, we chose an abstraction in which a *state* of a b-thread (abbr. BT-state) is defined only at the point where the Java thread executing the b-thread calls the API method `bSync`, declaring a synchronization point (see Figure 1). Specifically, we do not explore intermediate transitions in the code that b-threads execute between exiting one `bSync` and entering the next. This abstraction provides for reducing the size of the state-space, based on the assumption that b-threads interact only through events, which means that, for the purposes of model-checking, their activities between calls to `bSync` are considered atomic and are independent of each other. We believe that this assumption is reasonable, and does not unduly constrain the capabilities of behavioral programs.

BP-states: The state of a behavioral programming system (abbr. BP-state) when all its constituent b-threads are at a synchronization point is the Cartesian product of the BT-states of all b-threads.

Transitions: BPmc discovers the state transitions "on-the-fly". State-space traversal is carried out by executing all Java behavior modules per the collective execution algorithm of behavioral programming, as described above, and as defined formally in [18]. The BT-states of the individual b-threads are used to determine the next BP-state. Specifically, the set of successors of a BP-state *bps* are all the BP-states reachable by resuming and executing all b-threads with each of the possible nondeterministic selections of triggered events, as described in the nondeterminism paragraph below. The b-threads are resumed at the BT-states comprising *bps*, and are stopped when each of them reaches its next BT-state.

Nondeterminism: In the standard execution mode of BPJ, all event requests are subjected to a total order, to ensure the deterministic execution that at each synchronization point chooses the first event that is requested and not blocked. With our model-checking tool we relax the requirement of a total order and introduce nondeterminism by allowing the optional specification of b-threads and/or subsets of requests that have equal priorities. The requested events are examined in priority order as in the standard execution mode, but when one is found that is not blocked, all other requests of the same priority that are not blocked are considered nondeterministic alternatives.

Simulating environment behavior: Model-checking requires a closed system, hence events driven by the environment, and randomness, need to be simulated. In our approach, this "environment simulation" is done with b-threads dedicated for this purpose. Specifically, these can be very simple b-threads, which, repeatedly, at every synchronization point, request all possible environment-driven events, or may reflect other assumptions. For example, in model-checking the Tic-Tac-Toe application, since we focus on the strategy, we chose not to simulate user clicks, and instead, repeatedly request all nine X moves. Note that the b-thread `SquareTaken` ensures that only legal X moves are selected. The approach used for environment simulation may be extended towards a more general methodology, as described in Section 3.4.

Backtracking with `javaflow`: The design of BPmc depends on the ability to backtrack an executing Java thread

```
.
.
.
labelNextVerificationState( “A” );
bSync( … );
if( lastEvent == event1 ) {
      .
           .
      .
      labelNextVerificationState( “B” );
      bSync( … );
}

if( lastEvent == event2 ) {
      .
      .
      .
      labelNextVerificationState( “C” );
      bSync( … );
}
```

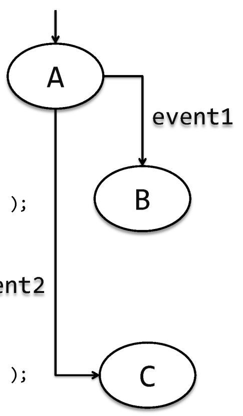


Figure 1: Behavior-thread states. Application behavior threads are coded as Java methods, and include general Java code and calls to the API method `bSync`, which synchronizes with other behaviors and waits for the next event. In the proposed abstraction, states of a behavior are the program states (in an operating-systems sense) at synchronization points (entries to `bSync`). When an event occurs, the b-threads waiting for it are resumed, and can examine it in the variable `lastEvent`. The execution constituting the transition between states is considered atomic. The abstraction assumes that b-threads communicate only via events. The state of the entire behavioral program is the Cartesian product of the states of its constituent behavior threads.

to a previous state. This is done with the help of the Apache `javaflow` package [1]. The `javaflow` package enables saving copies of a thread's execution stack at a any point during its execution, in objects called *continuations*. One can then resume the thread's execution as needed from the saved continuations.

