

Extending Behavioral Programming for Model-Driven Engineering

Thesis submitted in partial fulfillment
of the requirements for the degree of
“DOCTOR OF PHILOSOPHY”

by

Michael Bar-Sinai

Submitted to the Senate of
Ben-Gurion University of the Negev

February 2020

Beer-Sheva

Extending Behavioral Programming for Model-Driven Engineering

Thesis submitted in partial fulfillment
of the requirements for the degree of
“DOCTOR OF PHILOSOPHY”

by
Michael Bar-Sinai

Submitted to the Senate of
Ben-Gurion University of the Negev

Approved by the Advisor:

Approved by the Dean of the
Kreitman School of Advanced Graduate Studies:

February 2020
Beer-Sheva

This work was carried out under the supervision of

Dr. Gera Weiss.

In the Department of Computer Science,

Faculty of Natural Sciences.

Research-Student's Affidavit when Submitting the Doctoral Thesis for Judgment

I, Michael Bar-Sinai, whose signature appears below, hereby declare that:

X I have written this Thesis by myself, except for the help and guidance offered by my Thesis Advisors.

X The scientific materials included in this Thesis are products of my own research, culled from the period during which I was a research student.

X This Thesis incorporates research materials produced in cooperation with others, excluding the technical help commonly received during experimental work. Therefore, I am attaching another affidavit stating the contributions made by myself and the other participants in this research, which has been approved by them and submitted with their approval.

Date: 2020-04-07 Student's name: Michael Bar-Sinai Signature: 

Acknowledgments

I could not have made it through the long journey of writing this PhD dissertation without the advice, guidance and inspiration I received from several people along the way. First, I thank my advisor, Gera Weiss, for countless hours of discussions and refinement of ideas, narratives, papers, sentences, and code, as well as anecdotes, stories, and jokes. Advising a PhD thesis seems to be a delicate art, and your support and guidance navigated my PhD journey to surprising, interesting, and hopefully useful places. I thank my PhD committee members, Ronen Brafman, Guy Shani, and David Harel, who helped me focus this dissertation during its primordial phase. This work is based on the Behavioral Programming paradigm, presented by David Harel, Assaf Marron, and Gera Weiss; I thank Assaf Marron for fruitful collaborations, discussions, and guidance, as well as for arranging research group meetings where I presented early versions of works presented here. I thank David Harel for guidance, hosting these discussions at his research group at the Weizmann Institute of Science, and for coming up with the idea that I extend my Master's thesis to a PhD one. That seemed like a crazy idea at the time. It still does.

I have collaborated with fellow researchers on several papers, some related to this dissertation, and all interesting to work on. I thank Achiya Elyasaf, Aviran Sadon, Joel Greenyer, Maryann Martone, Fiona Murphy, Alexandra Wood, David O'Brien, Rotem Medzini, Reut Shmuel, and Maor Ashkenazi.

The Computer Science Department at Ben-Gurion University was always a friendly, supportive, and creative environment. I thank Ronen Brafman, Eyal Shimony, Mayer Goldberg, Jihad El-Sana, and Arnon Sturm for many wise advices, tips, and insights.

I am fortunate to have spent several formative and delightful years at the Institute of Quantitative Social Science (IQSS) at Harvard University. I thank Latanya Sweeney for generously inviting me to the weekly group meetings she held at the Harvard Data Privacy Lab, and for continuous mentorship. Under the supervision of Mercè Crosas I learned how to build robust scientific software systems, and that such systems are a worthwhile undertaking. I also learned how important the open-source culture is for scientific research and for creating user communities around tools. This knowledge greatly influenced my work on BPjs, a tool suite presented here. I thank Mercè Crosas, Gustavo Durand, Phil Durbin, Leonid Andreev, Danny Brooke, Christine Choirat, James Honaker, Kevin Condon, Michael Heppler, Ellen Kraffmiller, Elizabeth Quigley, Stephen Kraffmiller, Tania Schlatter, Sonia Barbosa, and Robert Treacy.

The members of the Harvard University Privacy Tools Project, led by Salil Vadhan, had greatly influenced my development as a software engineer and a computer scientist. I thank Stephen Chong, Micah Altman, and Kobbi Nissim for continuous mentorship in topics ranging from computer science to academic career, and for being there when I needed them.

I thank my anonymous reviewers for their comments which greatly improved this work. They encouraged me to include additional formal analysis parts, referred me to relevant related work that I overlooked, and helped me refine some of the statements this work makes.

During my professional career I worked alongside people that introduced me to interesting concepts, encouraged me to try new ideas, trusted me with designing complex systems, and helped me when these designs failed. I thank Tari Kipnis, Omer Rapoport, Mathias Fenouil, and Hilel Itzhaki.

I thank my Mom and Dad, who taught me that the world is an interesting place. They instilled in me the love for science, engineering and experimentation. They also bought me my first computer — not an easy task at a Kibbutz in 1980s Israel — and encouraged me to try programming.

Lastly, I want to thank my wife, Karen Lee, who motivated me to return to academia for graduate studies, and provided support, advice, help and inspiration throughout the course of this PhD.

*To all who found joy in programming, and then found themselves, late at
night, fixing a live production system.*

*And to my family — hopefully you won't have to hear “we'll do this when I'm
done with my PhD” anymore.*

Contents

Abstract	xii
1 Introduction	1
1.1 Contributions	5
1.2 Dissertation Outline	7
2 Background and Related Work	8
2.1 Behavioral Programming	8
2.2 And a Bit More Formally	13
2.3 Related Work	14
3 BPjs — A Foundation for Behavioral Programming Tools	21
3.1 Background and Related Work	23
3.2 Software Framework	24
3.3 Illustrative Examples	34
3.4 Evaluation	42
3.5 Conclusions	50
4 BP, Verification, and Pancakes	52
4.1 The Pancake Maker	54
4.2 Hot Synchronization Statements	59
4.3 Formal Definition of Hot Synchronization	61
4.4 Adding Verification to the Mix	63
4.5 Related Work	79
4.6 Conclusion	81

5 Semantic Variations using BP	82
5.1 Semantic Mapping to BP	83
5.2 Discussion	84
5.3 Case Study: Semantic Variations of LSC	86
5.4 A Visual Dictionary for LSC	88
5.5 Implementation	99
5.6 Semantic Variations	101
5.7 Related Work	104
5.8 Conclusion	105
6 BP Model Driven Engineering	106
6.1 Tracking Rover Control	109
6.2 On-Board Satellite Control and Testing	116
6.3 Related Work	131
6.4 Conclusion	132
7 Conclusions and Additional Proposed Research	134
7.1 Possible Future Research Directions	135

List of Tables

3.1	Average time required for BPjs to fully traverse the state space of a robot-in-a-house simulation b-program. Measurements were taken using OpenJDK 18.9 (Java 11), on a 2.9 GHz MacBook Pro. Each measurement was repeated 10 times.	43
3.2	Average time required for a robot to take 1000 steps in a house simulation (milliseconds). Measurements were taken using OpenJDK 18.9 (Java 11), on a 2.9 GHz MacBook Pro with 16G of RAM (out of 32) allocated to Java. Each measurement was repeated 10 times.	45

List of Figures

2.1	Execution cycle of a b-program. B-threads run in parallel.	
	To communicate with each other or with the environment, b-	
	threads submit synchronization statements to the b-program's	
	event arbiter. After submitting a statement, the submitting	
	b-thread is paused. When all b-threads have submitted their	
	statements, the b-program is said to have reached a synchro-	
	nization point. At this point, the event arbiter selects an event	
	that was requested and not blocked, and resumes all the b-	
	threads that requested or waited for that event.	9
2.2	Two code modules, and state spaces of the b-programs they	
	compose. Module <i>Hello World</i> consists of two b-threads: one	
	requesting the HELLO event, and the other requesting WORLD. A	
	b-program comprised of these b-threads alone is free to choose	
	the order in which the events occur (left state-space). Adding	
	the <i>Order</i> module eliminates the erroneous left branch, as its	
	order b-thread blocks WORLD until HELLO is selected.	12

3.1	<i>Interrupting events</i> do not add expressive power to BP, but do make the code more readable, by allowing programmers to declaratively label events as a reason for b-thread termination. Both above b-threads monitor an oven temperature and request that the heater is turned on when the temperature drops below 220 degrees. Both terminate when the baking is over (event done). However, the b-thread on the right expresses the different semantics of temp and done declaratively, whereas the b-thread on the left expresses them using control-flow. This makes the b-thread on the right is more readable, and less prone to programming mistakes.	26
3.2	Main BPjs packages, classes, and their inter-relationship. The model package is used to describe a b-program. Model execution is performed using the execution package. The analysis package is used to analyze b-programs (e.g., for verification). Host applications use BPjs by instantiating a BProgram object, and passing it either to a BProgramRunner for execution or to a DfsBProgramVerifier for analysis. Client code can alter execution and analysis behavior by providing custom implementations of interfaces such as EventSelectionStrategy (which implements the event selection algorithm), or ExecutionTraceInspection (for detecting violations during analysis). Detailed class and package documentation is available at [15]	29

3.3 BPjs used as a b-program runtime engine (left) and as a analysis engine (right). Dark blocks are closed for modification (except by means of altering their code), and light blocks can be supplied by the client code in order to alter system behavior. BPjs supplies reasonable default implementations for all light blocks, except for the b-program itself. The b-program (b-threads supplied by the programmer) interacts with its **BProgram** infrastructure using **bp**, an interface object placed in the application scope by BPjs. Client code can specify which event selection strategy the **BProgram** will use for selecting events requested by the b-program threads. The b-program and the **BProgram** supporting it are used both during execution and analysis. For program execution, the host application can monitor the b-program using a listener interface. It can send data to the b-program by queueing events into its external event queue. During analysis, the host application can specify which trace inspections should be executed, and how visited states should be stored. The host application monitors the analysis progress through a listener interface, similar to the execution use-case. When a violation is found, the host application is informed through the listener interface, and can decide whether analysis should continue or not.

33

3.4 Execution cycle of a b-program under BPjs. Execution begins with BPjs running the JavaScript source as a regular script (left). During this initial execution, the code registers JavaScript functions and b-threads. When the initial execution is completed, BPjs begins to execute the b-threads through the regular BP-cycle of concurrent b-thread progression, synchronization, and event selection (right).

35

3.5	Screenshots of the robot-in-a-house simulation application, built using a model-driven engineering approach, with BPjs as the model runtime and analysis engine. Clockwise, from top left: a model, described using the floor plan description language presented in Subsection 3.3.2; a running simulation; a detected safety violation (robot in trap); a detected liveness violation (robot stuck in a hot area indefinitely).	36
3.6	State space of a robot in a house simulation b-program, simulating the house shown in Figure 3.5. Nodes are synchronization points, edges are events. Hot synchronization points are marked with a thick orange line. Violating states, where a b-thread declared a failed assertion, are marked with a hexagonal shape. This graph was automatically generated by a tool that uses BPjs (see [6]) and later adjusted manually to fit this page.	41
3.7	Ratio between the duration required for a full state space traversal and various state space graph metrics. Ratio between traversal duration and state count, edge count, or their sum, increases polynomially with the size of the state space. However, the ratio between traversal duration and $ states \times edges $ remains constant as the state space grows. The two types of visited state storages (hashed and full) behave similarly.	44
3.8	BPjs talk at Devovx Belgium 2018. Devovx is a developer-led conference focusing on JVM languages. The fact that a talk about BPjs was accepted and well attended supports our view that BPjs, and BP in general, are ready for non-academic use. Photo: James Birnie's blog (http://www.jamesbirnie.com/2018/11/devovx-belgium-and-two-talks.html).	49

4.1	A computer-controlled pancake batter mixer, where mixture flow to the mixing bowl is controlled by a b-program. The host program, a traditional Java program, converts its inputs to events and feeds them to the model for processing. Additionally, the host program translates event selections made by the model to real-world actions.	55
4.2	The transition system of the plain pancake batter b-program. Ovals represent synchronization points (states), while transitions represent events. The text in the ovals indicates the number of doses of each mixture added up to that point. Possible program runs traverse the graph, starting at state (0,0) and eventually reaching state (5,5). All states are reachable if running only Listing 4.1. When adding the strict arbiter b-thread in Listing 4.2, only the green/3-line states are reachable. When replacing the strict arbiter with the RangeArbiter of Listing 4.4, the blue/2-line states also become reachable.	64

4.3	State-space graphs of four b-program, demonstrating different structures involving hot synchronization points. All b-programs contain two b-threads throughout. Progression is from top to bottom, where each row is a synchronization point. At each point, hot b-threads appear in dark red, and non-hot b-threads appear in light blue. <i>Cold Cycles</i> contain at least a single synchronization point where all b-threads are non-hot. These cycles do not breach any liveness requirements. <i>B-Program Hot Cycles</i> are composed of synchronization points that all have at least one hot b-thread (individually, each b-thread may be cold at some synchronization points during the cycle). These cycles may or may not violate liveness requirements, depending on the context and the requirement definition. <i>B-Thread Hot Cycles</i> consist of synchronization points in which at least one b-thread is always hot. If said b-thread represents a liveness requirement, a program execution that forever follows this cycle violates the requirement that b-thread models. <i>Hot termination</i> is a violation of a safety property, as the b-program run is finite. However, the requirement being violated may be described better in liveness terms.	73
5.1	Defining executable semantics for diagrammatic languages using querying and mapping. Queries select constructs from a diagram. Selected constructs are mapped to one or more <i>Construct Agent B-threads (CABs)</i> . Run together, those CABs generate a valid execution of the original program.	84

5.2	LSC describing a basic move in a card memory game. After the user flips two cards, a beep is emitted. If the cards are different, they are flipped back face-down. The first two events may or may not happen, and are thus <i>cold</i> (blue). The three subsequent events must take place once the first two events have occurred, and are thus <i>hot</i> (red). The Sync construct forces the Beep to occur following the second click	87
5.3	Visual Dictionary: an LSC Lifeline	90
5.4	Visual dictionary: Synchronization constructs. (I) SYNC, (II) Assignment, (III) Cold Condition, (IV) Hot Condition	91
5.5	Visual Dictionary: Message types. (I) Cold, executed message. (II) Hot, Executed message. (III) Cold, monitored message. (IV) Hot, monitored message.	93
5.6	Visual Dictionary: Sub-charts. (I) Loop. (II) Alternatives.	96
5.7	A simple LSC with a loop, and its XML representation.	100
6.1	A b-program used as a model in Model-Driven Engineering. During verification (left), the model is analyzed by a verifier. Additional b-threads may be added for simulating the system environment, adding assumptions in order to limit verification search space, and adding additional requirement b-threads not already contained in the model. During runtime (right), the b-program is put in a b-program runner, which serves as an adapter between the model and an ordinary software system. The same b-program is used for both runtime and verification, which eliminates the bugs that may emerge during model translation.	108

6.2 A race condition caught by verification. The leader rover (black ovals) moves forward at a steady pace. The faulty tracker rover (blue ovals) follows it. To align itself behind the leader, the tracker must move closer to it, and change its orientation. It starts by turning (frame A), and then moves forward at full speed. When it gets close enough to the leader (frame B), it must both slow down and turn. At this point, the `NotTooClose` b-thread requests a `GoSlowGradient` event, while the `SpinToTarget` b-thread requests a turn. The b-program selects the turn event first. By the time the `GoSlowGradient` event is selected, the leader is further away from the rover, and so the power field of the `GoSlowGradient` event is smaller than expected (frame C). 115

6.3 The hybrid lab layout. MATLAB and STK, running on a simulation computer, simulate the satellite's dynamics, environment, and various hardware components. The OBC and the simulation computer interact using the native electrical on-board connectors used by the real hardware that the PC simulates. . . . 130

Abstract

Scenario-based programming (SBP) is an evolving programming paradigm that enables the creation of systems with complex behavior from interaction of simple scenarios. The interacting scenarios can monitor and restrict each other, spawn new scenarios, and interact with the system’s environment. Among the advantages of SBP amenity to incremental development and to formal analysis, and the ability to align the code with system requirements.

The focus of this work is a variant of SBP called Behavioral Programming (BP). Scenarios in BP describe sequential system behaviors, and are thus called *behavior threads*, or *b-threads*, for short. B-threads interact with each other and with their environment using events, which they can request, wait-for, and block. A program written in BP — called a *b-program* — groups b-threads and provides them services such as event selection and environment interaction. B-programs are written using ordinary code, making them easy to adopt for practicing software engineers, as they allow using standard tools, such as existing text editors and version control systems.

This dissertation aims to bring BP closer to general software engineering. Its core is an extensible and modular definition of BP, based on a holistic approach for analysis and execution of b-programs. This definition — backed by a working tool suite called *BPjs* — views b-programs as models, and so b-program execution becomes an operation to be performed on a model, similar to model checking or state-space analysis. By defining extension points such as event selection algorithms and execution trace inspections, the proposed definition allows practitioners and researchers to easily extend, refine, and tailor BP to their needs, without having to deal with the infrastructure required for implementing a BP platform from scratch.

To enable an integrated system architecture, where b-programs take care of high-level decisions, and ordinary code handles simpler, low-level tasks, we introduce a bi-directional communication protocol between b-programs and ordinary software systems. This protocol facilitates reading input and feeding

it to the b-program, and listening to decisions made by the b-program and translating them to actions, such as actuating engines, writing to a database, or updating a GUI presentation layer.

Said BP model is not necessarily the input from the user. We present a way of translating models created in other modeling languages, diagrammatic or textual, to BP. The proposed methodology queries the native description of a model and translates the query results to b-threads. We demonstrate this approach on the Live Sequence Charts (LSC) language, and show how it can be used to examine semantic variations of the language being translated.

