On Visualization and Comprehension of Scenario-Based Programs

Nir Eitan, Michal Gordon, David Harel, Assaf Marron
Dept. of Computer Science and Applied Mathematics
Weizmann Institute of Science, Israel
firstname.lastname@weizmann.ac.il

Gera Weiss
Dept. of Computer Science
Ben Gurion University, Israel
geraw@bgu.ac.il

***Abstract*—We address the problem of comprehending cause and effect relationships between relatively independent behavior components of a single application. Our focus is on the paradigm of behavioral, scenario-based, programming, as captured by the language of *live sequence charts* (LSC) or its Java-based counterpart, BPJ. In this programming paradigm, multi-modal behaviors can be specified separately, and are integrated only at run time. We present a tool, with which the user can easily follow the decisions of the collective execution mechanism. It shows the behaviors and events that were executed at each point in time, and those that were delayed or abandoned, as well as the causes and reasons behind these run-time choices. The dynamic effects of such decisions on the system's behavior can be seen easily too.**

***Keywords*-trace visualization; behavioral programming; scenario-based programming; BPJ**

I. Introduction

We propose a tool for visualizing traces of behavioral programs in support of comprehension. The *behavioral programming* approach (BP) was introduced via the scenario-based programming language of *live sequence charts* (LSC) [1]. Later, a java-based counterpart, BPJ, was proposed too [2]. In the BP approach, behaviors are programmed independently, or semi-independently, of each other, and are interlaced at run-time, by a collective execution mechanism. This results in a cohesive, integrated system behavior.

In this paper, we address the problem of analyzing cause and effect issues in program traces. Specifically, we propose browsing, filtering, and grouping mechanisms for comprehending traces. Our main challenge is to enhance understanding of executions by best illustrating the flow of each behavior module as well as the indirect interaction between behaviors, and to be able to use this understanding to clarify the resulting sequence of actions and the reasons certain actions are not executed at all.

Our target problem domain is that of behavioral programs, but we believe that similar issues arise also in other programming contexts. Specifically, we concentrate on behavioral programs written with the Java package BPJ, available on [3]. Each program consists of modules, called *behavior threads* (or *b-threads*), that can request events, wait for events and block (i.e., forbid) events. Collective execution of b-threads uses an enhanced publish/subscribe protocol in which: (a) all b-threads synchronize and place their "bids", specifying requested events and blocked events; (b) the first event (subject to a given order) that is requested and is not blocked is chosen; (c) b-threads waiting for the event are notified; (d) the notified b-threads progress to their next states, where they all synchronize again and can place new bids. See [2] for more motivation, technical details, examples, etc.

Despite the apparent naturalness of behavioral programs, comprehending them may not be trivial. In the words of Green [4]: *"If a language highlights the conditions under which actions are to be taken, as in a rule-based language, then it probably obscures the sequential ordering of actions. [...] part of the notation design problem is to make the obscured information more visible"*. In a way, it is exactly this problem that we wish to address in the present paper; namely, the fact that the very constructs of the BP approach (and particularly its manifestation in BPJ) can easily obscure the actual sequence of execution. The visualization tool we propose here combines visualization of individual behaviors, information about bidding at synchronization points (*syncpoints*), and the resulting event trace. Together with navigation and filtering techniques, this trace visualization enhances comprehension of behavioral programs, and can help the user/programmer "see" not only why certain events were chosen but also why others were not (e.g., due to selection order or blocking).

II. Visualizing BPJ Traces

The tool uses a table-like display to visualize the run of a behavioral program. The display is interactive and uses distinct notations according to accepted cognitive guidelines [5], [6]. B-threads are depicted in columns ordered by priorities, and syncpoints are represented by rows ordered by time that intersect the b-thread columns.

Each cell in the table represents a b-thread state at a given syncpoint. The sets of events requested, waited-for, and blocked by the b-thread are shown in sub-cells marked *R*, *W*, and *B* (see Figure 1).