State-space exploration and state hashing: The algorithm used by BPmc for model-checking explores the state space by execution of the participating b-threads. During the search, the tool keeps track of visited BP-states, and when it reaches previously visited BP-states it backtracks to explore new paths in the state space. In order to backtrack to a previous BP-state, the execution stack of each of the behavior threads is restored to its state in that BP-state, using `javaflow`. In order to proceed to the next BP-state, an event is selected, and each b-thread that requested, or waited for, this event is resumed and is then suspended again at the next synchronization point. BPmc supports both breadth-first and depth-first searches (for safety properties), as well as nested DFS (for liveness properties), and can apply unconditional, strong, and weak fairness assumptions/constraints in the process, as elaborated upon below.

Safety properties: During the state-graph traversal, the running behavior threads can call an API method to mark BP-states as *bad*. For example, in the Tic-Tac-Toe program, when X completes a winning triple, the b-thread `DetectXWin` marks the next state as bad, which causes the printing of the game lost by O as a path (i.e., an event-trace) that leads to the bad state. In addition to handling bad states, the tool detects deadlock states without b-thread assistance: when there are no requested events in a BP-state, or all requested events are blocked, the BP-state has no successors, and BPmc identifies a deadlock. Recall that external environment events are simulated by b-threads, so that there are no additional events that can cause the system to exit from this BP-state.

Liveness properties: Liveness properties are defined and verified as follows. Similarly to safety, and inspired by the multi-modality of LSCs [7], behavior threads can mark BP-states as *hot*. The model checker verifies liveness properties by looking for cycles in the state graph that contain only hot states (this search uses a nested DFS algorithm similar to the one suggested in [2, page 211] for verification of liveness). In other words, the verified liveness property is that always eventually, the system is not in a hot state.

Fairness assumptions: BPmc may also be used to verify liveness properties subject to fairness constraints that may be unconditional, strong, or weak, as defined, e.g., in [28, 30]. Fairness assumptions accompany liveness verification in BPmc, by eliminating from consideration ”unfair” cycles, as follows. We maintain three sets of events for every cycle: those that are enabled in the cycle (requested and not blocked in at least one state), those that are continuously enabled in the cycle (enabled in every state of the cycle), and those that are actually triggered in the cycle.

Below, we list the three types of fairness constraints supported by BPmc. For each type, we cite the intuitive description from [2, page 129] and briefly describe our implementation in the context of behavioral programming:

- For unconditional fairness (“Every process gets its turn infinitely often”): a set of events is provided as input. Only cycles that contain an event from the set in their set of triggered events are considered fair and participate in the liveness verification.

- For strong fairness (“Every process that is enabled infinitely often gets its turn infinitely often”): a set of events is provided as input. A cycle is unfair if it contains events from the input set in its set of enabled events, but none of the events from the input set is in its set of triggered events.

- For weak fairness (“Every process that is continuously enabled from a certain time instant on gets its turn infinitely often”): a set of events is provided as input. A cycle is unfair if it has an event from the input set in its set of continuously-enabled events, but none of the events from the input set is triggered in the cycle.

BP-state labeling: The search process backtracks whenever it reaches a BP-state that was previously visited. Visited BP-states are identified according to a BP-state name, computed as the ordered concatenation (or Cartesian product) of the names of its constituent BT-states. Indirectly, state names are our way for allowing programmers to specify equivalence relation over states. Behavior threads can use an API method call to assign a name to their next BT-state. When a b-thread does not label its states, its component in the BP-state name is ignored when comparing BP-states.

BT-state caching: BPmc takes advantage of the fact that each of the many possible BP-states belongs to the Cartesian product of a relatively small set of BT-states. Each BT-state, with its corresponding thread stack (`javaflow` continuation), is saved at most once. BP-states only point to the constituent BT-states.

Application-assisted search pruning: In addition to pruning the search and backtracking automatically when revisiting a BP-state, b-threads can call an API method to explicitly force the tool to abandon the current search path and, by backtracking, continue the search in other paths. This is done, for example, by the b-thread `DetectDraw` after nine moves were completed with none of the players winning.