The end goal of this work is to enable BP-based model driven engineering — a development methodology where analyzable b-programs can be embedded in ordinary systems. To this end, we improved the analysis and verification of b-programs. Specifically, we describe verification of safety properties, and an extension of BP that allows verification of liveness properties. We identify two types of liveness violations, and examine a case where a safety property is better presented in liveness terms. We back these concepts by working code — additions to our BPjs suite discussed above. We report on two projects where we used BPjs as the modeling core in a model-driven engineering setting. The first project is an autonomous rover required to track a leading rover while maintaining safe distance. The second project is an on-board control software for a satellite. In the latter, we also demonstrate how BP can be used to orchestrate system tests.

Keywords: Behavioral Programming, Control, Reactive Systems, Executable Modeling, Model Driven Engineering (MDE), Model Based Software Engineering (MBSE), Model Analysis, Verification.

Chapter 1

Introduction

Controlling complexity is the essence of computer programming.

– Brian Kernighan [65]

Even without the hardware layers that they are built upon, software systems are immensely complex structures. The programmer must control this complexity in a way that allows for the development and delivery of systems that fulfill client requirements, and are free of critical bugs, while keeping to project schedules and budget limits. Or — if worst comes to worst — to provide an accurate prediction of the additional time and resources required to deliver the system.

The principal task of the software engineer runs much deeper than delivering a single version of a software system. Software engineers must organize their code such as to facilitate straightforward adaption to changing requirements, and to enable software updates by way of evolution from earlier versions — rather than by rewriting the system from the ground up. These imperatives direct software design toward separating the underlying code into clear modules, defining interfaces, creating documentation, and other associated activities so as to make a codebase manageable and reusable.

As complexity is central to software engineering, it is not surprising that one of the foundational texts of the field deals with complexity itself. In *No Silver Bullet* [16], Brooks distinguishes between *inherent complexity*, stemming

from the problem a program is attempting to solve, and *accidental complexity*, emerging from the technicalities of implementing the program itself with the available technologies, hardware limitations, and chosen design. The paper, published in 1986, goes on to claim that since many of the challenges of accidental complexity had been addressed by advances such as high-level programming languages and operating systems, there will be no “silver bullet” creating an order-of-magnitude improvement in speed or quality of software engineering for at least another 10 years. Notably, Brooks did not claim that the challenge of accidental complexity had been completely resolved; rather, he suggested that handling it required less than 90% of development effort. Proven correct, in 1995 Brooks extended his prediction for another 10 years, noting that *Complexity is the business we are in, and complexity is what limits us* [17, C. 17].

The work presented here focuses on engineering the inherent complexities of reactive software systems. These systems, as defined by Harel and Pnueli [52], react continuously to stimulations, whether from their environment or from internal sources. Reactive systems are very common, and include web servers, IoT devices, and autonomous vehicles. Because these systems maintain a complex state, their behavior at any given point is an outcome of multiple concurrent internal processes, together with all the input accumulated during a given run. This property makes it hard for humans to develop and test such systems, as humans tend to think linearly [33].

Behavioral Programming (BP) [50], a programming paradigm focused on creating complex system behavior by interleaving relatively simple linear scenarios, offers an elegant means of building such systems. A program written in BP — called a *b-program* — consists of numerous linear scenarios (called *b-threads*). B-threads communicate using a simple event-based protocol: they can request, wait-for, and block events. The BP runtime repeatedly selects events that are requested and not blocked (see Section 2.1).



I got my first “grown up” job in software engineering after graduating in

2006. While I took a mandatory class about formal requirements at university, at the time I didn't think they were relevant for practitioners. I knew about UML class diagrams and understood that UML had other diagrams too; but nevertheless, I wasn't sure what exactly they were, and in any case viewed them as useful doodles and not as a visual notation with formal semantics. However, even though I programmed in Java — a high-level programming language by all accounts — it didn't feel to me that its level was “high” enough for the engineering challenges I was facing. Granted, unlike programmers in the 1950s and 60s and 70s, I did not have to deal with assembly-level accidental complexities like jumps, labels, little/big endians, and direct memory addressing. Unlike my father, I have never had to gather up a program from the floor of a bus, after its punch cards spilled out of their box. I was always free to code using lists, objects, and maps directly, and do so on a very convenient workstation.

But the software itself, a multi-million dollar flight management system for commercial airlines — its requirements alone spanning thousands of pages, maintained by a specialized team — was not about objects and lists. It was about passengers, check-in procedures, baggage, and seat upgrade flows. Object-Oriented programming does do a good job in describing the data aspect of these concepts, because it allows programmers to code using terms from the problem domain. As an example, one can work with an object called **Flight**. But, when faced with the challenge of describing system behavior — e.g. the processes that must take place if a passenger who has ordered a special meal, and has missed her connecting flight, must be upgraded and moved, luggage and all, to a flight with a compatible partner airline in order to minimize arrival delay — we programmers had to fall back on writing algorithms to run on data structures, not fundamentally different than low-level procedural languages.

When multiple requirements pertained to the same concepts, we were obliged to manually compose these requirements in code when possible — if at all possible, given that requirements often contradicted each other. When requirements changed (as they often did, and still do), the code then had to

be updated to reflect the new changes, while still fulfilling the un-changed requirements — again, only if at all possible, given that changes of this nature may very well introduce contradictions into the system.

In subsequent positions, and in other projects, my sense about the insufficiency of high-level programming languages remained. When I decided to return to academia and study Behavioral Programming, I was looking forward for a break from the trenches of software engineering. The general plan was to go on an intellectual journey, guided by interests rather than applicability or relevance. As I write this, some 8 years on, I realize that this journey actually led me to an exploration of this gap between the high-level aspirations of current programming languages and the level of abstraction required for describing requirements for system behaviors. Worse still: I fear that the research presented here might be both relevant and applicable (see Chapters [5](#) and [6](#)).

The research presented here extends BP, explores its potential as a tool for executable modeling, and enables integrating BP-based models in traditional software systems. Models help address inherent complexity directly, by describing system behavior at the requirement level. BP is a good foundation for modeling behavioral system requirements, because b-threads resemble the informal textual requirements that they describe (this property of BP is known as *requirement alignment*). Thus, future flight management systems — and other systems as well — can be created with their requirements described in a formal, executable manner, as part of their code. This will allow formal requirement analysis, minimize client-programmer misunderstandings, and — because models can serve as a high-level documentation — reduce code-documentation mismatches. Most importantly, it creates an elegant framework for programmers to describe what the system is required to do — and what not — and have the BP runtime figure a way of composing these requirements, if and when possible. If requirements contradict each other, automated analysis tools will be able to detect the contradiction and highlight it in an informative manner (see Chapter [4](#)).

Building on Brooks’ distinction between accidental and inherent complex-

ity, we can divide the history of software engineering into two distinct phases, and to propose a third. This proposed division is based on the terms that are used in code:

1. *Machine Terms*: Programs written for machines. Code directly handles low-level, machine-specific issues such as registers, navigation in unstructured code, memory offsets in records, etc. At this stage, programmers deal mostly with accidental complexity.
2. *Computer-Science Terms*: Programs written for other programmers, translated into machine code using optimizing compilers. The code handles data structures and algorithms. At this stage, programmers deal with both accidental and inherent complexity. Most instances of accidental complexity occur during attempts to describe system behavior.
3. *Requirement Terms*: Programs written (in part) for clients or domain experts. The code contains requirements in a direct, executable form. Accidental complexity in describing system behavior is significantly reduced.

In [39], Harel responds to Brooks’ “No Silver Bullet” by proposing executable modeling as an approach that would provide a “truly major improvement” in software engineering. Brooks agrees that “modeling does address the essence”, and thus the use of executable models could be “revolutionary” [17, C. 17]. Both agreed, however, that executable modeling in and of itself will not provide the “silver bullet” of Brooks’ definition by 1996.

Per Brooks, his 1986 paper was not lamenting a bleak future, but rather presenting an optimistic statement, the understanding that “there is no royal road, but there is a road”. I hope the work presented here helps paving it.

1.1 Contributions

This dissertation makes the following contributions:

1. An extensible definition of BP, backed by an execution and analysis engine (BPjs). Beyond execution, the developed engine treats b-program execution state as data, enabling program analysis.
2. An integration protocol for embedding BP models in traditional software systems.
3. An extensible definition for event selection algorithms, allowing the usage of algorithm implementation during both execution and analysis. Using this definition for proposing new event selection algorithms.
4. New BP idioms:
 - Synchronization point metadata, allowing b-threads to pass relevant data to the event selection algorithm.
 - Adding an **assert** instruction to BP. This allows for the verification of safety properties during program analysis, and emergency stops during execution.
 - Adding a “hot b-thread” concept to BP. This allows for the verification of liveness properties.
 - An interrupting event set. This is a runtime-level support facility for common BP design pattern, where b-threads specify a set of events that, if any one of them is selected, the b-thread will quit.
 - Adding a unix-like **fork** instruction to BP. This is different from the existing b-thread registration instruction, in that it duplicates the calling b-thread’s heap and stack.
5. The implementation and integration of model-checking algorithms for BP, in the BP analysis framework. Model Checking of safety and liveness properties of BP models. The analysis engine presented here is expressive enough to verify LTL expressions [90].
6. A definition of two classes of liveness violations for b-programs, and an additional case where safety violations are better presented as liveness violations.

7. A methodology for Model Driven Engineering using BP as the model layer, based on the above contributions.
8. A method for describing the executable semantics of formal languages using BP. Introducing the concept of *construct agent b-threads* (CABs), b-threads that represent constructs in the defined language during execution and analysis.
9. A set of documentation, examples, and two case studies of the proposed BP definition and of BPjs. These examine and document the above contributions.
10. A new approach for state serialization and comparison. During program analysis, this approach allows visited state detection, and the exploration of multiple execution branches. To this end, we contributed code to Mozilla’s Rhino, the JavaScript engine our system is based on.

1.2 Dissertation Outline

The rest of this dissertation is organized as follows: Chapter 2 discusses the foundations on which this work relies, including an introduction to Behavioral Programming. It additionally presents related work. Chapter 3 introduces our extensible definition to BP, our added idioms, and our BP execution and analysis engine (BPjs). Chapter 4 describes how a b-program can be verified using BPjs, and discusses various violation types. Chapter 5 proposes a way of using BP for defining the executable semantics of formal languages. Chapter 6 presents two use cases where BPjs is used as the model execution layer in model-driven engineering: a satellite, and an autonomous rover tracking another rover. Chapter 7 concludes and proposes future research directions.

Chapter 2

Background and Related Work

This chapter presents the backdrop for the work reported in this dissertation. We start by describing Behavioral Programming (BP) [80], the programming paradigm on which my work is based. Next, we take a look at existing BP libraries and tools. Behavioral Programming is a type of Scenario-Based Programming (SBP), and so our next stop will broaden our view and examine other scenario-based languages. Finally, we consider other executable modeling and verification tools, in an attempt to present some of the context in which SBP lives.

2.1 Behavioral Programming

BP was introduced in 2010 by Harel, Marron, and Weiss [50, 51], as a programming paradigm focusing on reactive systems — systems that continuously react to external and internal stimuli [52]. In BP, programs are composed of multiple threads of behavior. These threads, called *b-threads*, run concurrently, communicating with each other using a synchronized, event-based protocol. A program composed of b-threads is called a *b-program*.

The b-thread synchronization protocol works as follows: when a b-thread wants to synchronize with its peers, it submits a synchronization statement to the b-program’s event arbiter. This statement declares the events the b-thread requests to be selected, the events that it waits for (but does not request), and

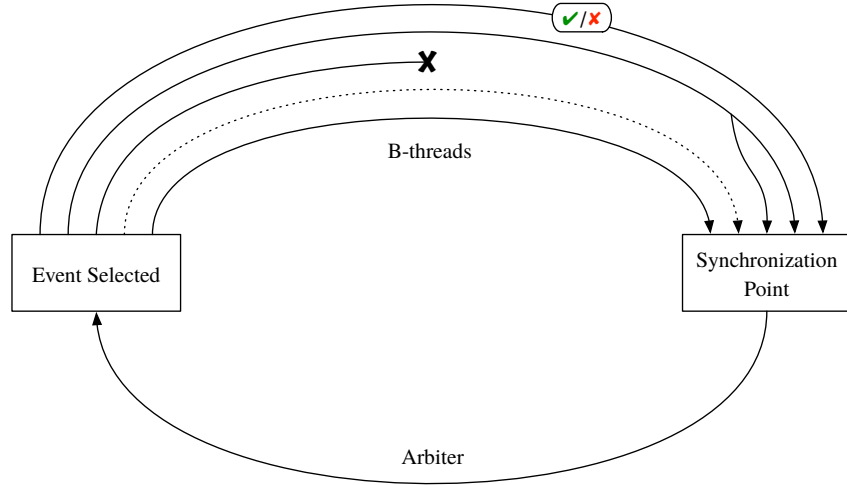


Figure 2.1: Execution cycle of a b-program. B-threads run in parallel. To communicate with each other or with the environment, b-threads submit synchronization statements to the b-program’s event arbiter. After submitting a statement, the submitting b-thread is paused. When all b-threads have submitted their statements, the b-program is said to have reached a synchronization point. At this point, the event arbiter selects an event that was requested and not blocked, and resumes all the b-threads that requested or waited for that event.

the events that it would like to block (forbid, prevent from being selected). After submitting the statement, the b-thread is then paused by the BP runtime. When all the b-threads have submitted their statements, the b-program is said to be at a *synchronization point*. When it arrives at such a point, the b-program’s arbiter selects an event that has been requested and is not blocked. It then resumes all the b-threads that requested or waited for that event. The other b-threads will remain paused until an event that they requested or waited for is selected. Their synchronization statements will be carried over to subsequent synchronization points, a process that continues until they are resumed. When running, b-threads can terminate, spawn new b-threads, and invoke assertions. Figure 2.1 illustrates this cycle.

We noted earlier that synchronization statements specify the events that a b-thread requests, waits for, and blocks. Interestingly, these specifications are of different types. The requested events must be iterable, so the arbiter can

generate candidate events for selection. The waited-for and blocked event sets can be specified logically, using a predicate¹. In other words, these are pure mathematical sets, and not necessarily iterable data structures.

2.1.1 A Simple B-Program

Following the great tradition set forth by Brian Kernighan in a 1974 internal memorandum about the C programming language, we will demonstrate the basics of BP using a “hello, world” program. The code used here is for BPjs [10], a tool for writing and analyzing b-programs written in JavaScript. We developed BPjs as part of this research project; it is described in depth in Chapter 3.

Our goal in this subsection is to create a b-program that will select two events: `HELLO` and `WORLD`, in this order. To this end, we will consider a few programs that may or may not be correct.

The “hello, world” version in Listing 2.1 begins by defining the two events involved. It then registers a single b-thread, which requests these two events in sequence. Since it is the only b-thread in its b-program, these events are not blocked, and are the only available events for the arbiter to choose from. Thus, the arbiter selects them in the order that they have been requested, and the program produces the correct event trace.

Some BPjs-specific technicalities should be clarified before we progress to the examples below. The code is written in JavaScript. The semantics of the programs are those of JavaScript, except for concurrency: JavaScript is mostly a single-threaded language, whereas code written in BPjs is inherently concurrent, and may be paused and resumed at synchronization points. All functionalities stemming from Behavioral Programming are accessed through the `bp` object. B-threads are JavaScript functions that take no arguments, and are registered as b-threads with `bp`.

Let us now elaborate the example. We have decided to split the program, based on its sub-tasks. In this case, one sub-task will be to emit the `HELLO`

¹Formally, a function $f : Event \rightarrow Boolean$

Listing 2.1: Sequential version of “hello, world” program

```
1 var HELLO = bp.Event("hello");
2 var WORLD = bp.Event("world");
3
4 bp.registerBThread(function(){
5   bp.sync({request:HELLO});
6   bp.sync({request:WORLD});
7 });
```

Listing 2.2: A modularized version of “hello, world” program. This program may omit the events in incorrect order, since the b-threads are not coordinated.

```
1 bp.registerBThread("bt-hi", function(){
2   bp.sync({request:HELLO});
3 });
4
5 bp.registerBThread("bt-world", function(){
6   bp.sync({request:WORLD});
7 });
```

event. The other sub-task will be to emit the **WORLD** event. Listing 2.2 contains a b-program broken into two b-threads, one for each sub-task. This structure is intuitive, very cohesive, and works — about 50% of the time.

The explanation for the frequent failure of the the program in Listing 2.2 is that the b-threads are not coordinated. That is: at the first synchronization point of the split b-program, both events are requested, but neither event is blocked. Thus, the arbiter is free to choose either of these events. In particular, it can choose **WORLD** before it chooses **HELLO**, which would make the b-program emit the events in the wrong order. The left state-space graph in Figure 2.2 illustrates this state of affairs.

To fix this, we add the **hello world patch** b-thread (Listing 2.3) to the split b-program. This b-thread blocks **WORLD** until **HELLO** is selected, thus forcing the b-program’s event arbiter to select the events in the correct order. The right graph of Figure 2.2 shows the state-space of the patched program.