We define three cell types: the *leader cell*, of which there is one per syncpoint, is the one that contains the event request chosen at this syncpoint; *active cells*, for b-threads that will advance because they wait for the chosen event, and

1063-6897/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPC.2011.10

IEEE computer society

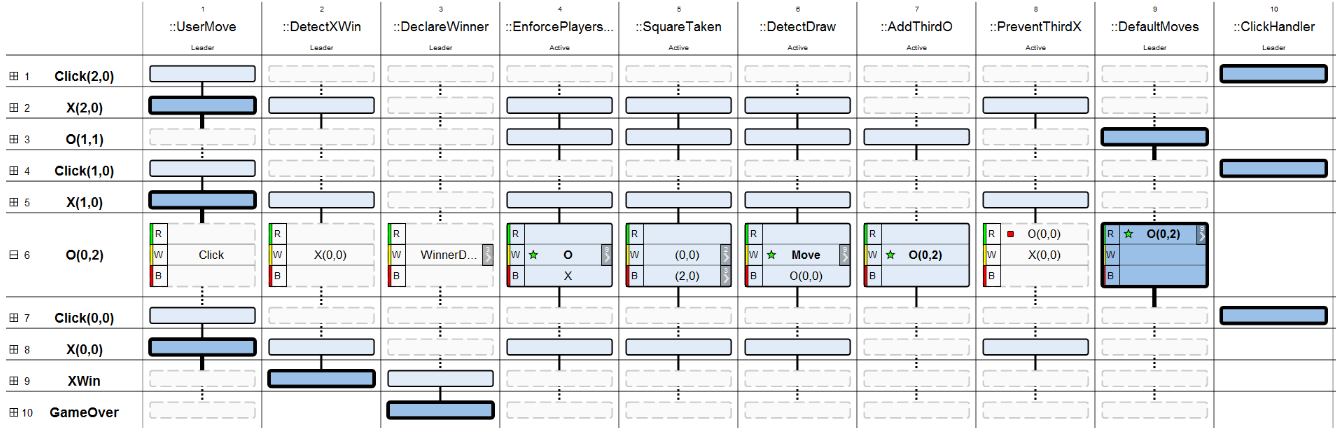

Figure 1. Visualizing a game of Tic-Tac-Toe. Columns show b-threads grouped by class. Each row is associated with a syncpoint and shows the triggered event. When a row is expanded (e.g., row 6) the events requested, waited-for and blocked by each b-thread are also shown.

non-active cells, for b-threads that do not advance. Each cell type is identified by both its background and its borderline: dark, medium or light background, and a thick, thin or dashed borderline, respectively; see Figure 3.

The user can perform focused scrolling by pressing the *shift* key: the table is scrolled down, but making sure that the column with the current leader cell remains in view. To follow a particular behavior thread, the user can scroll to the next leader cell in a given column using the navigation button adjacent to the column's name.

A. Events and events sets

As shown in Figure 2, the initial column on each syncpoint row shows the selected event, thus forming an event-trace. The copy of this event that appears in requested and waited-for event sets in all active cells is emphasized by a green star icon, clarifying why each b-thread advanced.

In behavioral programming, understanding why an event was not chosen is just as important as understanding why other events were. Accordingly, events that are blocked are marked by red square icons. Coupled with the priority order of the columns and the order of the events in the requested-events sets, the tool helps understand why a non-starred event instance was not selected.

BPJ supports both concrete event sets and rule-based sets; e.g., "all events of a given class". A button next to the set name enables viewing all displayable events in the set. In rule-base sets, which may be very large, only events that are requested at the syncpoint are shown.

B. Scalability and filtering

Traces can be very large. This is due not only to the number of events but also to the number of b-threads. A large number of b-threads may result from the programmer's preference to create a b-thread instance for every variation in behavior (perhaps using parameterized instances of the same b-thread class). There may also be b-threads that are dynamically created (and perhaps later deleted) at runtime. Each of these b-threads occupies an entire column, of which many cells may be empty.