Counterexamples as scenarios: As is the case with any model checker, counterexamples include the path to the bad state. This sequence of events can be directly translated to a b-thread that follows (waits for) the events in the sequence. Thus, in the context of behavioral programming, counterexamples provide readily usable guidance for constructing corrections or refinements. For example, in a naïve approach, a b-thread can be written to wait for a sequence of events that appears in a prefix of the counterexample, and then, with a higher priority than the rest of the application, to take different actions. Of course, a programmer considering one or more counterexamples can generalize the required correction, and perhaps also leverage the BT-state information available with the sequence of BP-states leading to the failure.

3.2 Initial experience and performance

To further illustrate the capabilities of BPmc, we now describe our initial experience in programming several examples, emphasizing the modeling of different problem types and the usage of BPmc interfaces for application-assisted model checking.

We tested and compared the functionality of the tool against a published industrial case-study and the familiar *dining philosophers* and *bridge-crossing* problems, as provided on the Spin benchmark web site at [21]. We also provide initial performance results. The code for these and other examples is available on-line at [17].

An industrial case study

We now describe the application of BPmc in the context of an industry case study of scheduling memory access in a signal processing board; see [11, 32]. The memory of a radar system is connected to several I/O buffers, which are, in turn, connected to input and output data streams that cannot be stopped. An arbiter component schedules the transfer from input buffers to memory and from memory to output buffers. For a given setup that includes the number of buffers, their sizes, their data rates, and a fixed memory transfer schedule, we wish to verify that input buffers never overflow and that output buffers are never exhausted.

With BPmc we were able to validate the schedule proposed in [32], by reproducing the original model-checking results obtained using SMV.

In [11] the model includes a complicating factor of a memory-refresh cycle every certain large number of clock ticks, during which memory transfers cannot occur. With BPmc, we were able to show that if the memory-refresh cycles are scheduled so that they occasionally interfere with scheduled memory transfers, then eventually an overflow occurs.

The model/code included the following main independent behaviors: The data-flow behavior from each buffer's point of view, the memory transfer schedule, and the scheduling of the occasional memory-refresh that may interfere with the above transfers.

Note that if behavioral programming idioms are incorporated in the (microcode) language of the final hardware system, and BPmc or a similar model checker are available for that environment, then the code used for modeling could be considered also for use “as is” in the developed hardware.

Dining philosophers

In the dining philosophers problem [9] several philosophers are sitting at a circular table, and are either eating or thinking. At the center is a large bowl of spaghetti, which requires two forks to serve and to eat. A fork is placed in between each pair of adjacent philosophers. Each philosopher may only use the fork to her left and the fork to her right. When a philosopher finishes eating, she puts down the two forks and begins thinking again. Our present goal is to check whether deadlocks, or other starvation conditions, may occur under various philosopher behaviors.

The problem is programmed/modeled behaviorally with the events of the picking up and the putting down of a given fork by a given philosopher (e.g. `PickUp-F2-by-P2` or `PutDown -F1-by-P2`), a b-thread for the behavior of each philosopher and a b-thread for each fork. In the classical version, each philosopher's b-thread repeatedly requests the sequence of events representing her picking up the fork to her right, picking up the fork to her left, putting down the right-hand fork, and then putting down the left-hand one. Each fork's b-thread repeatedly waits for an event of picking up the fork by either of its two adjacent philosophers and then blocks its picking up (again) until the fork is put down.

The application assists the search by labeling the four philosopher behavior states `T` (thinking – forks down), `1` (one fork up), `E` (eating – two forks up) and `F` (finished eating – one fork down). The fork b-thread states are `D` (down) and `U` (up). BPmc uses its basic algorithms and state-hashing capabilities to look for violations of liveness properties, and detect the deadlock conditions that are possible under this classical behavior. For example, a path to a deadlock state in the case of 3 philosophers is displayed as:

```
Verification failed:
init->[T, D, T, D, T, D]
PickUp-F2-by-P2->[T, D, T, D, 1, U]
PickUp-F1-by-P1->[T, D, 1, U, 1, U]
PickUp-F0-by-P0->[1, U, 1, U, 1, U]
 [1, U, 1, U, 1, U] is a deadlock state
```

where each line of the form `<event> -> <BP-State>` describes a BP-state along the path and the event whose triggering led the program to transition from the preceding BP-state to the current.