The last example demonstrates BP’s suitability for incremental, modular software development. We have changed the overall behavior of our b-program by adding new code, rather than by changing existing one. This addition can be done and undone without amendments to the rest of the program — useful

Listing 2.3: A b-thread imposing correct exdecution order on the b-program in Listing 2.2

```

1 bp.registerBThread("hello world Patch", function(){
2   bp.sync({waitFor:HELLO, block:WORLD});
3 });

```

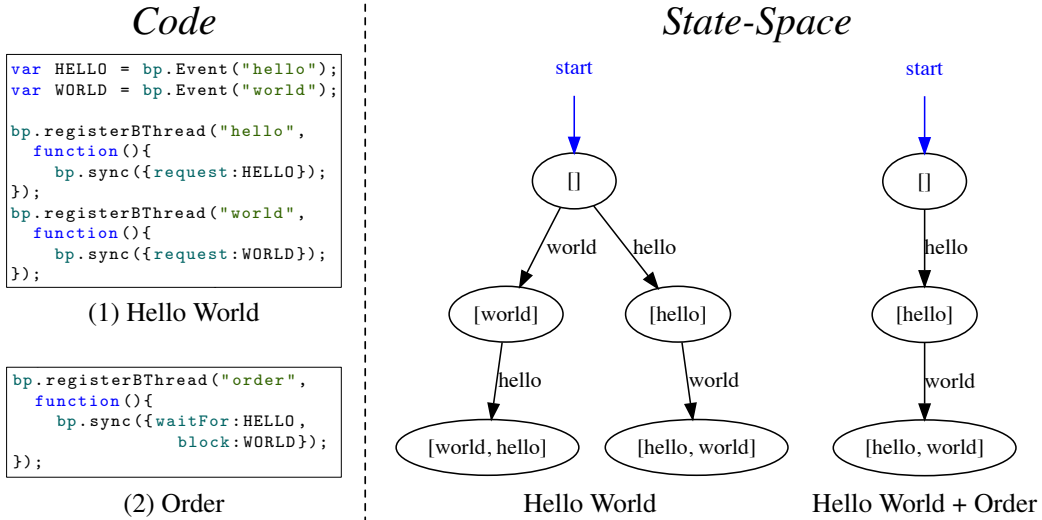


Figure 2.2: Two code modules, and state spaces of the b-programs they compose. Module *Hello World* consists of two b-threads: one requesting the **HELLO** event, and the other requesting **WORLD**. A b-program comprised of these b-threads alone is free to choose the order in which the events occur (left state-space). Adding the *Order* module eliminates the erroneous left branch, as its **order** b-thread blocks **WORLD** until **HELLO** is selected.

Listing 2.4: A b-thread adding the event `incremental` between the events `HELLO` and `WORLD`.

```

1 bp.registerBThread("add incremental", function(){
2   bp.sync({waitFor:HELLO});
3   bp.sync({request:bp.Event("incremental"), WORLD});
4 });

```

if we decide that the order of the events is not important, or that it needs to be set using another algorithm.

To further demonstrate BP's incrementality, we will conclude by updating the trace of our b-program, such that it becomes `<HELLO, incremental, WORLD>`. We can do this by simply adding the `add incremental` b-thread in Listing 2.4 to the b-program. No other change is needed.

2.2 And a Bit More Formally

This section proposes a formal mathematical foundation for b-program execution analysis². The definition we give here adapted version of definition given in [50]. This adaptation — adding *waitFor* as an explicit part of a b-thread definition rather than having it implicitly defined via a transition system — creates a definition is more closely aligned with BPjs' syntax.

Recall that a labeled transition system is defined as a quadruple $\langle S, E, \rightarrow, init \rangle$, where S is a set of states, E is a set of events, \rightarrow is a transition relation contained in $(S \times E) \times S$, and $init \in S$ is the initial state [64]. The runs of such a transition system are sequences of the form $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \dots \xrightarrow{e_i} s_i \dots$ where $s_0 = init$, and $\forall i = 1, 2, \dots, s_i \in S, e_i \in E$, and $s_{i-1} \xrightarrow{e_i} s_i \in \rightarrow$.

Definition 2.2.1 (b-thread). A b-thread is a tuple $\langle S, E, \rightarrow, init, R, W, B \rangle$ where $\langle S, E, \rightarrow, init \rangle$ forms a labeled transition system, $R: S \rightarrow 2^E$ associates a state with the set of events requested by the b-thread when in it, $W: S \rightarrow 2^E$ associates a state with the set of events waited for by the b-thread when in it, and $B: S \rightarrow 2^E$ associates a state with the set of events blocked by the b-thread

²The title of this section is of course a reference to sections in Harel and Marelly [48] with a similar role.

when in it. A b-thread enters a state s_i when requesting synchronization (for BPjs, this is done by invoking `bp.sync`).

The runs of a set of b-threads (a b-program) are defined below. This is the definition given in [50], with the addition of W .

Definition 2.2.2 (b-program execution). The runs of a set of b-threads (called a b-program):

$$\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, W_i, B_i \rangle\}_{i=1}^n$$

are the runs of the labeled transition system $\langle S, E, \rightarrow, init \rangle$, where $S = S_1 \times \dots \times S_n$, $E = \bigcup_{i=1}^n E_i$, $init = \langle init_1, \dots, init_n \rangle$, and \rightarrow includes a transition

$$\langle s_1, \dots, s_n \rangle \xrightarrow{e} \langle s'_1, \dots, s'_n \rangle$$

if and only if:

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

and, for all $i = 1, \dots, n$:

$$\underbrace{(e \in W_i(s_i) \cup R_i(s_i) \Rightarrow s_i \xrightarrow{e} s'_i)}_{\text{advancing b-threads}} \bigwedge \underbrace{(e \notin W_i(s_i) \cup R_i(s_i) \Rightarrow s_i = s'_i)}_{\text{non-advancing b-threads}}$$

This definition allows for more than one run of a given set of b-threads, depending on the order in which events are selected from the set of requested and not blocked events. This non-determinism is one of BP's strengths, as it allows BP to specify partial order over events.

2.3 Related Work

A number of BP runtime and analysis implementations exist, spanning a variety of programming languages of different types:

BPJ [50], a BP library for writing behavioral programs using Java, is the first BP library. Over time, a few mutually incompatible versions have emerged, as researchers forked the code to adapt it to specific needs. A b-program verifier for detecting safety violations was presented in [46]. For technical reasons, it cannot work beyond Java 5.

BPC [42] is a framework for writing behavioral programs in C++. It offers customizable events and event selection. BPC also supports time constraints — if an event is not selected within a given timeout, b-threads are resumed, with no event selected. BPC is not designed to be embedded in host applications, but rather to run as a standalone framework. This affects the way BPC deals with inputs, as a b-thread that must wait for input will delay synchronization. BPC supports limited verification, based on examining the state graph of each b-thread, then composing and analyzing the program graph using an external tool.

In [83], Marron, Weiss, and Weiner implement BP in Blockly [100], and use this implementation to create a user interfaces for web applications. This implementation is interesting, in that it achieves b-thread parallelism using co-routines and a single OS thread. In this respect, it differs from BPJ and BPC, which use a single OS thread per b-thread, and from BPjs (Chapter 3), which uses a pool of OS threads and a continuation mechanism.

Another BP-based library for developing front end for web applications is react-behavioral³ by Luca Matteis. It relies on EcmaScript 6’s generators for pausing b-threads after submitting synchronization requests.

Weiner, Weiss, and Marron present a proof-of-concept of BP working in Erlang in [102]. Here, each b-thread runs in its own Erlang thread. Synchronization statements are submitted to a central process, which also performs event selection and reporting.

BP is one form of Scenario-based Programming (SBP) [23, 48], a paradigm for creating executable models by combining independent, modular scenarios. Each scenario describes a different aspect of overall desired and undesired system behaviors. At runtime, an execution engine interweaves the various

³<https://github.com/lmatteis/react-behavioral>

scenarios. These compositional semantics enable the alignment of a software system’s structure with the requirements it is expected to fulfill. Additionally, SBP is intuitive, as it aligns well with how people think about complex systems [33].

SBP was introduced through the language of Live Sequence Charts (LSC) and its Play-Engine implementation [48]. LSC is a diagrammatic programming language that extends classical message sequence charts, mainly through a universal interpretation and two modalities: must/may, and monitor/execute. Developed by Damm and Harel [23], the first implementation of this language is Play-Engine [48]. A UML-compliant variant is implemented by the PlayGo tool [47, 77]. The semantics of the PlayGo version, which differ slightly from Play-Engine’s, are described in [82].

An LSC system consists of scenarios and objects. Each scenario describes a facet of the system’s behavior, and is described in a live sequence chart (abbreviated: LSC). Overall system behavior is the result of the concurrent execution of all the LSCs that the specification contains.

ScenarioTools [36] is a modeling workbench, based on the Scenario Modeling Language (SML), a textual extension of LSC. ScenarioTools can combine SML with graph transformation to model structural changes in the described system. Models created with ScenarioTools are executable, and can be analyzed through simulation. IOSM-K, a more recent version of SML, implements SML using the Kotlin programming language. Similar to the Blockly BP implementation mentioned above, it relies on co-routines for pausing its scenarios.

Scenario-based Programming research results cover, among others, run-time lookahead (smart play-out) [45], synthesis [55, 69, 78], model-checking [46] (see also Chapter 4), compositional verification [41], specification realizability analysis [55, 79], abstraction-refinement mechanism [62], test generation [74], automatic correction [44, 62], synchronization relaxation [40], and polymorphic scenarios [75].

Scenario-Based Programming can inter-operate with other programming paradigms. Harel, Marron, Nissim, and Weiss integrated SBP with fuzzy

logic [49]. In [81], Marron, Hacoheh, Harel, Mulder, and Terfloth integrate SBP with Statecharts.

One key difference between LSC-based languages (including SML variants) and BP is that in LSC, events are messages sent from one object to another. That is, LSCs model conversations between objects. When modeling an “ambient” event, with no inherent source or destination, LSC programmers often use self-messages on a dedicated lifeline. In BP, on the other hand, events are selected rather than sent. Accordingly, they do not have a source or a destination. In order to implement LSC using BP, one can add source and destination data to events (see Section 5.3).

Our description of BP above noted that the arbiter selects an event that is “requested and not blocked”. This leaves a lot of wiggle room for the arbiter to determine *which* event, requested and not blocked, should be selected. The first BP implementation, BPJ, selected events based on b-thread priority. With the original BPJ, each b-thread has a unique priority; the arbiter selects the event that not blocked, and requested by the b-thread with the highest priority. Subsequently, this was relaxed by adding a priority distance parameter, called “b-thread epsilon”, to b-programs. After the relaxation, two b-threads were considered as having the same priority if the difference between their unique priorities was less than the b-program’s b-thread epsilon.

Other event selection algorithms have also been proposed. Smart play-out, presented by Harel et al. [45], uses formal verification methods to inform event selection during runtime. An adaptive learning algorithm for optimal event selection was developed by Eitan and Harel [26], and a planning-based approach to event selection was developed by Segall and Harel [54]. When no look-ahead is involved in the event selection algorithm, BP execution is comparable to naïve LSC’s play-out, as defined for Play Engine [48]. In particular, the system might choose an event sequence that violates system requirements, even when a non-violating sequence is available.

BPC, PlayGO, and BPjs take a more modular approach to event selection, by applying the Strategy design pattern [32] to the event selection problem. In this approach, the BP infrastructure allows the programmer to specify the

event selection algorithm to be used.

BP’s event selection is essentially an agreement protocol, allowing multiple b-threads (or, more generally, scenarios) to agree on an event. This does not imply that the event must be explicitly requested, however. In [63], Katz, Marron, Sadon, and Weiss propose a way of generating agreeable events on the fly, using synchronization requests containing rich logical constraints, and constraint solvers (such as SAT or SMT).

2.3.1 SBP Systems and Their Environment

BP, and SBP in general, were created for building reactive systems. Because such systems react to their environment, the scenarios/b-threads must be able to read data from it. In LSC, a special lifeline called ENV is responsible for representing the environment. “Regular” lifelines communicate with it by sending and receiving messages, similar to regular lifelines. Some LSC extensions allow making safety and liveness assumptions on the environment [78]. In BP, events carrying data from the environment are either generated by b-threads which access external resources and translate them to events, or by non-BP parts of the system that pass these events through an API.

In order to simplify system scenario design, SBP systems often adopt the assumption of *Logical Execution Time* [58], which claims that the execution rate of internal events is much higher than that of external events; thus, a burst of internal events in-between two external events (also called *super-step*) will take zero time, relative to the external dynamics. When a system needs to perform a long-running computation, it does so using a non-BP thread, which sends the computation result to the running SBP system asynchronously. Similar design patterns are used in GUI systems to prevent the user interface from “freezing”.

In real-time applications, the Logical Execution Time assumption must be validated, e.g., based on the sampling rates of sensors, hardware speed, and expected computation duration. Our experience shows that this assumption is realistic, as we used it to implement control software for a quad copter and

for a satellite model (see Section 6.2).

2.3.2 Beyond Scenario-Based Programming

Scenario-Based Programming is a form of executable modeling — an approach that describes systems using formal system models with executable semantics [99]. Other executable modeling languages include Executable UML (xUML) [84] and Foundational UML (fUML) [86]. Both use a subset of the UML diagrams, and an action language for creating models that can be executed and simulated. For example, xUML uses UML class diagrams to define classes, and UML state machine to define instance behavior. The idea of creating a model from complementary formalisms is called *heterogeneous modeling* [72].

Umple [73] takes a different approach to executable UML modeling, by using code generation and language augmentation. It adds UML concepts to mainstream languages, and can present UML models in code and graphically, with bi-directional updates. An Umple developer can therefore define relations between classes using UML notation; Umple will generate the required code when the model is realized. Umple uses UML Statecharts for modeling dynamic system behavior. These state machines can be analyzed at the model level, using execution scenarios [2]. Umple can generate code for Java, C++, and PHP, making it platform-independent. Notably, Umple is implemented in Umple, which shows its strength.

The above body of work focuses on system modeling. That is, the above languages and systems attempt to capture a useful abstract view of the system they they model, and assert its properties. JavaPathFinder (JPF) [57], developed at NASA and open-sourced in 2005, takes a different approach: it verifies Java programs directly.

JPF is a versatile modular Java virtual machine, aimed at program analysis and verification. The core JPF system can verify a Java program by running all possible thread interleaving combinations, and enumerating over all its random decision points. JPF is a mature project with a large user community.

Its many modules support advanced features, such as symbolic execution⁴ and constraint solver integration⁵.

However, as with any verifier, JPF cannot escape the state explosion problem. Even though JPF supports partial order reduction, verifying concurrent programs — which requires verification of many possible interleaving combinations — is resource intensive. In Subsection 3.4.1, we compare the performance of JPF and BPjs in verifying similar behavioral programs.

⁴<https://github.com/SymbolicPathFinder>

⁵<https://github.com/psycopaths/jconstraints>

Chapter 3

BPjs — A Foundation for Behavioral Programming Tools

A preliminary version of this paper was presented in Models 2018 [10]. The text presented here is a Journal version of that conference paper, edited for length and repetition.

This chapter presents BPjs, an extensible engine for running and analyzing behavioral programs. BP was first introduced in 2010 [50]. Since then, a number of tools for executing b-programs have been introduced, some which also support various forms of verification (see Section 2.3). While some of these tools are more mature than others, none present a holistic approach for the analysis and execution of b-programs as does BPjs. BPjs treats b-programs as data structures; consequently, model execution becomes a specific case of model analysis. BPjs analyses b-program using direct execution, thus also enabling the analysis of b-programs developed for extended versions of the basic BP model.

BPjs was designed as a platform for creating tools based on BP. Its design was informed by the existing body of work on BP, in that our objective is that BPjs will be able to support most of these works. In the few specific cases that a work disagreed on some aspect of BP (e.g. event selection), we created a common interface around the disagreement area, and added extension points

to BPjs. As a result, BPjs implements an extensible, generalized version of BP. In particular, BPjs supports:

1. Extensible runtime design, which allows users to extend event types and event selection strategies without changing BPjs itself.
2. A well-defined communication protocol between traditional software systems and b-programs, which allows for the embedding of BPjs in traditional applications, e.g., for creating systems using a model-driven engineering approach.
3. Extensible analysis engine, which allows users to specify state and trace inspections, and to identify single or multiple violations in a given b-program. The analysis is done such that models developed with user-defined extensions can also be analyzed.
4. Extensions to the synchronization statements, which allows b-threads to submit an optional parameter that can be used by either the event selection algorithm or the analysis engine.

Our objective with BPjs is to create an infrastructure that will allow developers to focus on the application at hand, or on the extension mechanisms they want to develop, without the need to be concerned about core BP execution technicalities. As an example, we describe here how we implemented the first tool, which supports the verification of liveness properties in b-programs.

To borrow a term from Gosling’s introduction of Java [34], BPjs was designed as a “blue collar” tool suite, to make the working programmer’s job easier. It handles the infrastructure required for BP, and allows programmers to focus on the challenge they need to solve. We have used BPjs in a number of projects, demonstrating thus that it can be used as a BP infrastructure in developing complex and reliable reactive systems.

Good infrastructure is not purely technical issue, given that it also facilitates the discovery of theoretical results. For example, while developing the BPjs-based verification tool, I found that liveness violations in scenario based

programs can be divided into two types, with significantly differing semantics. To the best of my knowledge, this distinction had not been identified hitherto.

3.1 Background and Related Work

Numerous BP runtime and analysis implementations exist (see Section 2.3). However, only two of these can be considered as attempts to create a robust research infrastructure (as opposed to a proof of concept).

BPj [50] allows for writing behavioral programs using Java. Unlike BPjs, BPJ was not designed to be an embedded, modular framework. Over time, a few mutually incompatible versions emerged, as researchers forked the code to adapt it to specific needs. A b-program verifier was presented in [46]. However, it can only detect safety violations, and for technical reasons cannot work beyond Java 5. Our work on BPj informed many of the decisions we took while designing BPjs. In particular, the decisions to use a modular design, an IDE-agnostic project format, and an open, collaboration-friendly code repository all emerged from our experience with BPJ.

BPC [42] is a framework for writing behavioral programs in C++. Like BPjs, it offers customizable events and event selection strategies. However, and unlike BPjs, BPC is not designed for embedding b-programs in host applications, and supports limited verification, using model transformation. BPC supports time constraints, a feature not currently supported by BPjs.

BPC and BPj both use an OS thread for each b-thread, making b-threads expensive. This structure encourages programmers to reduce the amount of b-threads used in their b-programs — a serious design limitation, the equivalent of requiring Object Oriented programmers to limit the amount of objects in their programs. BPjs, on the other hand, allows multiple b-threads to use a single OS thread. Where required, BPjs can run a b-program with numerous b-threads using a single OS thread. While this is ostensibly a technical difference, it allows programmers using BPjs to use as many b-threads as required by their design. Subsection 3.4.1 presents a case where a b-program successfully runs on BPjs, but exhausts computer resources on BPJ.