One mechanism that reduces the number of columns in the display is *grouping* b-thread instances by class into one column. The sets of requested, wait-for and blocked events shown in each cell are the union of the corresponding sets over all instances.

Another feature reduces the number of columns by *filtering* the b-threads to be displayed subject to the following classification, based on their functionality in the trace:

- *Non-active*: b-threads that did not block or request anything and did not advance either; they waited for events that did not occur in the trace.
- *Active-follower*: b-threads that advanced when certain events occurred, but did not block or request events.
- *Active*: b-threads that requested or blocked events, but advanced only when event requests of other b-threads were selected.
- *Active-leader*: b-threads that contributed to driving the run. At least one of their event requests was chosen at a syncpoint (they have at least one leader cell).

Depending on the user's goals he/she can choose which b-thread types to display. For example, to simply examine the run's progress it is useful to view active and active-leader b-threads, while to understand why something did not occur it may be appropriate to view also non-active b-threads.

Long traces can be handled by *focusing* on a trace segment between two syncpoints. Once such a segment is displayed, one may navigate forwards and backwards in the full trace, or jump to the next active cell of a specific b-thread.

Finally, *collapsing* reduces cell size by hiding its details, thus allowing for a broader view of the trace in a single screen. This can be done for a specific syncpoint or for the entire trace. In collapsed mode, the color and borderline of

the cell continue to indicate the cell's type (see Figure 3) .

C. Implementation

The BPJ package run can create an XML file that contains trace information, with all necessary details about b-threads, syncpoints and events. The tool reads this file to create the trace visualization. The application was developed as a web application, enabling easy sharing of run results with others, with no need for software installation. The visualizer was written using JavaScript and Raphael [7] — a JavaScript library that uses scalable vector graphics (SVG), and which enables zooming in and out without losing sharpness.

III. Example

In this section, we use our tool to examine a BPJ program that plays Tic-Tac-Toe against a human. Two players, X and O, take turns in marking a 3x3 board, each trying to form a horizontal, vertical or diagonal line to win. See [3] for a demonstration of this and other examples.

The initial screen shows statistics that can help the user in initial filtering decisions. For example, one can select only the 11 active-leader b-threads and the 16 active b-threads and ignore, for the time being, the 96 non-active and 91 active-follower ones. This can also help in making grouping or filtering decisions. In the Tic-Tac-Toe example the over 200 b-thread instances arise from only 14 classes. But each class represents an "externally meaningful" behavior. For example, the 48 instances of `DetectWinByX` look for all possible sequences of events that constitute a win by player X. Grouping these instances together reduces the cognitive load by removing some information, and helps in understanding the overall run. In Figure 1 both filters were used, and in many cases the combination of class name and chosen event already provides sufficient explanation.

We now show how to use the visualization to study a section of a normal run of the program. A partial view showing the first six syncpoints and a subset of the class group columns is shown in Figure 2. The X and O events are triggered alternately, each with its own `(row,column)` position. The `Click` events requested by `ClickHandler` represent the human player's button clicks that are translated by the program into X moves. By examining the syncpoints where O events occurred, one can see what strategy (class) was taken by the computer. By examining the column of the class or the specific b-thread instance of the leading cell, one can see the main events that contribute to a specific decision.

