Behavioral Programming

David Harel
Weizmann Institute of Science

Assaf Marron
Weizmann Institute of Science

Gera Weiss
Ben Gurion University of the Negev

ABSTRACT

We describe an implementation-independent programming paradigm, *behavioral programming*, which allows programmers to build executable reactive systems from specifications of behavior that are aligned with the requirements. Behavioral programming simplifies the task of dealing with under-specification and conflicting requirements by enabling the addition of software modules that can not only add to but also modify existing behaviors. A behavioral program employs specialized programming idioms for expressing what must, may, or must not happen, and a novel method for the collective execution of the resulting scenarios. Behavioral programming grew out of the scenario-based language of *live sequence charts* (LSC), and is now implemented also in Java and in other environments. We illustrate the approach with detailed examples in Java and LSC, and also review recent work, including a visual trace-comprehension tool, model-checking assisted development, and extending behavioral programs to be adaptive.

1. INTRODUCTION

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 [25] incrementally from their expected behaviors.

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

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

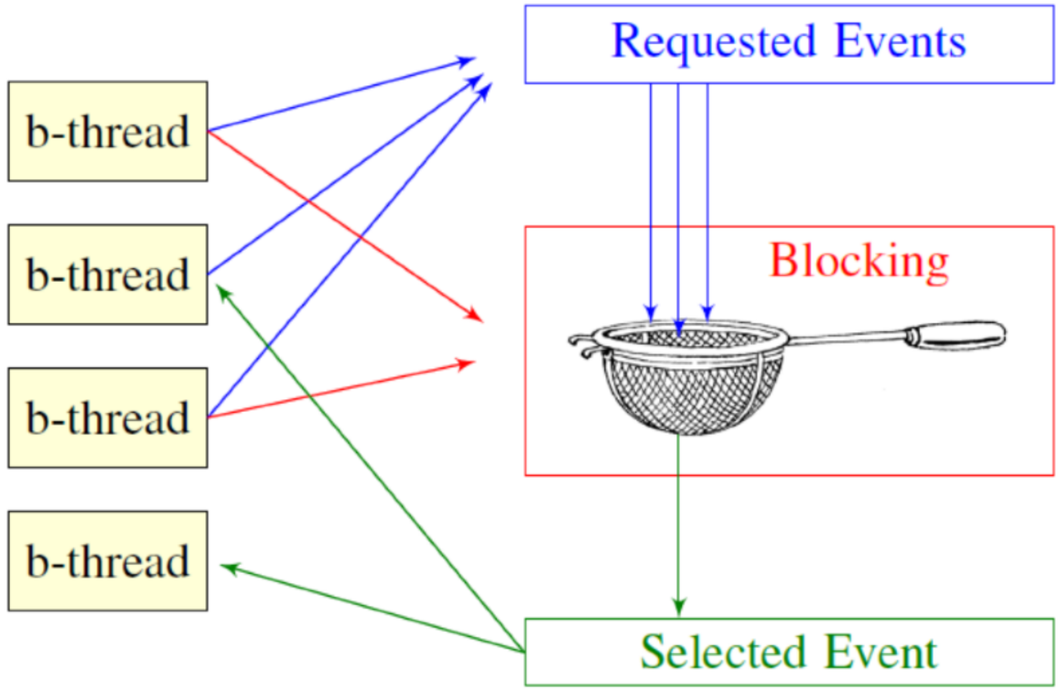
`PreventThirdX`: After the `X` player marks two squares in a line, the `O` player should try to mark the third square (to foil the attack);

`DefaultOMoves`: When other tactics are not applicable, player `O` should prefer the center square, then the corners, 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 detailed in sections 2 and 3. Full behavioral implementations of the game, in Java and Erlang, are described in [22] and [48], respectively. In [18] 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 that 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, e.g., to play at different expertise levels.

Naturally, composing behaviors that were programmed without direct consideration of mutual dependencies raises questions about conflicts, under-specification, and synchronization. We deal with these issues by using composition operators that allow both



1. All b-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 b-threads.
2. An event that is requested and not blocked is selected;
3. b-threads that requested or wait for the selected event are notified;
4. The notified b-threads progress to their next states, where they place new bids.