3.2 Software Framework

The overarching goal of BPjs is to provide a common infrastructure for extension and usage of BP. To this end, it provides facilities to execute, analyze, and embed b-programs. BPjs supports an open-ended, parameterized version of BP, where programmers can alter event selection strategies, event types, and inspections used in the verification process. BPjs also provides an interface, using the listener design pattern, for host application to act based on selected events.

3.2.1 Additions to the BP Paradigm

In common with most programming paradigms, Behavioral Programming defines a limited number of basic concepts, which gives a lot of freedom to implementors of concrete systems. Our goal in creating BPjs was to support as much of the existing body of work as possible, but without sacrificing the extensibility that would underpin future research and use. To this end, we implemented the following additional concepts in BPjs. Some of these concepts are orthogonal to BP semantics, while others had to be embedded in it.

Assertions B-threads can make assertions while being executed, by calling `bp.ASSERT(expr,expln)` (where `expr` is a boolean expression, and `expln` is a string containing a human-readable message. If `expr` evaluates to **false**, the assertion fails, and the next synchronization of the b-program is marked as invalid. If a b-program arrives at an invalid state, it is considered to be violating its requirements. During analysis, the trace leading to the invalid state will be presented as a counter example. During execution, the violating b-program will be halted. BPjs will report the failed assertion to the host application, which allows the host to recover in an informed way.

Hot Synchronization Points A b-thread signals that it must eventually advance beyond a synchronization point by marking the point as *hot*. Through analysis, BPjs can report runs where a b-thread is trapped in a hot synchro-

nization point indefinitely. Marking a synchronization point as hot is done by calling `bp.hot(true).sync(...)`. The term *hot* is borrowed from Live Sequence Charts [23], where it has a similar meaning.

Synchronization Point Parameters In classic BP, b-threads submit only a synchronization statement when they synchronize with their peers. BPjs allows b-threads to pass an optional, second parameter while synchronizing. This parameter is part of the b-thread’s synchronization statement, and as such is available to all the algorithms working with the state of a b-program at a synchronization point. These include event selection algorithms and execution trace inspectors. One example of the use of this parameter is an event selection strategy, which allows b-threads to specify their priority at any synchronization point.

Interrupts BPjs allows b-threads to specify *interrupting* events, in addition to specifying requested, waited-for, and blocked ones. If a b-program selects an event that a b-thread has specified as interrupting, then that b-thread is terminated. B-threads may register a “final words” function, which will be invoked in such cases.

Interrupting events is a convenience feature, and does not add strength to the language. Rather, they are an incorporation of the following design pattern into BPjs’ syntax: When an event make a b-thread no longer required, this b-thread waits for it, together with the events it waits for as part of its normal operation. When an event the b-thread waits for is selected, that b-thread first checks whether said event makes it irrelevant. If so, the b-thread terminates. Otherwise, the b-thread responds to the event (see Figure 3.1). We noted that this was a reoccurring pattern, and thus decided to support it at the library level. This support improves code readability by providing programmers with a declarative way of labeling which events require termination, and which are waited-for as part of the normal b-thread scenario.

<pre> 1 bp.register("monitor-1",function(){ 2 while (true) { 3 var evt=bp.sync({ 4 waitFor:[temp, done]}); 5 if (evt === done) return; 6 if (evt.data.value < 220) { 7 bp.sync({request:heaterOn}); 8 } 9 } 10 }); </pre>	<pre> 1 bp.register("monitor-2",function(){ 2 while (true) { 3 var evt=bp.sync({waitFor:temp, 4 interrupt:done}); 5 if (evt.data.value < 220) { 6 bp.sync({request:heaterOn}); 7 } 8 } 9 } 10 // </pre>
--	--

Figure 3.1: *Interrupting events* do not add expressive power to BP, but do make the code more readable, by allowing programmers to declaratively label events as a reason for b-thread termination. Both above b-threads monitor an oven temperature and request that the heater is turned on when the temperature drops below 220 degrees. Both terminate when the baking is over (event **done**). However, the b-thread on the right expresses the different semantics of **temp** and **done** declaratively, whereas the b-thread on the left expresses them using control-flow. This makes the b-thread on the right is more readable, and less prone to programming mistakes.

Fork B-threads can call `bp.fork`, which, as in C, forks the b-thread into two b-threads. Forking differs from the situation where a b-thread registers a new b-thread, in that `fork` also duplicates the call stack of the thread.

3.2.2 Software Architecture

For BPjs to analyze b-programs, it must view them as models or as data structures. This does not prevent BPjs from executing programs. Rather, it makes program execution another operation performed on a b-program. To reflect this view, the BPjs design (shown in Figure 3.2), is divided into three main packages:

model

This contains implementations of the main BP concepts. The `BProgram` class captures the context in which b-threads run — it includes an event selection algorithm (implemented using the `EventSelectionStrategy` class), and an external event queue, through which host applications can send data to b-threads. This package also contains the base class for

events (called `BEvent`), and `EventSet`, which groups `BEvents`. `EventSet` is not a data structure, but rather an abstract predicate. This allows programmers to capture abstract concepts, such as “all events whose name starts with X.” Perhaps surprisingly, this package does not contain a `BThread` class. Instead, it has a class called `BThreadSyncSnapshot`, which captures the state of a b-thread at a synchronization point. From the BPjs point of view, an executing b-thread is a series of `BThreadSyncSnapshots`. This design feature makes it easier to hypothesize about b-program state-spaces.

execution

This contains classes for executing b-programs and monitoring their runs. The `BProgramRunner` class, which executes `BProgram` instances and reports back to the host application, resides in this package.

analysis

This contains classes for traversing and inspecting b-program state-spaces, and reporting results. The `ExecutionTrace` class describes the possibly infinite execution of a b-program, consisting of a series of b-program states and the events that connect them. `ExecutionTraceInspection` is the base class for inspecting these traces for violations. While `DfsBProgramVerifier` is currently the only object that performs state-space traversal, we expect other such objects to be added to this package in the future.

Internally, BPjs uses Mozilla Rhino [85] to execute JavaScript code. Rhino supports capturing continuations of running programs. Captured continuations can be inspected and serialized. During model execution, BPjs uses this feature to pause b-threads at synchronization points, and to allow a single Java thread to execute multiple b-threads. During analysis, BPjs uses this feature to traverse a b-program’s state-space as follows: When a b-program arrives at a synchronization point, BPjs stores all its b-thread continuations. BPjs then resumes the b-program multiple times — once for each event that

is requested and not blocked at that point. This allows BPjs to analyze b-programs directly, without the need to pass through an intermediate model transformation phase.

BP specifies that at each synchronization point, the b-program should select an event that is *requested and not blocked*. This requirement gives BP engines the freedom to choose a specific event at synchronization points where more than a single event meets this criteria. Since many event selection algorithms have been proposed (see Section 2.3), and since our objective in the creation of BPjs was to construct a common platform for BP, we decided to make the event selection algorithm pluggable. To this end, we have defined the `EventSelectionStrategy` interface, with support for two methods. The first method accepts a b-program at a synchronization point, and returns a set of selectable events; the second accepts a set of selectable events, and chooses a single event from them. This functional separation, between identifying the events that can be selected and actually selecting an event, allows BPjs to use the same event selection strategy object for both program analysis and execution. BPjs ships with four strategies:

1. *Random* This strategy selects at random an event that is requested and is not blocked.
2. *Prioritized by B-Threads* This strategy allows the programmer to assign a priority to each b-thread. When presented with a set of events that are requested and are not blocked, it selects the event with the highest priority requested by the b-thread. B-thread priorities are not required to be unique.
3. *Prioritized by Synchronization Points* B-threads may add an “irritation factor” metadata to their synchronization statements. When presented with a set of events that are requested and not blocked, this strategy will select an event at random from the subset of events that are requested, not blocked, and whose requesting synchronization statement has the maximal irritation factor.

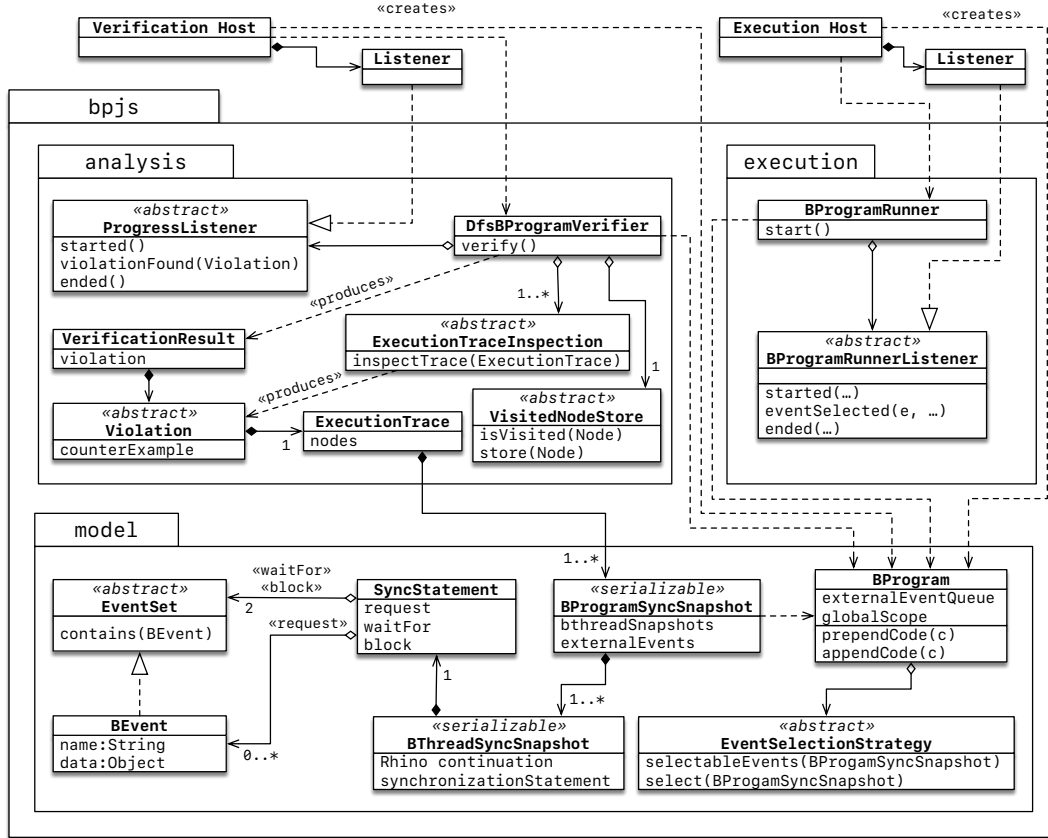


Figure 3.2: Main BPjs packages, classes, and their inter-relationship. The `model` package is used to describe a b-program. Model execution is performed using the `execution` package. The `analysis` package is used to analyze b-programs (e.g., for verification). Host applications use BPjs by instantiating a `BProgram` object, and passing it either to a `BProgramRunner` for execution or to a `DfsBProgramVerifier` for analysis. Client code can alter execution and analysis behavior by providing custom implementations of interfaces such as `EventSelectionStrategy` (which implements the event selection algorithm), or `ExecutionTraceInspection` (for detecting violations during analysis). Detailed class and package documentation is available at [\[15\]](#).

4. *Ordered Events* A b-thread can request multiple events by passing an event array in its synchronization statement. While other strategies treat this array as a set of events, this strategy treats it as a priority queue. Thus, for each b-thread, it will only consider selecting the first requested and not blocked event.

Students and researchers working with BPjs have developed other strategies. Examples include a round-robin strategy that attempts to be fair with regard to b-threads wait times; a strategy which sets priority according to event type; and a few strategies based on machine learning.

3.2.3 Software Functionalities

BPjs can be used in two manners: as a BP runtime embedded in a host Java program, or as a tool for b-program analysis (see Figure 3.3). When used as a runtime (see Listing 3.1), the host application creates a `BProgram` object, and provides it with JavaScript source code. It then uses a `BProgramRunner` to execute the `BProgram` instance. The host application sends data to the b-program by enqueueing events in the b-program’s external event queue. It reads data from the b-program by listening to events selected by the b-program, through a listener interface provided by the `BProgramRunner`.

Host applications can alter b-program behavior by providing a custom event selection strategy, or by directly manipulating the b-program’s global scope. Another way of altering the behavior of a b-program is by adding additional b-threads to its source code — `BProgram` supports this approach too, through methods that allow for appending and prepending code.

When using BPjs as an analysis/verification library (see Listing 3.2), the host application begins by creating a `BProgram` instance, in the same way as in the model execution use case. Depending on the properties being verified, the host may add additional b-threads to detect violating states, simulate an environment, or limit search space. The host then passes the b-program to a `DfsBProgramVerifier` for verification. The verifier traverses the b-program’s state space by executing the b-program, capturing a continuation of its b-

Listing 3.1: BPjs used as a model execution engine. The host application instantiates a `BProgram` object, and passes it to a `BProgramRunner` for execution. The host receives updates about the b-program’s progress through a listener interface, and can pass data to it by enqueueing events in its external event queue. Prior to starting program execution, the host can write data directly to the b-program’s global scope, and can specify the event selection strategy to be used.

```

1 BProgram bprog = new StringBProgram(source);
2 bprog.setEventSelectionStrategy(new SomeCustomSelectionStrategy());
3 bprog.putInGlobalScope("answer",42);
4 bprog.appendSource(createExtraBThreads(...));
5 BProgramRunner rnr = new BProgramRunner(bprog);
6 rnr.addListener(new BProgramRunnerListener(){
7     @Override
8     public void eventSelected(BProgram bp, BEvent theEvent){
9         // handle selected event
10    }
11    // additional b-program lifecycle handling
12});
13rnr.run();
14...
15bprog.enqueueExternalEvent( new BEvent("data from host") );

```

threads at each synchronization point. By invoking these continuations multiple times — each time with a different event that was requested and not blocked — the verifier examines all possible advancement options available to the b-program at each of its synchronization points.

Because the state-space is a connected graph (and not a tree), a b-program can reach a certain state through multiple traces. Thus, during analysis, when BPjs arrives at a state, it needs to check whether it has already visited this state. To this end, BPjs uses a visited state storage object, implementing the `VisitedStateStore` interface. BPjs ships with three state storage implementations: a full state storage, a storage that uses a hash value of stored states, and a “forgetful” visited state storage, useful for programs whose state space is known to be a tree.

Listing 3.2: BPjs used as a model analysis engine. The host application instantiates a `BProgram` object and passes it to a `DfsBProgramVerifier` for analysis. The host receives updates on the progress of the analysis through a listener interface. If a violation is found, the host is informed, giving it the option to decide whether the analysis should continue or not. Before starting the analysis, the host can write data directly to the b-program’s global scope, specifying which event selection strategy should be used. The host can adjust various analysis aspects, such as the maximal depth of the DFS traversal, the set of inspections used, and how visited states are stored.

```

1 BProgram bprog = new StringBProgram(source);
2 bprog.setEventSelectionStrategy(new SomeCustomSelectionStrategy());
3 bprog.putInGlobalScope("answer",42);
4 bprog.prependSource(createExtraBThreads(...));
5 DfsBProgramVerifier vfr = new DfsBProgramVerifier();
6 vfr.setMaxTraceLength(1024);
7 vfr.setVisitedStateStore(new BThreadSnapshotVisitedStateStore());
8 vfr.addInspection(ExecutionTraceInspections.FAILED_ASSERTIONS);
9 vfr.setProgressListener(new DfsBProgramVerifier.ProgressListener(){
10     @Override
11     public boolean violationFound(Violation vln, DfsBProgramVerifier vfr){
12         // handle violation
13         return true; // true continues the analysis; false terminates it.
14     }
15     // additional analysis lifecycle handling
16 });
17 final VerificationResult res = vfr.verify(bprog);

```

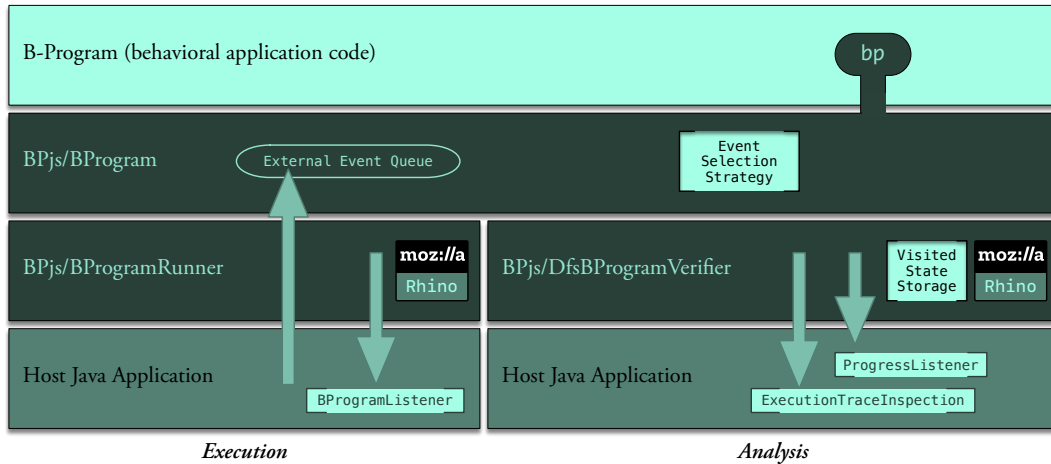


Figure 3.3: BPjs used as a b-program runtime engine (left) and as a analysis engine (right). Dark blocks are closed for modification (except by means of altering their code), and light blocks can be supplied by the client code in order to alter system behavior. BPjs supplies reasonable default implementations for all light blocks, except for the b-program itself. The b-program (b-threads supplied by the programmer) interacts with its **BProgram** infrastructure using **bp**, an interface object placed in the application scope by BPjs. Client code can specify which event selection strategy the **BProgram** will use for selecting events requested by the b-program threads. The b-program and the **BProgram** supporting it are used both during execution and analysis. For program execution, the host application can monitor the b-program using a listener interface. It can send data to the b-program by queueing events into its external event queue. During analysis, the host application can specify which trace inspections should be executed, and how visited states should be stored. The host application monitors the analysis progress through a listener interface, similar to the execution use-case. When a violation is found, the host application is informed through the listener interface, and can decide whether analysis should continue or not.