As in the symmetry-breaking approach in [9], one of the philosophers is left-handed, and thus first picks up the fork to her left, rather than the one to her right. As expected, this setup is then proven by the tool to be deadlock-free, following checking of all possible states.

For liveness testing, the philosopher behavior b-thread marks the non-eating states as hot, and BPmc is then able to detect the starvation conditions that are possible if fairness is weak.

Bridge crossing

In the bridge-crossing problem, four persons cross a bridge that can hold up to two people simultaneously. They have a single torch that must be used when crossing. The persons p1, p2, p3, and p4 can cross the bridge in 25, 20, 10, and 5 minutes, respectively. When two people cross together, they move at the slower person's pace. The torch cannot be tossed over the bridge. The question is: can all four get across the bridge in 60 minutes or less?

We modeled the problem as a Java program using BPJ, as follows. The events `p1Go,p1Ret,...,p4Go,p4Ret,tGo,tRet` respectively represent each of the persons, or the torch, going over the bridge in one direction, or returning. Person-crossing events followed by a torch-crossing event model these persons crossing holding the torch; e.g., the event sequence `p1Go,p2Go,tGo,p1Ret,tRet` means that persons p1 and p2 crossed with the torch, and p1 returned with the torch.

The model/program includes several classes of b-threads. The first has an instance for each of the persons, which repeatedly requests the events of that person crossing and returning, ignoring the constraints. Another b-thread repeatedly requests the events of the torch being taken over and being returned, and blocks persons from walking in the opposite direction. Another b-thread models the constraints that the torch can cross only with a person, and that at most two persons can cross together (the b-thread blocks torch events that are not separated by one or two person events, as well as sequences of three consecutive person events).

Finally, the b-thread Watcher 'awakes' as the result of any event and keeps track of all persons' positions and the total elapsed time as implied by the identity of the crossing persons. This b-thread turns the search into "branch-and-bound" by pruning it and forcing backtracking when the total time exceeds 60 minutes, or when a previously observed person configuration is reached and the total time is not better than the previously observed best time to reach that person configuration. When all persons are together on the desired side of the bridge, and the elapsed time is not greater than 60 minutes, Watcher marks the state as "bad", forcing BPmc to output the solution as an event sequence that constitutes a counterexample, e.g.,

```
Verification failed:
p3Go,p4Go,tGo,p3Ret,tRet,p1Go,p2Go,tGo,p4Ret,tRet,
p3Go,p4Go,tGo
```

Performance table

The table in Figure 2 lists performance results of initial testing of the BPmc tool — before any substantial performance optimization — compared with the performance benchmark results published on the Spin web site at [21]. The purpose of the table here is only to illustrate the basic capabilities of the tool, while further optimization (perhaps application-specific) and detailed benchmarking are a subject for future research.

3.3 Limitations

As mentioned earlier, the purpose of our proof-of-concept BPmc tool is to demonstrate the viability of using model checking in the development of behavioral programs. The tool itself has limitations, some derived from our limited experience with it while others are more inherent in the approach itself.

The tool focuses on the behavioral programming facets of the models encoded in the Java programs, and does not explore alternate Java execution paths, thread scheduling alternatives or other nondeterministic choices of the Java code as may be done, for example, using JavaPathfinder [31]. The tool also assumes that b-threads do not share data, and communicate only via the BPJ event-selection mechanism. Nevertheless, the tool's narrower focus opens up opportunities for improvements in performance and scalability, as well as flexible incorporation into run-time environments of behavioral programs.

The tool relies on state labeling by the application. Automatic detection of BT-state equality based on thread stack contents remains as a future endeavor.

In BPJ, for standard application execution, each b-thread executes in a separate Java thread, enabling parallelism and exploitation of multi-core hardware for the execution between synchronization points. For simplicity of our initial implementation, and as we have not yet fully explored the capabilities and limitations of the javaflow package, the current version of our tool executes as a single Java thread.