The first strategy chosen is `DefaultMoves`, and it marks square `(1,1)`. This means that there was no higher priority move. Indeed examining all b-threads at this syncpoint shows that no b-thread to the left of `DefaultMoves` requested any event. Clicking the event set of the cell, one can see the 9 default moves, from which the first possible one was chosen. Note that `O(2,2)` is blocked by `SquareTaken` , due to X's first move. At the next syncpoint in which O

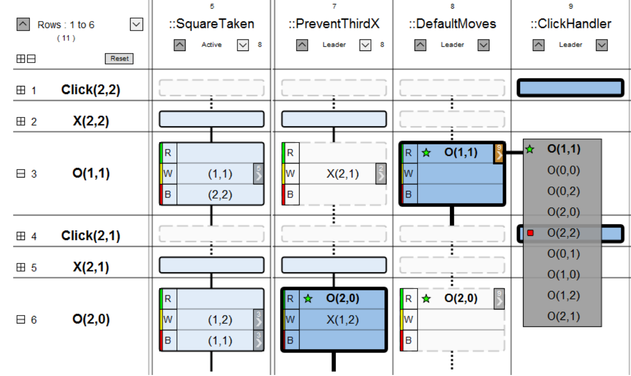

Figure 2. B-thread `PreventThirdX` takes the lead from `DefaultMoves` after events `X(2,2)` and `X(2,1)` are triggered.

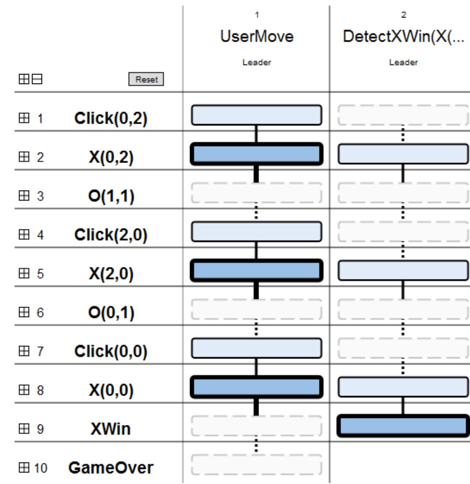

Figure 3. The b-thread instance `DetectXWin(X(0,2),X(2,0),X(0,0))` of class `DetectXWin` incorrectly declares a victory after events X(0,2), X(2,0) and X(0,0) (requested by `UserMove`) are chosen.

makes a move (number six), the leading b-thread class is `PreventThirdX`. Looking at X's two steps, we see that X almost completed the lowest row. This b-thread class has a higher priority than `DefaultMoves`, the event it requests is not blocked, and there are no other requested events from b-threads with higher priorities.

The visualization can also help find programming errors. Suppose, for example, that a user played against the computer and was notified that he/she won before completing any line. To search for the faulty b-thread, we can select only leader threads and look at the collapsed visualization, as shown in Figure 3. Following the trace upward, we can verify that the player's moves were captured correctly, but that some instance of class `DetectXWin` requested the `XWin` event following the sequence `X(0,0),X(0,2),X(2,0)`, which does not form any line. Hence this faulty instance should probably be removed.

In another game trace, shown in Figure 1, a user creates

a row of three Xs and wins. However, the game was programmed with the intent that the O player (the computer) never loses, so there appears to be a problem. To find the fault we visualize the trace, grouping b-threads by class and showing both active and leader b-threads. First, with the details hidden, one can find the "bad move": it is the last move by O, which did not intercept the two Xs in a row (see syncpoint 6), and was `O(0,2)` instead of `O(0,0)`. Why did this happen? The b-thread `PreventThirdX` indeed did request `O(0,0)`, but this event was blocked. Looking a little to the left, we see the reason: `DetectDraw` blocks O(0,0). Once this problem is pinpointed, it is easy to fix.

In general, bugs can most often be detected by looking for the first syncpoint where the behavior deviates from expectations, and working backwards from there.

IV. Related Work and Discussion

Hamou-Lhadj and Lethbridge's survey [8] covers eight different trace visualizers and their solutions for scalability. Capabilities for *trace exploration*, i.e., streamlined browsing, and *trace compression*, i.e., reducing trace size by removing some components, are discussed and compared. Our tool's trace exploration is done by sequential browsing through syncpoints or by focused scrolling through leader cells, and trace compression is based, like in most other tools, on clustering similar patterns; e.g., the behavioral-programming-specific class grouping.