Listing 3.3: A “hello world” program, consisting of three b-threads.

```
1 var HELLO = bp.Event("hello");
2 var WORLD = bp.Event("world");
3
4 bp.registerBThread("bt-hi", function(){
5   bp.sync({request:HELLO});
6 });
7 bp.registerBThread("bt-world",function(){
8   bp.sync({request:WORLD});
9 });
10 bp.registerBThread("hello world Patch", function(){
11   bp.sync({waitFor:HELLO, block:WORLD});
12 });
```

3.3 Illustrative Examples

This section demonstrates the use of BPjs as an embedded BP runtime and as a model checker. We will begin by revisiting the “hello world” program from Section 2.1.1 — this time focusing on the BPjs-specific aspects. After this, we will present a more complex example, also using Section 3.4, to evaluate BPjs’ performance.

3.3.1 Hello Revisited World

The code in Listing 3.3 shows the complete program constructed in Section 2.1.1. To recap: the code begins with the definitions of two events, and then registers three b-threads. BPjs programs are written using JavaScript, with a `bp` object added to their global scope. The `bp` object is the proxy for all actions related to behavioral programming. Here, it is used for event definition, b-thread registration, and synchronization. B-threads synchronize by calling `bp.sync`. This method accepts two arguments: a synchronization statement and an optional synchronization parameter, as discussed above. Synchronization statements are JavaScript objects with fields for requested events (zero, one, or event array), and waited-for, blocked, and interrupting event sets.

It should be noted that the code for “hello world” does not refer to any b-program. In particular, no b-program is ever instantiated or started. BPjs does this outside the program’s code, which assumes that a single, global b-program

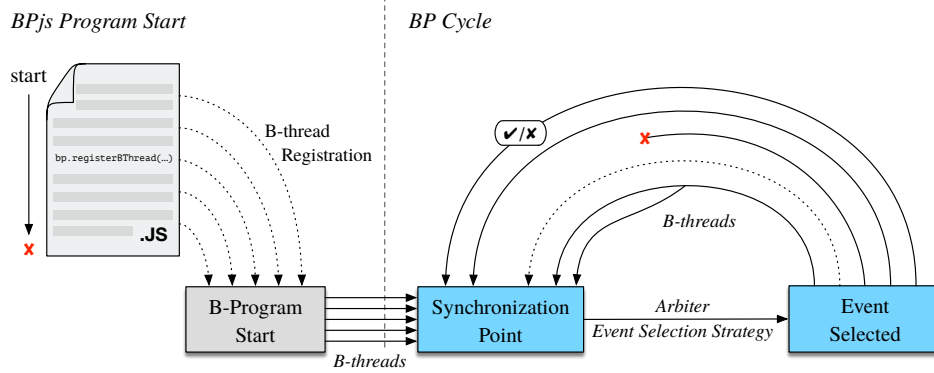


Figure 3.4: Execution cycle of a b-program under BPjs. Execution begins with BPjs running the JavaScript source as a regular script (left). During this initial execution, the code registers JavaScript functions and b-threads. When the initial execution is completed, BPjs begins to execute the b-threads through the regular BP-cycle of concurrent b-thread progression, synchronization, and event selection (right).

exists. BPjs executes a program by first running its JavaScript file(s) to completion, and collecting b-threads registered by calls to `bp.registerBThread` during this initial run. It then runs the b-threads concurrently, which starts the classic BP cycle. Figure 3.4 shows this process.

The decision to have BP code assume the existence of a global b-program simplifies the code, which is an important language design goal. It does not prevent system designers from using multiple b-programs in a system, since this assumption is only valid for the JavaScript code, and not the for host Java code. Thus, system designers are free to concurrently run multiple b-programs at the host level, using multiple `BProgramRunners`.

3.3.2 A Robot in a House

We now turn to a more complex example, demonstrating how BPjs can be used for execution and analysis in a model-driven engineering approach. To this end, we present an application for visually simulating the movement of a robot in a house. For this demonstration, we defined a house using a floor plan consisting of a two-dimensional array of cells. There are two types of

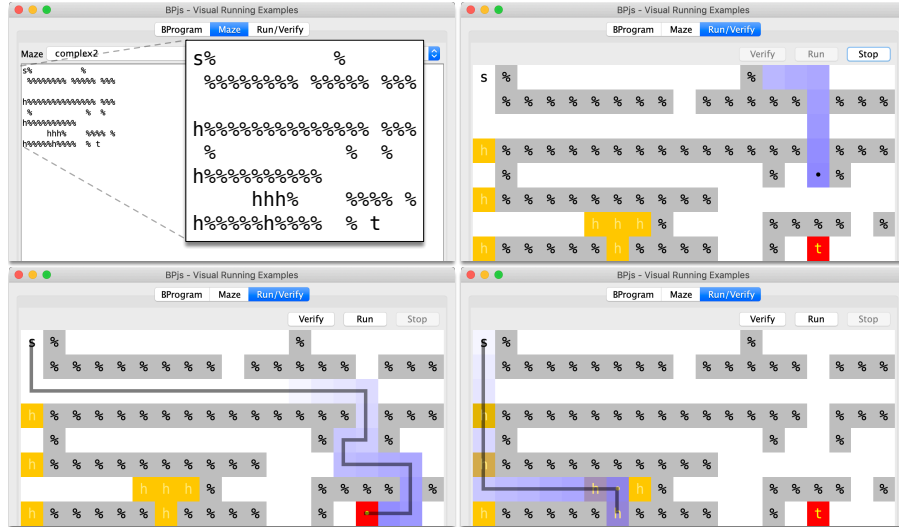


Figure 3.5: Screenshots of the robot-in-a-house simulation application, built using a model-driven engineering approach, with BPjs as the model runtime and analysis engine. Clockwise, from top left: a model, described using the floor plan description language presented in Subsection 3.3.2; a running simulation; a detected safety violation (robot in trap); a detected liveness violation (robot stuck in a hot area indefinitely).

cells: walls and spaces. In the floor plan, space cells are of three sub-types: regular, hot, and trap. Our objective is to ensure that the robot will never enter a trap cell, and that it should not stay on hot cells indefinitely (more formally: a robot should visit non-hot cells infinitely often).

The application shown in Figure 3.5 allows users to create a floor plan of a house, using a domain-specific language. With a floor plan, the application can simulate robot movement, and detect whether the robot will fall into a trap or stay in a hot area indefinitely. This application was designed using the model-driven engineering approach. With this approach, business logic is described and articulated through a model, rather than through standard imperative code. Compared to code, which focuses on computation, models offer a higher level of abstraction, closer to the problem domain. Here, the model is a b-program created by parsing the floor plan. The host application (written in Java) uses BPjs to execute or analyze the house model. All simulation decisions are made by the model — the host is only responsible for the creation and

presentation of the model.

Our floor plan description language is based on ASCII art drawings. Some characters have unique semantics: spaces denote space cells, `t` denotes trap cells, and `h` denote a hot cell. The robot’s starting point is denoted by `s` — otherwise a regular space cell. All other characters denote walls. Generating a b-program model for a house described in this language amounts to traversing the floor plan ASCII drawing, adding appropriate b-threads for each of its characters. Listing 3.4 shows the b-thread templates for the generation of b-threads for various space cell types. The templates are parameterized with cell coordinates.

We introduced this approach, where b-threads are used to define the executable semantics of formal language constructs, in [9]. We explore this further in [5].

The state-space of a “robot in a house” b-program is shown in Figure 3.6. The floor plan used to generate this b-program is the same as the plan used for the simulations in Figure 3.5.

Our model generates a random robot walk in a house as follows: individual space cells repeatedly wait for an entry event to any of their neighboring cells (`adjacentCellEntry` event set in Listing 3.4). When such entry occurs, the space cell requests an entry event with its own coordinates. While doing so, it also waits for any other entrance — in effect, making the space cell in question cancel the request if the robot chooses to enter another cell. The robot walk is started by a `Start` b-thread, requesting an entry event to a specific cell.

A *trap* cell is a space cell. That is, a `Space` b-thread with its coordinates is present in the b-program. In addition to this b-thread, a `Trap` b-thread with its coordinates is also present. The `Trap` b-thread announces that the robot was trapped immediately after the robot entered its cell, by requesting the `ROBOT_TRAPPED_EVENT` and blocking all other events. This means that `ROBOT_TRAPPED_EVENT` will be the only selectable event at the synchronization point immediately after the robot enters the trap.

Like trap cells, *hot* cells are space cells with an added b-thread. With these cells, a `Hot` b-thread waits for an entry event to the cell it represents. When

Listing 3.4: Code for generating b-threads for the house simulation program described in Subsection 3.3.2. Rows 1-13 contain definitions of events and event sets that are used by simulation b-threads. The four `addXXX` functions, parametrized with cell coordinates, add b-threads for the various types of cells to the b-program. In order to create a b-program that simulates a robot moving through a specific house, the floor plan parser (not shown, available in [6]) traverses an ASCII drawing of said floor plan, and invokes `addXXX` functions according to the character it encounters.

```

1 function enterEvent(c,r) {
2   return bp.Event("Enter (" + c + "," + r + ")");
3 }
4
5 function adjacentCellEntries(col, row) {
6   return [enterEvent(col + 1, row), enterEvent(col - 1, row),
7           enterEvent(col, row + 1), enterEvent(col, row - 1)];
8 }
9
10 var anyEntrance = bp.EventSet("AnyEntrance", function(evt){
11   return evt.name.indexOf("Enter")==0;});
12 var cellWait = [anyEntrance, ROBOT_TRAPPED_EVENT];
13
14 function addSpaceCell( col, row ) {
15   bp.registerBThread("Space(c:"+col+" r:"+row+")", function(){
16     while ( true ) {
17       bp.sync({ waitFor:adjacentCellEntries(col, row) });
18       bp.sync({ request:enterEvent(col, row),
19               waitFor:cellWait });
20     });
21 }
22
23 function addTrapCell(col, row) {
24   bp.registerBThread("Trap(c:"+col+" r:"+row+")", function(){
25     while ( true ) {
26       bp.sync({ waitFor:enterEvent(col, row) });
27       bp.sync({ request:ROBOT_TRAPPED_EVENT,
28               block:bp.allExcept( ROBOT_TRAPPED_EVENT ) });
29     });
30 }
31
32 function addStartCell(col, row) {
33   bp.registerBThread("Starter(c:"+col+" r:"+row+")", function() {
34     bp.sync({ request:enterEvent(col,row) });
35   });
36 }
37
38 function addHotCell( col, row ) {
39   bp.registerBThread("Hot(c:"+col+" r:"+row+")", function(){
40     while ( true ) {
41       bp.sync({ waitFor:enterEvent(col, row) });
42       bp.hot(true).sync({ waitFor:cellWait });
43     });
44 }

```

such an event is selected, it then waits for an entrance to another cell, using a *hot* synchronization point. As discussed in Subsection 3.2.1, hot synchronization points indicate to the BPjs analyzer that the submitting b-thread must eventually advance beyond said point. In order for this to happen, an entry event to another cell must eventually be selected. This means that the robot must, at some point, leave the hot cell.

If a floor plan has two adjacent hot cells, a robot may leave one hot cell and immediately enter another. In this case, a different **Hot** b-thread will become hot at the next synchronization point. In the model presented here, a run in which a robot moves in an infinite loop of hot cells will violate the requirement that a robot must eventually enter a non-hot cell. Thus, we deploy an execution trace inspector to look for cycles in the program’s state-space where, in each state, there is at least one hot b-thread. Other inspectors may look for cycles where a single b-thread is hot throughout the cycle, ignoring the overall picture.

Model Execution In order to simulate the robot’s movement around the house, the Java host application uses a **BProgramRunner**, in a manner similar to the steps presented in Listing 3.1. The host starts by composing b-program source code from the floor plan parser and the floor plan created by the user. It then creates a **BProgram** instance using the generated code, and passes this to an instance of **BProgramRunner**. It then starts the runner, and listens to events selected by the b-program. When an **Enter(x,y)** event is selected, the host updates the displayed floor plan by moving the robot icon to cell (x,y) . When the user clicks a floor plan cell, the host application enqueues an entry event to that cell in the b-program’s external event queue. Chapter 6 presents an in-depth assessment of the use of BPjs as an embedded model-execution engine.

Model Verification A house model is verified against two formal requirements: That the robot never falls into a trap, and that the robot will not get stuck in a hot area indefinitely. The host application uses BPjs to verify the

Listing 3.5: A b-thread capturing the requirement “A robot should not fall into a trap”.

```

1 bp.registerBThread("Robot not falling into trap", function(){
2   bp.sync({waitFor:ROBOT_TRAPPED_EVENT});
3   bp.ASSERT(false,"The robot fell into the trap.");
4 });

```

model, in a manner similar to the process described in Listing 3.2. The host starts by generating a `BProgram` instance, as in the execution use-case. In addition to the code used for execution, the application adds a b-thread that will cause a false assertion when the `ROBOT_TRAPPED_EVENT` is selected (Listing 3.5). The host proceeds to instantiate a `DfsBProgramVerifier` object, and passes the newly generated `BProgram` object to it. The host then specifies the relevant inspections for the verifier to use (one looking for failed assertions, and another looking for hot cycles in the b-program’s state space). It then starts the verification process, and receives in return a verification result from the verifier. If a violation is identified (e.g. the robot enters a trap), the verification result will contain a counter example leading to the discovered violating state. In this case, the host application will display the route of the robot that led to the violation, by reconstructing it from the events in the counter example execution trace. Chapter 4 expands on b-program verification.

The second requirement, “a robot should visit non-hot cells infinitely often”, is a liveness property [3]. As such, it can only be violated by infinite runs. BPjs detects these violations by finding hot cycles in b-program state graphs. To the best of our knowledge, BPjs is the first tool able to detect liveness violations in b-programs. The idea was sketched in [46], but never realized. Furthermore, the modular design of BPjs allows it to distinguish between cycles in which a single b-thread is indefinitely hot, and cycles where the b-program is indefinitely hot, but each of its b-threads is non-hot infinitely often. These cases represent different types of violations. To the best of our knowledge, this distinction — further discussed in Subsection 4.4.3 — is new.

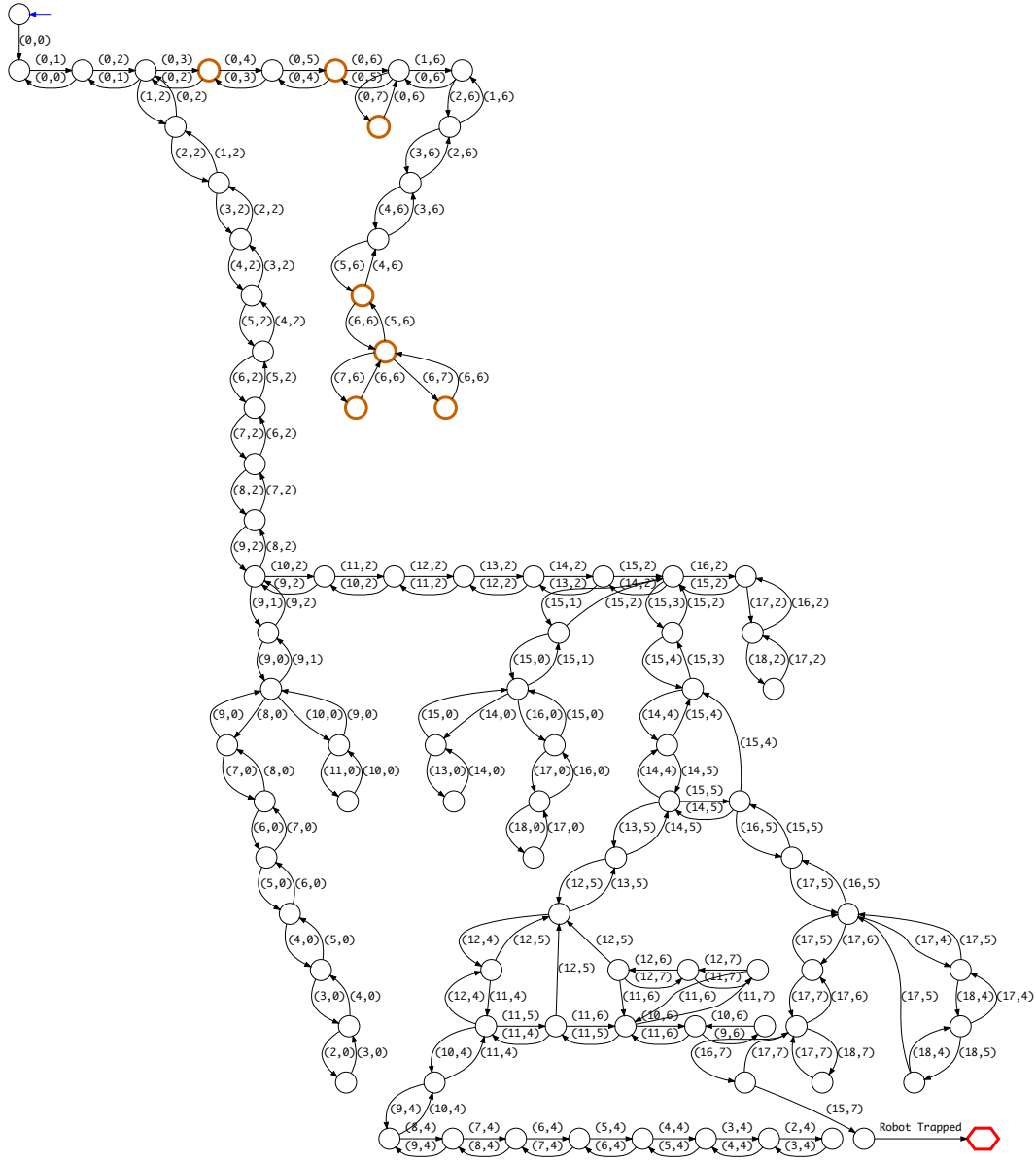


Figure 3.6: State space of a robot in a house simulation b-program, simulating the house shown in Figure 3.5. Nodes are synchronization points, edges are events. Hot synchronization points are marked with a thick orange line. Violating states, were a b-thread declared a failed assertion, are marked with a hexagonal shape. This graph was automatically generated by a tool that uses BPjs (see [6]) and later adjusted manually to fit this page.