The javaflow package is presently a dormant project in Apache. We hope that this or a similar package will become an integral part of Java, and that similar capabilities will be available in other languages where behavioral programming concepts can be readily implemented.

BPmc presently requires that all participating b-threads be registered at the beginning of the run, and does not support BPJ's ability to handle dynamic addition of b-threads. BPmc also requires that the Java threads of the b-threads do not terminate. Applications that were not designed in this way may require some changes — e.g., instead of terminating, a b-thread should wait for an event that by convention is never requested by other b-threads.

Finally, while BPmc supports model-checking of arbitrary behavioral Java programs, its performance depends on that of the application at hand. Thus, the programmer may be required to optimize the application's performance in order to model-check it in a reasonable mount of time. We hope that certain practices, such as when to reuse event objects rather than instantiating them, will eventually be developed in this context.

3.4 Some methodological comments on blocking and compositional verification

The approach described above in Section 3.1 under *Simulating environment behavior*, can be extended as follows.

First, a behavioral component comprised of one or more b-threads may optionally be verified separately from other b-threads, by adding (simple) b-threads that repeatedly, nondeterministically, request *all* possible events that are known in the application, or at least those that may be requested by the external environment or collectively by the rest of the application.

Second, one may then be able to accelerate verification by using b-threads that eliminate certain events, or sequences thereof, from the rest-of-the-system behavior, with event blocking. These b-threads express assumptions about the behavior of the rest of the system or implement partial order reduction, when certain sequences are known to be equivalent. They may be added especially for this purpose, or may already be a part of the verified system, as is the case in the Tic-Tac-Toe application with the b-thread SquareTaken.

Our feeling is that this sculpting of environment behavior, by combining "generator" b-threads that repeatedly request all possible events and other b-threads that only block

	Time (seconds)			States		
	Spin/BEEM database	BPmc counterexample	BPmc no deadlock	Spin/BEEM database	BPmc counterexample	BPmc no deadlock
4 dining philosophers	0	0.031	0.063	80	50	80
6 dining philosophers	0	0.063	0.0172	729	528	728
12 dining philosophers	4.26	3.812	342	531440	46632	531440
4 persons crossing bridge	0	0.547	N/A	96194	24	N/A

Figure 2: Initial performance results for BPmc, both with and without deadlock, using a Lenovo T410 laptop. Published Spin data is shown for general reference, noting that it is not always clear whether the Spin data include deadlock or not, and that they were carried out on a different processor. Entries showing 0 were copied from Spin data, and apparently reflect rounding.

events, is a promising approach in terms of code succinctness. For illustration, to print prime numbers in this programming style, one can use a generator b-thread that requests events labeled with all integers in ascending order starting with 2. Whenever such an event occurs, a listener b-thread prints its label, and starts a b-thread that immediately blocks all events associated with multiples of this label. Though there is a potential for a very large number of b-threads (hence the example is only for illustration of the principle) there are only three b-thread classes, and all but two of the instances vary only by a parameter.

This intuition is supported by a work (to be published separately) in which we were able to prove that explicit blocking allows for constructing systems with smaller building blocks (exponentially smaller, in some cases) than those possible when blocking, or some equivalent thereof, is not allowed.

Finally, the set of all runs of the composition of components verified as described above, is contained in the intersection of the verification runs of each individual component. Formalizing this, and demonstrating when implementing it with event blocking reduces the overall cost of system verification, is also a topic for future research. It will also be interesting to explore whether there are benefits to be derived from incorporating blocking idioms into model-checkers other than BPmc.

4. RELATED WORK

Our model-checking of behavioral programs follows the research on application of formal methods for analysis of live sequence-chart specifications (LSCs), which we now summarize briefly. In [14] an execution mechanism for LSCs is proposed, called smart-play-out, which applies a model checker to the problem of searching for executions that avoid (certain kinds of) deadlocks and failures. A similar execution mechanism is also possible using AI planning techniques [19]. One facet of our work here can be viewed as a natural continuation of this idea: in smart play-out a counterexample to the statement that there is no good execution is used to drive an actual execution of the system, whereas earlier we mentioned the use of output traces of BPmc as new forbidden scenarios that help improve the system under construction. In both cases model checking is used to discover better ways to execute a program rather than just to prove properties thereof.