Two LSC visualization tools have been built in our group in recent years. The first, the *Tracer* [9], visualizes traces of live sequence charts. It uses a hierarchical Gantt chart to follow scenarios throughout execution. The present work provides more details about the interaction between b-threads. The second LSC tool is the scenario inter-dependency visualizer (SIV) [10], which shows static and dynamic inter-dependencies between LSC scenarios using graph-based visualization. In the static view, charts are represented as nodes and event dependencies as edges. In the dynamic view, run-time dependencies are drawn as lines connecting related appearances of an event in different charts. The current tool adds a trace view, showing in a single screen the evolution of b-thread dependencies over time. Integrating these two LSC tools with BPJ appears to be an interesting future endeavor.

An interactive visualization of time-ordered events was introduced in Extravis [11]. It displays a summary time-line view and a more detailed circular view of call relationships between structural elements in a given time range. In our current work, b-thread relationships are shown using icons rather than crossing edges, and the vertical dimension communicates details about the dynamic flow of scenarios.

Trace visualizers for multi-thread applications, which are not behavior-based, such as [12], use a similar display of time and threads indicating if threads are run, blocked (e.g., due to mutual exclusion) or are waiting for an event. In a recent extensive literature survey [13] Cornelissen et al. suggested that "The importance of understanding multi-threading behavior is not reflected by the current research body". It may be interesting to see if features of our tool (e.g., classifications and grouping) can contribute to trace visualization for classical (non-behavioral) multi-threaded applications and for aspect-oriented programs.

In addition to further evaluation, future directions of research for behavioral trace visualization include trace compression based on event-sequence patterns as in [14], trace comparison, and integration with the development environment.

Acknowledgment

The research was supported in part by an Advanced Research Grant to DH from the European Research Council (ERC), by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University.

References

[1] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, pp. 45–80, 2001.

[2] D. Harel, A. Marron, and G. Weiss, "Programming Coordinated Behavior in Java," in *ECOOP*, 2010, pp. 250–274.

[3] BPJ Visualization site. [Online]. Available: http://www.cs.bgu.ac.il/~geraw/SupWebSite/

[4] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131–174, 1996.

[5] D. Moody, "The "Physics" of Notations: a Scientific Approach to Designing Visual Notations in Software Engineering," in *Proc. of the 32nd ACM/IEEE Int. Conf. on Software Engineering*, ser. ICSE '10, vol. 2, pp. 485–486.

[6] J. Bertin, *Semiology of Graphics: Diagrams, Networks, Maps*, 1983.

[7] (2008) Raphael. [Online]. Available: http://raphaeljs.com

[8] A. Hamou-Lhadj and T. Lethbridge, "A survey of trace exploration tools and techniques," in *Proc. of the 2004 conf. of the Centre for Advanced Studies on Collaborative research*, ser. CASCON, pp. 42–55.

[9] S. Maoz and D. Harel, "On Tracing Reactive Systems," *Software and Systems Modeling*, pp. 1–22, 2010.

[10] D. Harel and I. Segall, "Visualizing Inter-Dependencies between Scenarios," in *Proc. of the 4th ACM symposium on Software visualization*, ser. SoftVis, 2008, pp. 145–153.

[11] B. Cornelissen, A.Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. van Wijk, "Execution trace Analysis Through Massive Sequence and Circular Bundle Views," *J. Syst. Softw.*, vol. 81, pp. 2252–2268, 2008.

[12] M. Kessi and J. Vincent, "Performance Monitoring and Visualization of Large-Sized and Multi-Threaded Applications with the Paje Framework," in *Proc. of the Int. Multi-Conf. on Computing in the Global Information Technology*, 2006.

[13] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, 2009.

[14] B. Cornelissen and L.Moonen, "Visualizing Similarities in Execution Traces," in *Int .Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, 2007, pp. 6–10.