3.4 Evaluation

This section presents evaluations of BPjs, covering performance and usability aspects. In particular, we evaluate:

1. How performant is BPjs as a runtime engine?
2. How performant is BPjs as a model analysis tool?
3. How intuitive is BPjs for programmers used to ordinary programming?

3.4.1 Performance Evaluation

We evaluated BPjs’ performance separately with regard to each of its usages: as a model analysis tool, and an execution engine. For the measurements presented, we used the robot movement simulation b-program presented in Subsection [3.3.2](#). We used square floor plans with no walls; a single start cell (located at the top left corner of the floor plan); a single trap cell (bottom right), and two hot cells (bottom left and top right). We choose these floor plans as a benchmark case, since the state-space of b-programs generated for this type of floor plan contains the highest state and edge counts for the given floor plan size. Thus, BPjs will work harder in this orientation than with a floor plan with similar dimensions, but with walls. Additionally, this type of floor plan lends itself to parameterization, an important trait when testing the impact of input size on performance. Last but not least, the state space of these b-programs — where most states have an out-degree of about 4 — is, in our experience, a typical case.

Each of the measurements were repeated 10 times and averaged. Unless otherwise noted, the measurements were taken on a 2.9 GHz Intel Core i9 MacBook Pro with 32GB RAM, of which 16GB was allocated to Java. The JVM used was OpenJDK 18.9 (Java 11).

To evaluate the performance of BPjs as a verification and analysis engine, we measured the time required to fully traverse the state space of a house simulation b-program. We used different floor plan sizes, and two types of

Plan Size	States	Edges	Time (msec)	
			Full	Hash
5×5	27	80	2,256 \pm 105	2,205 \pm 112
7×7	51	168	10,289 \pm 440	8,193 \pm 107
10×10	102	360	37,467 \pm 450	33,734 \pm 468
15×15	227	840	187,629 \pm 10,612	184,289 \pm 5,712
100×100	402	1520	658,478 \pm 9,201	640,358 \pm 7,269

Table 3.1: Average time required for BPjs to fully traverse the state space of a robot-in-a-house simulation b-program. Measurements were taken using OpenJDK 18.9 (Java 11), on a 2.9 GHz MacBook Pro. Each measurement was repeated 10 times.

visited state storage: storage that stores the states as objects, and storage that uses hash codes. The results are listed in Table 3.1, and analyzed in Figure 3.7. Verification times for the two storage types are similar — the hash-based storage performed slightly better. This is due to the fact that full state storage uses hash-based optimizations to quickly differentiate between non-matching states, and only uses full-state comparison to resolve possible hash conflicts. The slight difference in favor of the hash-based storage mostly reflects the fact that it did not require as much effort from the memory management sub-system.

There is a near-constant ratio (1.06 ± 0.07 full, 0.982 ± 0.46 hash) between the duration required to fully traverse the state space of a robot-in-a-house program, and the state space’s $|states| \times |edges|$ metric. This can help predict verification times when state and edge counts are known.

To compare BPjs’ verification performance against other verification alternatives, we implemented a similar robot-in-a-house program, using a modified version of BPJ [6], verifying it with NASA’s JavaPathFinder (JPF) [57]. BPJ needed to be modified in order to support a random-based event selection strategy¹. JPF is a versatile modular Java virtual machine, aimed at program analysis and verification. We used the same machine and memory allowance, but were obliged to use an older version of Java (1.8.0_201), as JPF cannot

¹This is another example demonstrating the advantages of BPjs’ modular approach.

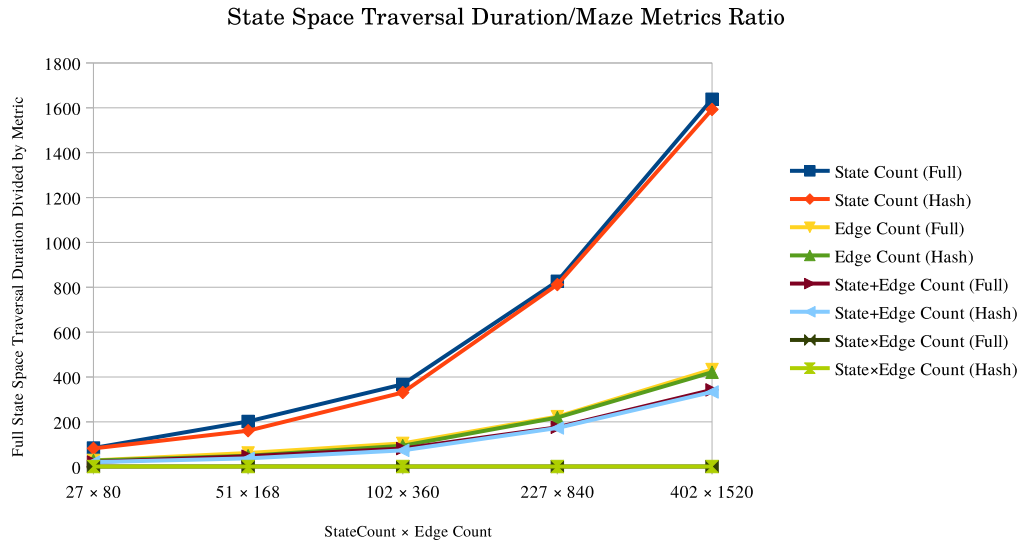


Figure 3.7: Ratio between the duration required for a full state space traversal and various state space graph metrics. Ratio between traversal duration and state count, edge count, or their sum, increases polynomially with the size of the state space. However, the ratio between traversal duration and $|states| \times |edges|$ remains constant as the state space grows. The two types of visited state storages (hashed and full) behave similarly.

Plan Size	B-threads	BPJ (msec)	BPjs (msec)
5×5	25	65.2 ± 6	238.4 ± 72
10×10	100	89.6 ± 12	403.5 ± 74
20×20	400	189.4 ± 17	1,094.7 ± 91
50×50	2,500	2031.2 ± 20	8,461.3 ± 102
100×100	10,000	<i>Out of memory</i>	12,605.7 ± 110

Table 3.2: Average time required for a robot to take 1000 steps in a house simulation (milliseconds). Measurements were taken using OpenJDK 18.9 (Java 11), on a 2.9 GHz MacBook Pro with 16G of RAM (out of 32) allocated to Java. Each measurement was repeated 10 times.

run on Java 11. To traverse the program state space, we used the core JPF system, which verifies a program by running all of its possible thread interleaving combinations, and enumerating the overall random decision points. Thus, JPF’s view of a b-program’s state space is much larger than BPjs’: BPjs only counts synchronization points as states, whereas JPF looks at thread interleaving options and calls to `java.util.Random`. Not surprisingly, the JPF verification process was much longer, taking 85 seconds to verify a 1×1 floor plan (containing 3 b-threads), visiting 438,568 states. While trying to verify a 2×2 floor plan (6 b-threads), JPF ran out of memory after 8:31 minutes.

We explored two other alternatives for verification, and found that both were not viable. BPMC, a model checker based on BPJ [46], does not support Java versions later than Java 5, which was released in 2004 and has not updated since 2009. Thus, we do not believe that it can be considered as a practical tool. Our attempts to verify the BPjs robot-in-a-house program using JPF failed for technical reasons, as JPF does not support some of the Java constructs used by BPjs.

To evaluate BPjs as a runtime engine, we measured the time required for a robot to perform 1000 moves in a house simulation program. For this test, we removed the trap cell, to ensure that runs were not terminated prematurely. We repeated the experiment with a range of floor plan sizes, and compared the results against a similar b-program executed using BPJ. The results (shown in Table 3.2) show that BPJ is about 5 times faster than BPjs. This is to

be expected, as Java is a compiled language whilst JavaScript is interpreted. Additionally, Rhino cannot use any optimizations while working with continuations, which forces BPjs to turn off all runtime optimizations during execution. However, because BPjs can run multiple b-threads using the same OS thread, it can execute b-programs with a larger concurrent b-thread count. When it attempted to run a b-program generated for a 100×100 floor plan, containing around 10,000 b-threads, BPJ exhausted its 16G memory allowance; BPjs, however, was able to run the program to completion.

3.4.2 Usage Evaluation

BPjs has been used in various projects and experiments, both by us and by other research students. The following research studies report on some of these usages. In [9], we used BPjs to execute live sequence charts, after translating these charts to b-programs. This allowed us to support variants of the LSC language through model transformation. In [35], we used BPjs in a model-driven solution for a leader-follower challenge presented by the MDETools 2018 workshop. Participants were required to develop control software for a rover, such that it could follow another rover at a safe distance. To this end, we wrapped a model written in BPjs in a traditional Java application. The Java layers were responsible for passing telemetry and sensor data to the BPjs model, and for actuating the simulated environment on the basis of selected events. The BPjs model was responsible for navigation decisions, such as acceleration and turning. In [5], we used a similar approach for implementing the on-board control software for a cube satellite. In the two latter works, we used BPjs’ analysis features to verify that the developed models complied with a set of formal requirements.

Another tool that uses BPjs’ analysis features is StateSpaceMapper (available at [6]). StateSpaceMapper is a utility program for diagramming the state-spaces of b-programs. Figures 2.2 and 3.6 were generated using this tool.

As a teaching platform, we have used BPjs in an undergraduate class for computer science (CS) and information systems engineering (ISE) students.

Students used BPjs to implement a number of projects, including a web-based PacMan game, a Blockly-based interface for behavioral programming, and the implementation of strategies in computer games. A student survey at the end of the term showed that the students found the material interesting (4.5/5) and relevant (4.5/5).

In the course feedback, a 3rd year CS student wrote: “the whole idea of this system is very interesting for me, because it is actually a different approach to problems than the approach we, as students in CS, are used to. The way the BP engine works ... might be strange in the beginning, but when I got into it - it looked really logical and obvious”. Another student commented: “Using the decision engine based on request, wait-for, and block, was initially hard to understand, but after a few examples I was able to understand it and enjoyed using it. I found this way of thinking interesting and challenging”. One student concluded his feedback by saying: “Finally, I can say that this system is revolutionary in the way it sees and solves problems, but at the same time is really friendly to the user.”

BPjs was used as the model core of a graphical, web-based rule engine developed by a team of students and researchers during a 28-hour hackathon. The system went on to win first prize².

In order to determine whether practitioners similarly find BPjs interesting and innovative, we submitted a talk about it to Devovx Belgium 2018³. Devovx is a community-led developer conference, focused on the Java community. For the 2018 conference, Devovx Belgium received 1600 talk submissions, of which 167 were selected by the content committee. Our talk, titled “Rethinking Software Systems: A friendly introduction to Behavioral Programming” was one of the talks accepted. This supports our view that BP is of interest to practitioners, and that BPjs may be a platform capable of sparking collaborations between academia and practicing software engineers. We received additional evidence supporting this presumption in the week before the conference, when Java developer at JUVU listed our talk as one of his 3 “must

²First place in HackBGU, <https://bit.ly/2riUjB0>

³<https://devovx.be>

see” talks (out of a total of 217)⁴.

The talk was initially scheduled to take place in one of the smaller rooms. However, following feedback from the Devovx conference website — which allows conference goers to indicate anticipated and/or popular talk abstracts — our talk was moved to a larger room with a capacity of 650 seats. We could not determine the exact attendance, but the room was quite full (see Figure 3.8).

The talk itself was rated 73 out of 217 by Devovx attendees⁵. As of March 23 2019, its YouTube video had been viewed 787 times, making it the 95th most-viewed of the 212 videos available on Devovx 2018’s YouTube channel. Currently, however, we are not aware of any use of BPjs outside academia⁶.

While our data from Devovx 2018 is anecdotal, it does indicate that practitioners are interested in BP and in BPjs, and are willing to invest valuable conference time to learn about them.

3.4.3 Evaluation Summary

In summary, we can say that BPjs is reasonably performant, both as a runtime engine and as a model analysis tool. While it is not as fast as BPJ, which enjoys Java’s compiled, statically typed nature, its execution time is still reasonable, and its lower memory requirements allow it to run b-programs with more b-threads than BPJ. As for intuitiveness for programmers, we have seen that computer science and software engineering students can use BPjs for creating non-trivial systems, with little or no help.

Threats to Validity. Our performance tests included a large amount of b-threads sharing the same code, and had about 4 selectable events at each synchronization point. Additionally, the `waitFor` and `block` event sets did not contain complex logic. BPjs may perform differently for B-program whose

⁴<https://www.juvo.be/blog/devovx-2018-juvos-must-sees>

⁵<https://www.linkedin.com/pulse/top-100-rated-talks-from-devovx-belgium-2018-stephan-janssen/>

⁶The video did encourage a Kotlin developer to download and try the Kotlin version of ScenarioTools [36], which supports BP through a textual version of LSC.



Figure 3.8: BPjs talk at DevOxx Belgium 2018. DevOxx is a developer-led conference focusing on JVM languages. The fact that a talk about BPjs was accepted and well attended supports our view that BPjs, and BP in general, are ready for non-academic use. Photo: James Birnie’s blog (<http://www.jamesbirnie.com/2018/11/devOxx-belgium-and-two-talks.html>).

b-threads vary a lot or that require complex event filtering logic. Regarding intuitiveness evaluation, the main threat to validity is that we have used university students. Students may be more open to new ideas, as they have less experience than professionals, and have recent experience in learning new paradigms (e.g. because of taking a Functional Programming class). Thus, our intuitiveness evaluations might be positively biased.

3.4.4 Quality Assurance

BPjs is a complex software system, and some parts of it are tricky to implement correctly. To ensure its quality, we use unit tests, in combination with continuous integration and code coverage analysis. At the time of this writing, BPjs’ code repository contains 198 unit tests, which cover 84.78% of BPjs’ code⁷. Developers are encouraged to run these tests on their machines during development prior to committing the code to the central repository. Quality assurance, however, does not rely on developer’s discipline, but rather on con-

⁷Detailed report is available at <https://coveralls.io/builds/22502875>

tinuous integration. Whenever new code is pushed to the central repository, a continuous integration server builds the library from its updated sources, and runs all unit tests available in code. If any of these tests fail, the developer is automatically informed.

A similar process is used to ensure that BPjs' on-line documentation is up-to-date. BPjs' high-level documentation is available in a web site, which includes tutorials, examples, and other similar resources⁸. The source code for this website is stored with BPjs' source code. When new code is pushed to the central code repository, the documentation server is informed, and re-builds the documentation website.

BPjs' low-level documentation, which covers its packages, classes and methods, is created using Javadoc⁹. When a new version of BPjs is published to Maven Central, a new version of its low-level documentation is automatically made available on-line.

This level of automation was not possible without offerings provided for free, by commercial companies that support open-source software. We are grateful for the services of GitHub (central code repository), TravisCI (continuous integration), Coveralls.io (code coverage analysis), ReadTheDocs.io (high-level documentation publishing), Javadoc.io (low-level documentation publishing) and Sonatype (Maven Central binary repository).

3.5 Conclusions

This chapter presented BPjs, a platform for developing software systems based on the Behavioral Programming paradigm. Unlike previous BP systems, which were designed to serve specific goals, BPjs aims to be an open platform on which advanced BP-based tools can be created. BPjs views b-programs as data structures, and can both analyze and execute them. We used it to implement the first BP verifier capable of identifying liveness violations. BPjs supports features that were not available in previous BP systems; it also allows for

⁸See <https://bpjs.readthedocs.io/en/latest/>

⁹See <https://docs.oracle.com/en/java/javase/13/javadoc/javadoc.html>

the embedding of b-programs in traditional software systems by supporting a well-defined communication protocol, another feature which was hitherto unavailable. BPjs is an open-source project, available at GitHub^[10]. On-line documentation, reference, and tutorials, are available at its documentation site^[11].

BPjs can be improved in a number of ways, which we leave to future work (by us and others). These include better performance (especially during verification), lower memory requirements, new event selection strategies, improved debugging and logging tools, and a heuristics-driven verifier. Porting parts of the substantial body of work already developed using other BP tools will also be a worthwhile undertaking.

Software packages such as ROS [93] (robotics), the R language [94] (statistics), and Zelig [21, 61] (statistic modeling) provide a platform for tool creation, and a common infrastructures for sharing ideas and reusing code. They accelerate research and facilitate its dissemination not only in academia, but also across the campus fences and with industry, schools, and hobbyists. BPjs follows these packages by providing a common environment for Behavioral Programming, both extensible and easy to use. We hope that in due course BPjs will become a boring part of BP research — a reliable software package that helps make creating the exciting parts easier.

¹⁰<https://github.com/bThink-BGU/BPjs>

¹¹<https://bpjs.readthedocs.io>

Chapter 4

BP, Verification, and Pancakes

One strong point of BP — and SBP in general — is the fact that programs created with these paradigms are amenable to verification. Thus, BP can potentially be used to create highly reliable systems, whose correctness would not merely be anecdotally demonstrable through testing, but could also formally proven by verification. However, as is to be expected, there is a gap between identifying this potential and actually realizing it.