In [26], a translation of properties specified in LSCs to temporal logic is proposed. In [6], the above tools for the analysis of LSCs are applied to case-studies in the field of telecommunication. In [15], more case studies are described and conclusions regarding methodologies and modeling strategies are drawn. In [8], LSCs are used as a language for specifying properties of UML models. In [12] model-checking is used to prove certain equivalence properties between different executions of a collection of scenarios and [24, 25] extend the results to scenarios that specify aspects.

The present paper can be viewed as contributing a tool that can be used in the context of the work mentioned. In these kinds of uses, and in contrast to approaches that translate the specification into a model that can be analyzed by a conventional model-checker, the main advantage of the proposed model-checking approach is that the translation step is bypassed, reducing practical limitations and possibilities for inconsistencies. The direct approach advocated here also allows "automatic" support for the idioms in the specification language, which may be rich, as in the case of BPJ and Java. In addition, the choice to implement the model checking algorithms inside the verification engine (and not include a model-checker as a black box), facilitates optimization of the search for the specific purpose of finding conflicts and deadlocks in behavioral specifications. This choice also allows special-purpose algorithms for the domain of interest, such as synthesis of reactive strategies. It remains to be seen if and how our approach can be extended to incorporate also symbolic model-checking [4, 29].

Besides being inspired by the ideas behind smart play-out, our approach is influenced by Java Pathfinder (JPF) [31], a Java virtual machine that explores alternative execution paths of Java programs, looking for safety property violations like deadlocks or exceptions. We adopted the idea of directly executing all possible paths as a method for exploring the state space, instead of explicitly constructing the transition relation graph. However, our tool is not a virtual machine but a Java program that applies `javaflow` [1] for executing a set of b-threads in a controlled environment. Moreover, BPmc does not explore all possible executions of multi-threaded Java code, but only all possible selections of events in behavioral programs. This focus yields an abstraction that reduces the search space and allows for more efficient search strategies, especially when a large number of threads is involved.

In fact, before embarking on the development of BPmc we attempted to apply JPF to the Tic-Tac-Toe application.

We believe that the slow performance we experienced (over 80 minutes), was due to JPF's attempts to explore alternative thread schedules for the more than 200 threads in this implementation. This is despite efforts to adapt JPF, both through parameter setting, and by modifying the code of the scheduling mechanism. With BPmc, we were eventually able to verify the application in eight seconds. It is reasonable to assume, though, that with more experience with JPF one would be able to obtain additional significant improvements in performance. A more detailed comparison of our approach with JPF and possible applications of JPF to improve performance and functionality of model-checking behavioral programs are left for future research.

The proposed iterative use of our tool in development, as described by the example in Section 2, is inspired by techniques and methodologies that leverage iterative model-checking and refinement for development, synthesis, and program repair, such as [3, 5, 20, 22]. In addition to the proposed "manual" usage of BPmc in development of behavioral programs, this tool may be useful also in applying the above methods systematically in the context of behavioral programming.

The discussion in Section 3.4 is related to compositional and assume-guarantee reasoning, described, e.g., in [13]. In our behavioral programming framework, the system is composed naturally of multiple relatively small processes that run in parallel. Thus, compositional reasoning may be used to reduce the complexity of the verification.

In [27], a computational model is proposed where the "normal" execution of scenario-based programs is carried out nondeterministically — concurrently across multiple paths. This approach to execution may be applicable also to behavioral Java programs.

5. CONCLUSION

We have demonstrated a method, and a supporting tool, for the direct model checking of behavioral Java programs. We have shown that besides using the tool for verifying nearly-completed applications, model-checking can be used during development for early identification of conflicts and underspecification. The behavioral programming approach is particularly suitable for such iterative, incremental refinement, since it accommodates refinement by addition of new behavior components rather than by changing existing code. The integration of the tool with the behavioral execution mechanism makes it possible to use a single interface for defining both the transition system and its desired properties, further enabling alignment of implementation with requirements. Finally, the modular incrementality of behavioral programs suggests the future possibility of compositional verification that combines succinct modeling with efficient execution.