In this chapter, we present a number of methods for verifying the safety and liveness properties of behavioral programs. Verification is performed using direct code execution, with no model transformations. Our extensible analysis method is based on traversing the state-space of an analyzed b-program, and finding violating states and “hot” cycles. We identify and discuss two types of liveness violations; a case where it makes sense to talk about liveness violations in the context of finite runs; and the fundamental differences between liveness and safety properties in the context of Behavioral Programming. These methods are supported by BPjs, our BP execution and analysis platform, presented at Chapter [3](#).

This work is informed by previous attempts to verify programs written in BP and/or SBP (see Section [4.5](#)). The direct formal verification of b-programs was hitherto limited to safety properties. The verification of liveness properties, outlined briefly in [\[46\]](#), has never been realized. The work presented here can verify both types of properties, and can do so using direct verification. Ad-

ditionally, the work presented here allows dynamic b-thread additions, which were not supported in [46].

An alternative approach to direct verification is model transformation. Under this approach, a b-program is analysed using a three-stage process. First, it is translated to the input language of an existing model checker (e.g. SPIN [60]). Second, the model checker is used to perform an analysis. Third, the analysis result is translated back to b-program terms. This approach was taken by multiple previous works, both for BP [42] and for LSC [45, 55, 77, 78]. The main advantage of model transformation is the ability to build upon an existing tool, and, given a correct bidirectional translation, use symbolic model checking [18, 91]. The main disadvantages are the need to develop and implement a bidirectional translation algorithm between the source b-program and the model language, which often introduces “fine print” regarding what features of the language can be analysed, and the requirement to perform the analysis using model checker terms. Direct verification, on the other hand, does not require translation, and allows analysis engines to work using BP terms, which makes it easier to work with and extend them. Similar considerations led [46] to use direct verification as well.

Our use of a modular design allows new types of verifications and analyses to be implemented without the need to negotiate the complex technicalities involved in creating a verification engine. This design approach facilitates research, and indeed made it possible for us to identify the two violation types described above.

The rest of this chapter is organized as follows. Section 4.1 presents a running example of a model-driven system based around a behavioral model. Section 4.2 presents a method for declaring the liveness properties of behavioral programs. Section 4.4 discusses the verification of behavioral programs, using examples from Sections 4.1 and 4.2. Section 3.1 discusses related work, and Section 4.6 presents our conclusions.

4.1 The Pancake Maker

Let us consider a mixer for preparing pancake batter, controlled by a b-program (Figure 4.1). In this example, we will employ a few variants of this system to demonstrate how behavioral programs can be verified, and will discuss their liveness and safety properties. In order to allow for a comparative discussion, this section presents the mixer variants; Section 4.4 discusses how these can be verified. The code used in this section is available in the chapter’s code appendix 7.

Pancake batter is made out of two mixtures: dry (flour, baking soda, salt) and wet (eggs, buttermilk, vanilla extract). Our batter mixer has a mixing bowl, and two containers equipped with computer-controlled valves, one for each type of mixture. The mixer is controlled by a program built using a model-driven approach; control decisions are made by a b-program, serving as a model. A traditional program (“host”) wraps the model, feeds it with external inputs, and responds to its decisions by actuating the mixer’s components.

Specifically for a BP model, the host program translates external inputs to events, enqueueing them in the b-program’s external event queue. The model publishes its decisions by selecting events. The host b-program listens to the model’s event selection, and acts accordingly. In this example, when the b-program selects an `ADD_DRY` event, the wrapping program pours a single dose of dry mixture into the mixing bowl; when the b-program selects an `ADD_WET` event, the wrapping program pours a single dose of wet mixture into the bowl.

The proposed batter mixer design has two notable properties. First, all of the decisions are made by the b-program, serving as a model. The host program’s logic, on the other hand, is trivial. Thus, this paper can focus on the verification and correctness of the b-program, without sacrificing system correctness. Second, the b-program is not aware of the specific events that the host program reacts to. From an engineering perspective, this separation allows for different machines to use the same model, and vice-versa. However, it also means that the developers of the model and of the host program must agree on an interface, in the form of a set of events and their semantics.

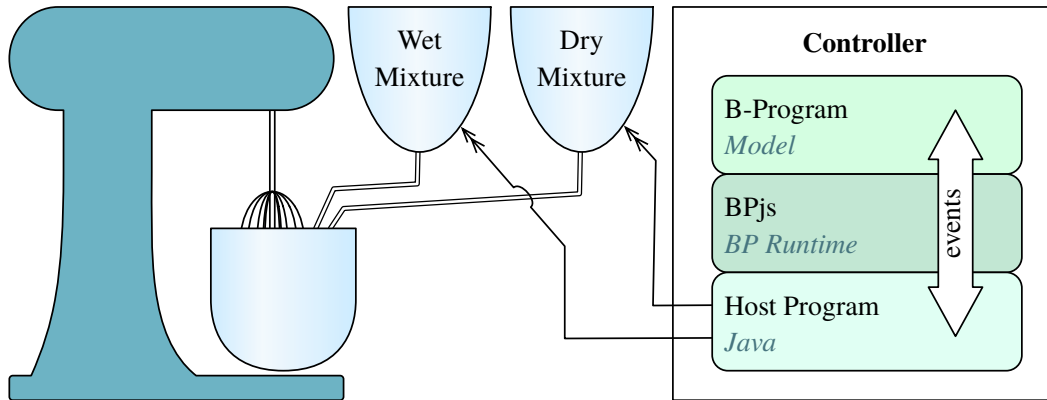


Figure 4.1: A computer-controlled pancake batter mixer, where mixture flow to the mixing bowl is controlled by a b-program. The host program, a traditional Java program, converts its inputs to events and feeds them to the model for processing. Additionally, the host program translates event selections made by the model to real-world actions.

Listing 4.1: Naive b-program for preparing batter for 5 pancakes. Consists of two b-threads, each requesting the addition of 5 doses of ingredient mix.

```

1 bp.registerBThread("AddDries", function(){
2   for ( var i=0; i<5; i++ ) {
3     bp.sync({request: ADD_DRY});
4   });
5 bp.registerBThread("AddWets", function(){
6   for ( var i=0; i<5; i++ ) {
7     bp.sync({request: ADD_WET});
8   });

```

4.1.1 Plain Pancake Program

Let's assume that our mixer is required to make batter for five pancakes. This requires five doses of each mixture. The b-program presented in Listing 4.1 prepares the correct amount of batter by having two b-threads — one for each type of mixture — request five addition events.

The program in Listing 4.1 does not by itself specify any order for adding the mixtures. Consequently, the batter may become too thick or too thin during the preparation process, such as when several dry doses are added before any wet doses. Extreme batter thickness values can damage the mixer's engine, as they may cause it to run too fast or to exert too much effort. In

Listing 4.2: A b-thread that restricts pancake batter thickness by forcing the addition of a dry mixture dose after each addition of a wet mixture dose. The addition or removal of this b-thread from a b-program does not require updating the code of other b-threads.

```
1 bp.registerBThread("StrictArbiter", function(){
2   while (true) {
3     bp.sync({waitFor: ADD_WET, block: ADD_DRY});
4     bp.sync({waitFor: ADD_DRY, block: ADD_WET});
5   });
```

order to control the thickness of the batter, we must impose restrictions on the order in which the mixture's doses are added to the mixing bowl.

One way of controlling the addition order is by adding the **StrictArbiter** b-thread shown in Listing 4.2. This b-thread repeatedly waits for an addition of a wet dose, while blocking the addition of a dry dose. Once a wet dose has been added, it will block the further addition of wet doses until a dry dose has been added. It is worth noting that this b-thread does not request any events — it simply waits for, or blocks them. More importantly, it can be added to and removed from a b-program without affecting the program's other b-threads.

The **StrictArbiter** b-thread in Listing 4.2 resolves the problem of extreme batter thickness, by restricting the controller b-program to a single sequence of events (namely, `ADD_WET`, `ADD_DRY`, `ADD_WET`, ...). This type of solution is overly restrictive — recall that we wanted to avoid extreme batter thickness values, not to keep the batter at strictly 1:1 ratio between the mixtures.

Now, let's consider a situation where a batter mixer runs out of dry mixture. While the dry mixture tank is being re-filled, the mixer may still be able to add more wet mixture, as long as the batter does not become too thin. To allow this, we replace the **StrictArbiter** b-thread with two b-threads: one for monitoring batter thickness, and another for blocking addition events when batter thickness reaches values outside of defined range. A detailed description of these b-threads follows.

ThicknessMeter, a b-thread that monitors batter thickness, is shown in Listing 4.3. It maintains a thickness index by listening to addition events.

Listing 4.3: A b-thread that keeps track of batter thickness. After each addition event, this b-thread updates its internal thickness index, sharing it with the rest of the b-program by requesting a **Thickness** event that holds the updated thickness index in its data field.

```

1  var ADDITION_EVENTS = [ADD_WET, ADD_DRY];
2  bp.registerBThread("ThicknessMeter", function(){
3      var thickness=0;
4      while ( true ) {
5          var evt = bp.sync({waitFor:ADDITION_EVENTS});
6          if ( evt.equals(ADD_DRY) ) {
7              thickness++;
8          } else {
9              thickness--;
10         }
11         bp.sync({
12             request:bp.Event("Thickness", thickness),
13             block:ADDITION_EVENTS
14         });
15     });

```

After each addition event, it announces the updated thickness index by requesting a **Thickness** event, while blocking all addition events. This block is required, because if another addition event is selected before the thickness event, the thickness event's data will become stale.

The second b-thread we add is **RangeArbiter** (Listing 4.4), which keeps the batter thickness within acceptable range by listening to batter thickness events and blocking wet or dry mixture additions when the thickness index crosses a threshold. For example, when batter thickness index is too high, **RangeArbiter** blocks **ADD_DRY**.

With **RangeArbiter** in place instead of **StrictArbiter**, our batter mixer can advance even when other parts of the system block the addition of, say, wet mixture. And it is able to do so while keeping the the batter thickness in its healthy range.

4.1.2 Preparing Blueberry Pancakes

We now add a new feature to our batter mixer: preparing blueberry pancakes. This is done by attaching a controlled tank containing blueberries to the mixer, and defining the **BLUEBERRIES** event, which prompts the host program to add a

Listing 4.4: A b-thread that keeps the thickness of the pancake batter within range, but allows the event selection mechanism a degree of freedom.

```

1 var THICKNESS_BOUND = 2;
2 bp.registerBThread("RangeArbiter", function(){
3   while ( true ) {
4     var thicknessEvt = bp.sync({waitFor:THICKNESS_EVENTS});
5     var thickness = thicknessEvt.data;
6     var block;
7     if ( Math.abs(thickness) >= THICKNESS_BOUND ) {
8       block = (thickness>0) ? ADD_DRY : ADD_WET;
9     } else {
10      block = bp.none;
11    }
12    var evt = bp.sync({waitFor:ADDITION_EVENTS, block:block});
13  });

```

dose of blueberries to the batter. B-thread **Blueberries**, shown in Listing 4.5, is responsible for requesting this event.

In the context of this chapter, we hold that the quality of a blueberry pancake is proportional to the number of whole blueberries that it contains (as opposed to the number of blueberries burst during the preparation process). To keep blueberries from being burst, two conditions must be met at the time that the blueberries are added. First, there must be enough batter in the mixing bowl. Second, the batter must be relatively thin. B-threads **EnoughBatter** and **BatterThinEnough**, also in Listing 4.5, block the blueberry addition event until these conditions are met.

Depending on the order in which the dry and wet mixtures are added, there may be cases where the **BLUEBERRIES** event is blocked by **BatterThinEnough** throughout the execution of the program. A run where no blueberries are added would breach system requirements. Section 4.2 proposes a way of detecting this breach.

4.1.3 Blueberry Pancake Server

So far, our mixer has prepared batter for a single batch of pancakes, and thus all of its runs were finite. We now consider a mixer variant, which repeatedly prepares batter for a single batch, releases the batter (presumably to an automated pan — left for future work) and then prepares batter for the next batch.

Listing 4.5: A set of b-threads responsible for adding blueberries. The `Blueberries` b-thread requests a `BLUEBERRIES` event which, when selected, causes the host program to add blueberries to the pancake batter. The other b-threads will block the addition of blueberries until there is enough batter, or if the batter is too thick.

```

1 bp.registerBThread("Blueberries", function(){
2   bp.sync({request:BLUEBERRIES});
3 });
4
5 bp.registerBThread("EnoughBatter", function(){
6   bp.sync({waitFor:ADDITION_EVENTS, block:BLUEBERRIES});
7   bp.sync({waitFor:ADDITION_EVENTS, block:BLUEBERRIES});
8   bp.sync({waitFor:ADDITION_EVENTS, block:BLUEBERRIES});
9 });
10
11 bp.registerBThread("BatterThinEnough", function(){
12   while ( true ) {
13     var blk;
14     var thicknessEvt = bp.sync({ waitFor:THICKNESS_EVENTS,
15                               block:blk,
16                               interrupt:BLUEBERRIES});
17     blk=(thicknessEvt.data>=0) ? BLUEBERRIES:bp.none;
18   });

```

This variant's runs are infinite, which makes it an interesting case study with regard to liveness properties.

The server code, part of which appears in Listing 4.6, builds on the code in Listing 4.1. The mixture addition loops are followed by a wait for a `RELEASE` event, and are wrapped in an infinite loop. A newly added b-thread releases the batter when the bowl contains enough of it. Parts of the code similar to those presented earlier have been omitted for brevity — the full code is available in the chapter's code appendix 7.

4.2 Hot Synchronization Statements

In some circumstances, neither of the blueberry pancake mixers presented in Section 4.1 will be allowed to add blueberries to the batter. This happens in runs where the amount of dry mixture is always equal to or more than the amount of wet mixture. In such cases, the `BLUEBERRIES` event is blocked by `BatterThinEnough` throughout the execution of the program. This type of re-

Listing 4.6: Parts of the pancake server code. Mixture-adding b-threads run in an infinite loop, where they first add the required amount of doses to the mixer bowl, then wait for the batter to be released. A **Releaser** b-thread is responsible for releasing the batter when it reaches a set threshold.

```

1 var RELEASE = bp.Event("RELEASE_BATTER");
2
3 bp.registerBThread("Dry", function(){
4     while ( true ) {
5         for ( var i=0; i<DOSE_COUNT; i++ ){
6             bp.sync({request:ADD_DRY});
7         }
8         bp.sync({waitFor:RELEASE});
9     }});
10
11 bp.registerBThread("Releaser", function(){
12     var doseCount = 0;
13     while ( true ) {
14         bp.sync({waitFor:ADDITION_EVENTS});
15         doseCount++;
16         if ( doseCount === (DOSE_COUNT*2) ) {
17             bp.sync({request:RELEASE,
18                     block:ADDITION_EVENTS});
19             doseCount=0;
20         }
21     }});

```

quirement violation poses an interesting challenge for behavioral programming, since the ability to block events is central to the paradigm. In the blueberries case, however, it is OK to block the BLUEBERRIES event *for a while*, as long as it is eventually selected. This requirement cannot be expressed using existing BP synchronization statements; so we will borrow the *must/may* modality idiom from another scenario-based programming language: Live Sequence Charts (LSCs) [48].

A b-thread can mark its synchronization statement as *hot*, stating that it eventually must leave it. In other words, runs where a b-thread is forever stuck at a synchronization point to which it submitted a hot synchronization statement violate program requirements. We say that a b-thread is *hot* at a given synchronization point, if it has submitted a hot synchronization statement to that point.

The term *hot* is borrowed directly from LSC, where it has a similar semantic meaning, albeit with reference to different constructs (see Section 3.1).

Listing 4.7: A b-thread waiting for the BLUEBERRIES event to be selected, using a hot synchronization point. By submitting a hot synchronization statement, a b-thread states that it cannot be blocked at its current location indefinitely. In the present case, this means that the BLUEBERRIES event must eventually be selected. Adding this b-thread to the blueberry batter mixer from Section 4.1 will allow verification to detect runs where blueberries are never added.

```

1 bp.registerBThread("MustAddBlueberries", function(){
2   bp.hot(true).sync({waitFor: BLUEBERRIES});
3 });

```

In BPjs, marking a synchronization point as hot is done using a builder-like pattern, by calling `bp.hot(true).sync(...)`. For example, the b-thread in Listing 4.7 hot-waits for a BLUEBERRIES event to be selected. Such synchronization statements do not force event selection. They do, however, allow formal verification processes to detect runs where events that should eventually be selected never are. This type of verification is explained in Section 4.4.

We now turn to the last modification of our batter mixer. In an attempt to convert our blueberry pancake into a healthy meal, we have decided to add kale to the batter. The amount of kale should, eventually, be equal to the amount of blueberries. To this end, we define an `ADD_EXTRAS` event, with two data fields: one for the amount of blueberries added, the other for the amount of kale. Two new b-threads (one for blueberries, the other for kale), monitor the amount of added extras, and make requests to ensure that the amounts are balanced. Listing 4.8 contains selected part of the code (the full program is available in [7]). Both b-threads request the addition of their respective ingredient using a hot synchronization statement, since the ingredient portion *must* eventually be added.