Acknowledgments

We thank Michael Bar-Sinai for identifying the suitability of `javaflow` continuations for this project, Guy Wiener for insightful comments on a draft of this paper, and the anonymous reviewers for their valuable comments.

6. REFERENCES

[1] Apache Commons. The Javaflow component. `commons.apache.org/sandbox/javaflow/`.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[3] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. *Model Checking Software*, pages 102–122, 2001.

[4] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*, *LNCS* 1855, pages 154–169, 2000.

[6] P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. *Software and System Modeling*, 7(2):157–175, 2008.

[7] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[8] W. Damm and B. Westphal. Live and let die: LSC based verification of UML models. *Sci. Comput. Program.*, 55(1-3):117 – 159, 2005.

[9] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[10] N. Eitan, M. Gordon, D. Harel, A. Marron, and G. Weiss. On visualization and comprehension of scenario-based programs. In *Proc. 19th IEEE Int. Conf. on Program Comprehension (ICPC)*, pages 189–192, 2011.

[11] J. P. Ernits. Memory arbiter synthesis and verification for a radar memory interface card. *Nord. J. Comput.*, 12(2):68–88, 2005.

[12] M. Glusman and S. Katz. Model checking conformance with scenario-based specifications. In *Proc. 15th Int. Conf. on Computer Aided Verification (CAV)*, *LNCS* 2725, pages 328–340, 2003.

[13] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16:843–871, 1994.

[14] 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)*, *LNCS* 2517, pages 378–398, 2002.

[15] D. Harel, H. Kugler, and G. Weiss. Some methodological observations resulting from experience using LSCs and the play-in/play-out approach. In *Scenarios: Models, Transformations and Tools*, pages 26–42, 2003.

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

[17] D. Harel, A. Marron, and G. Weiss. The BPJ package. `www.cs.bgu.ac.il/~geraw`.

[18] D. Harel, A. Marron, and G. Weiss. Programming coordinated scenarios in Java. In *Proc. 24th European Conf. on Object-Oriented Programming (ECOOP)*, *LNCS* 6183, pages 250–274, 2010.

[19] 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), *LNCS* 4424, pages 485–499, 2007.

[20] T. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. *Automata, Languages and Programming*, pages 188–188, 2003.

[21] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. `spinroot.com/spin/whatispin.html`.

[22] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Proc. 17th Int. Conf. on Computer Aided Verification (CAV)*, *LNCS* 3576, pages 226–238, 2005.

[23] N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E. J. A. Hubbard, and M. J. Stern. Formal modeling of C. elegans development: A scenario-based approach. In *Proc. 1st Int. Workshop on Computational Methods in Systems Biology (CMSB)*, *LNCS* 2602, pages 4–20, 2003.

[24] E. Katz. *Verifying Scenario-Based Aspect Specifications*. PhD thesis, Technion - Israel Institute of Technology, Computer Science Department, 2006.

[25] E. Katz and S. Katz. Verifying scenario-based aspect specifications. In *Proc. Int. Symp. of Formal Methods Europe (FM)*, *LNCS* 3582, pages 432–447, 2005.

[26] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal logic for scenario-based specifications. In *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *LNCS* 3440, pages 445–460, 2005.

[27] H. Kugler, C. Plock, and A. Roberts. Synthesizing biological theories. In *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV)*, *LNCS* 6806, pages 579–584, 2011.

[28] D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proc. 8th Int. Colloq. on Automata, Languages, and Programming (ICALP)*, *LNCS* 115, pages 264–277, 1981.

[29] A. Pnueli, Y. Sa'ar, and L. D. Zuck. Jtlv: A framework for developing verification algorithms. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, *LNCS* 6174, pages 171–174, 2010. `jtlv.ysaar.net/`.

[30] J. Queille and J. Sifakis. Fairness and related properties in transition systems – a temporal logic to deal with fairness. *Acta Inf.*, 19:195–220, 1983.

[31] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.

[32] G. Weiss. Optimal scheduler for a memory card. Technical report, IST-2001-35304 AMETIST Project, Weizmann Institute of Science, 2002.