4.3 Formal Definition of Hot Synchronization

In this section we extend the formal definition for BP, presented in Section 2.2, to support the *hot synchronization* concept. To this end, we add a function $H: S \rightarrow \{0, 1\}$ to Definition 2.2.1 (b-thread). This function marks statements

Listing 4.8: Code for balancing the addition of blueberries and kale to our proposed blueberry-kale pancake batter maker. The `BlueberryAdder` and `KaleAdder` b-threads monitor the blueberry/kale ratio in the batter, and request events to adjust it if it becomes unbalanced. The b-threads use hot synchronization requests, as the ingredients must eventually be added. Some repeated code has been omitted for brevity.

```

1 function addExtrasEvent( blueb, kales ) {
2   return bp.Event("ADD_EXTRAS", {blueberries:blueb,
3                                   kales:kales});
4 }
5 var ADD_EXTRAS = bp.EventSet("sADD_EXTRAS", function(e){
6   return e.name.equals("ADD_EXTRAS");
7 });
8
9 bp.registerBThread( "KaleAdder", function(){
10   var fruitIndex=0;
11   while (true) {
12     var evt = null;
13     if ( fruitIndex > 0 ) {
14       evt = bp.hot(true).sync({request:addExtrasEvent(1,0),
15                               waitFor:ADD_EXTRAS});
16     } else {
17       evt = bp.sync({waitFor:ADD_EXTRAS});
18     }
19     fruitIndex = fruitIndex + evt.data.blueberries-evt.data.kales;
20   });
21 bp.registerBThread( "BlueberryAdder", function(){...});

```

as *hot* ($H(s) = 1$) or non-hot ($H(s) = 0$). The extended definition of a b-thread is as follows:

Definition 4.3.1 (b-thread). A b-thread is a tuple $\langle S, E, \rightarrow, init, R, W, B, H \rangle$ where $\langle S, E, \rightarrow, init \rangle$ forms a labeled transition system, $R: S \rightarrow 2^E$ associates a state with the set of events requested by the b-thread when in it, $W: S \rightarrow 2^E$ associates a state with the set of events waited for by the b-thread when in it, $B: S \rightarrow 2^E$ associates a state with the set of events blocked by the b-thread when in it, and $H: S \rightarrow \{0, 1\}$ is a labeling function that indicates if state is hot (1) or cold (0).

The b-program definition (see Definition 2.2.2) does not need to be updated to support hot synchronization. However, we can now say that a b-program $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, W_i, B_i, H_i \rangle\}_{i=1}^n$ is *hot* at synchronization point s , if:

$$\exists i \in [0..n] \text{ such that } H_i(s) = 1$$

4.4 Adding Verification to the Mix

Using the formal definitions for b-thread (Definition 4.3.1) and b-program (Definitions 2.2.2), b-programs can be *verified* against a set of formal requirements. Similar to traditional verification, the model checker traverses a transition system whose states represent the states the analyzed program can be in. In BP program analysis, however, the transition system states represent synchronization points, rather than memory values and program counter locations. Thus, the number of states involved in verifying a b-program is smaller than that involved in a traditional verification process.

The transitions of a b-program's transition system are labeled by events. Outgoing transitions of each state represent events that were requested and not blocked at that state. Incoming transitions represent events that bring the b-program to that synchronization point, from the transitions's respective source state. It is important to note that, except when using random number generators or accessing external resources, a b-program's progression between

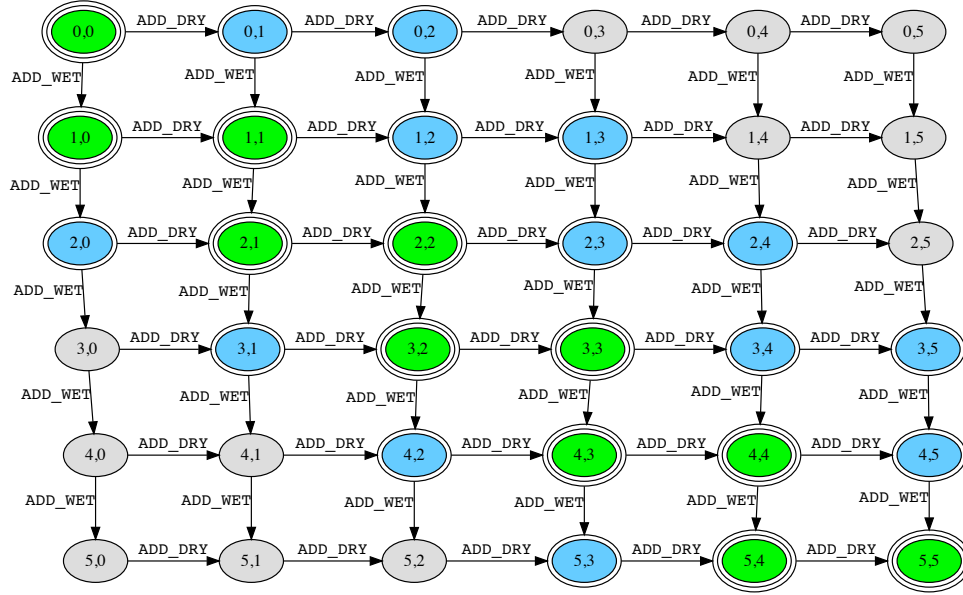


Figure 4.2: The transition system of the plain pancake batter b-program. Ovals represent synchronization points (states), while transitions represent events. The text in the ovals indicates the number of doses of each mixture added up to that point. Possible program runs traverse the graph, starting at state (0,0) and eventually reaching state (5,5). All states are reachable if running only Listing 4.1. When adding the strict arbiter b-thread in Listing 4.2, only the green/3-line states are reachable. When replacing the strict arbiter with the **RangeArbiter** of Listing 4.4, the blue/2-line states also become reachable.

synchronization points/states is completely determined by an ordered list of events. Thus, a series of events leading to a state containing a requirement violation constitutes a counter-example, equivalent in value to counter examples generated by traditional model checkers.

Figure 4.2 presents the transition system of the plain pancake batter maker described in Listing 4.1. It also demonstrates how the arbiter b-threads in Listings 4.2 and 4.4 limit the possible runs of the plain batter mixer.

It sometimes makes sense to add b-threads to a b-program to support its verification process. We refer to the verified b-program, which contains

the core b-program and verification-only b-threads, as the *greater b-program* (borrowing “greater” from urban planning, as in “Needham is a town in greater Boston”). Common examples of such b-threads include:

Requirements

B-threads that directly model system requirements, by waiting for events and informing the model checker of violations. This is done either by marking states as problematic (see Subsection 4.4.1), or by using hot synchronization statements for detecting cases where an event that must eventually be selected, is never selected. These b-threads do not interfere with the execution of the core b-program; they only wait for events, and do not request or block them.

Environment Simulation

B-threads that simulate the b-program’s environment, by requesting events normally requested by the host program, based on its interaction with the environment (e.g. user input or sensor data). These b-threads turn the greater b-program into a b-program that does not require external events in order to progress; as a result, a deadlock in a greater b-program is likely a bug (see Subsection 4.4.2).

Assumptions and Focus

B-threads that direct or keep the verifier in a specific part of the program’s state-space. For example, when focusing on cases where blueberries are to be added, it makes sense to add a b-thread that prevents runs where blueberries are never added, e.g. by blocking `ADD_DRY` at specific points.

Under BPjs, verification of a b-program occurs as follows. A b-program verifier traverses the b-program’s state graph using depth-first search, with a possible bounded depth. During this traversal, the verifier searches for the following violations¹:

¹Users can change the search parameters, and add new inspections as well. The items listed here are the default set.

1. Nodes marked as having violations
2. Deadlocks
3. Hot Cycles
4. Hot Terminations

If any of the items listed above is found, the verifier informs the host application of the violation. It additionally provides a counter example, in the form of a trace containing program states and events. The host can then decide whether verification should continue or not. These traces are useful counter examples, as their events use the same terms as those used by the programmers writing the b-program.

BPjs uses direct verification: transition between two synchronization points is done using code execution. To this end, BPjs captures a continuation of each b-thread when it calls `bp.sync`. The continuation is serialized for storage, and de-serialized for the traversal from its node to a next one. For this technically complex feat, we rely in part on the features of Mozilla Rhino [85].

4.4.1 Safety Properties

Safety properties describe a scenario where “something went unrecoverably wrong” [3]. In this case, this means that a b-program has reached a state that violates one or more of its requirements. For the plain pancake batter mixer (Listing 4.1) composed with the `RangeArbiter` of Listing 4.4, this would mean that batter thickness is no longer within the predefined range.

We propose that states be marked as “bad” using *assertions*. B-threads call `bp.ASSERT`, passing to it an expression that evaluates to a boolean value, and an optional human-readable description of what was violated if the expression evaluates to `false`. Listing 4.9 shows `RangeVerification`, a b-thread that tracks batter thickness and confirms that it is within valid range.

During verification, false assertions are used to mark synchronization points as having a violation. Listing 4.10 shows a verification output for a b-program

Listing 4.9: A b-thread modeling the formal requirement “batter thickness has to be between -3 and 3”. This b-thread maintains a local variable which monitors batter thickness. Each time an addition event is selected, the variable is updated and tested against the assertion. This is a typical *requirement b-thread*, in that it does not interfere with the core b-program’s events, and is aligned with the requirement it validates.

```

1 bp.registerBThread("RangeVerification", function(){
2   var thickness=0;
3   while (true) {
4     var evt = bp.sync({waitFor:ANY_ADDITION});
5     if ( evt.name == ADD_WET.name ) thickness--;
6     if ( evt.name == ADD_DRY.name ) thickness++;
7     bp.ASSERT(Math.abs(thickness)<3,
8               "Batter thickness out of range (" +thickness+"));
9   });

```

Listing 4.10: Verification result of the pancake batter b-program with `RangeArbiter`. The output shows the violated assertion, and presents the sequence of events that led to it. Some lines have been omitted for brevity.

```

1 # Verification completed.
2 # Violation type: FailedAssertion
3 # Failed assertion: Batter thickness out of range (thickness: 3)
4 #   By b-thread: RangeVerification
5 # Counter example:
6 # [BEvent name:ADD_DRY]
7 # [BEvent name:ADD_WET]
8 # [BEvent name:ADD_WET]
9 # [BEvent name:ADD_DRY]
10 # [BEvent name:ADD_DRY]
11 # [BEvent name:ADD_DRY]
12 # [BEvent name:ADD_DRY]

```

composed of the plain batter mixer, a modified version of the `RangeArbiter`, and `RangeVerification` (Listings 4.1, 4.4, and 4.9, respectively). The modification to `RangeArbiter` was the common mistake of replacing `>=` with `>`. This outputs a counter-example: a possible run of the composed b-program, where batter thickness goes beyond the predefined range. The trace to the violating state is listed as well. On the state graph in Figure 4.2, this is equivalent to traversing the path $(0,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (2,2) \rightarrow (2,3) \rightarrow (2,4) \rightarrow (2,5)$.

Assertions can also be useful at runtime, where they will halt program

execution if their condition evaluates to **false**. This is a form of *runtime verification* [1], prompting the emergency shutdown of a program if selected conditions have been violated. Should this occur, the BPjs runtime informs the host program of the failed assertion. This gives the host the opportunity to enter a “safe mode”, to re-start the b-program with different parameters, or even to automatically patch it [43, 62].

Discussion - Safety properties Under BP, when a safety requirement can be phrased as series of n events, we can model it using a *forbidden scenario* b-thread. This b-thread waits for the first $n - 1$ events, and then blocks the last event. Whenever a forbidden scenario b-thread is added to a b-program, that b-program is safe-by-construction with regard to the safety requirement that the forbidden scenario models.

Formally, suppose a run $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \dots \xrightarrow{e_{n-1}} s_n$ violates a safety requirement when it gets to state s_n . By adding a b-thread $b_{fix} = \langle S_{fix}, E_{fix}, \rightarrow_{fix}, init_{fix}, R_{fix}, W_{fix}, B_{fix}, H_{fix} \rangle$, where $\forall i \in [0 \dots n - 2]. \{e_i\} = W_{fix}(s_i)$, $\{e_{n-1}\} = B_{fix}(s_{n-1})$, and $\forall j \in [0 \dots n]. R_{fix}(s_j) = \emptyset$, we can prevent the b-program from taking this execution path. Adding b_{fix} will not affect the b-program run up to s_{n-1} , as it only waits for events until that state; it does not request or block them.

Forbidden scenarios can only be used for internal system processes. For example, there is no point blocking an event announcing that the tracking rover has fallen too far behind its leader (assuming that the leader is an external system). This is why failed assertions are useful — they can declare cases that the system does not have full control over as invalid. If the system did have full control over the occurrence of a requirement-violating event, it would have simply blocked it, e.g. by adding an appropriate b_{fix} as described above.

One type of safety property cannot be expressed using the assertion mechanism: Deadlocks.

Listing 4.11: Verification result of a b-program composed of plain pancake maker, blueberry adder, and thickness monitor (Listings 4.1, 4.3, and 4.5, respectively). The verification found a deadlock, where `Blueberries` requests the `BLUEBERRIES` event, which is blocked by `BatterThinEnough`. The b-program is considered deadlocked as it cannot advance any further, but requested events remain unfulfilled. Some lines have been omitted for brevity.

```

1 # Verification completed.
2 # Found Violation:
3 # Deadlock: [BEvent name:ADD_BLUEBERRIES]
4             requested by:{Blueberries}
5             blocked by:{BatterThinEnough}
6 # Counter example trace:
7 # [BEvent name:ADD_DRY]
8 # [BEvent name:Thickness data:1.0]
9 # [BEvent name:ADD_DRY]
10 # [BEvent name:Thickness data:2.0]
11 # [BEvent name:ADD_DRY]
12 # [BEvent name:Thickness data:3.0]
13 # [BEvent name:ADD_DRY]
14 # [BEvent name:Thickness data:4.0]
15 # [BEvent name:ADD_WET]
16 # [BEvent name:Thickness data:3.0]
17 # [BEvent name:ADD_DRY]
18 # [BEvent name:Thickness data:4.0]
19 # [BEvent name:ADD_WET]
20 # [BEvent name:Thickness data:3.0]
21 # [BEvent name:ADD_WET]
22 # [BEvent name:Thickness data:2.0]
23 # [BEvent name:ADD_WET]
24 # [BEvent name:Thickness data:1.0]
25 # [BEvent name:ADD_WET]
26 # [BEvent name:Thickness data:0.0]

```

4.4.2 Deadlocks

A b-program is deadlocked if it has reached a synchronization point where all requested events are blocked, and there is at least one requested event. In these cases, said b-program cannot advance, even though some of its b-threads do attempt to do so. For example, when running the plain pancake b-program composed with the blueberry addition (Listings 4.1, 4.3, and 4.5), the composed program might end in a situation where b-thread `Blueberries` requests the addition of blueberries, but `BatterThinEnough` blocks the relevant event. Listing 4.11 shows an output of a verification of this program.

Discussion - Deadlocks As event blocking is a central concept in behavioral programming, deadlocks are a recurring concern for developers. In particular, developers must avoid blocking large event sets, such as “all events except X”. To this end, formal analysis for detecting deadlocks is an important aspect of the behavioral programmer’s toolbox.

B-programs that listen to external events require a more subtle approach to deadlock detection. This is because even if at a given synchronization point no b-thread can advance, an external event that is waited-for may be requested by the environment, and allow the b-program to break out of the lock. This creates an important distinction between a b-program and a greater b-program built around it: a b-program can sometimes reach a deadlock as part of its normal operation, since its environment might generate events that will break the lock at some point in the future. A greater b-program should never reach a deadlock, since it also contains an environment simulation.

Formally, we say that a b-program $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, W_i, B_i, H_i \rangle\}_{i=1}^n$ is deadlocked at state s if

$$\underbrace{\bigcup_{i=0}^n R_i(s) \setminus \bigcup_{i=0}^n B_i(s) = \emptyset}_{\text{No selectable events}} \quad \bigwedge \quad \underbrace{\bigcup_{i=0}^n R_i(s) \neq \emptyset}_{\text{Requested events exist}}$$

In some cases, while “Deadlock” can provide a low-level explanation for why things went wrong, it may miss the bigger picture regarding *what* went wrong. In the blueberry addition verification trace example (Listing 4.11), it is not incorrect that the BLUEBERRIES event was requested and blocked, and that consequently the b-program as a whole could not advance. However, a better explanation of the violation would be “we requested blueberries but they were never added”. This type of explanation can be achieved by verifying liveness properties.

Listing 4.12: B-threads for typical liveness properties, expressed in LTL in the b-thread name. B-threads may wait for or request events mentioned by the requirement they model. This is an idea carried over from LSC’s execute/-monitor modality, and is orthogonal to the hot/cold modality (also referred to as must/may).

```

1 bp.registerBThread("◇X", function(){
2   bp.hot(true).sync({request:X}); // execute/initiate
3 });
4 bp.registerBThread("□◇Y", function(){
5   while ( true ) {
6     bp.hot(true).sync({request:Y}); // monitor
7     bp.sync({waitFor:ALL});
8   }
9 });

```

4.4.3 Liveness Properties

Liveness properties, as defined by Baier and Katoen in [3], require infinite runs to conform to a requirement, and do not constrain finite runs. Accordingly, execution traces violating liveness requirements contain an infinite loop or visit an infinite amount of states, while those violating safety requirements are linear. Another difference between safety and liveness properties in BP is that for safety properties, the blocking idiom often allows b-programs to be correct-by-construction: if an event should not happen at a given situation, it can just be blocked. On the other hand, ensuring that a b-program complies with liveness requirements requires verification.

A typical liveness property would be “eventually X happens” or “Y happens infinitely often”. The B-threads in Listing 4.12 express these requirements in BP, by marking a synchronization statements as *hot*. This signals that they must eventually leave said point.

Due to the inherently concurrent nature of behavioral programs, their liveness properties are more elaborate than those of single-threaded programs. At a given synchronization point, some b-threads may be hot (which means that they must eventually leave said point), while other b-threads may stay at that point indefinitely without violating any requirement. Additionally, we need to consider the semantics of a b-program terminating when one or more of its

b-threads is hot.

Verification of liveness properties of a b-program is performed by examining the b-program's state-space graph, and searching for hot cycles and hot terminations. These state-space graph structures are further explained below, and are shown in Figure [4.3](#).

Cold Cycle

A cycle where at least one synchronization point consists of non-hot statements only. These cycles do not violate any liveness requirements, since a b-program can stay at this cold synchronization point indefinitely.

B-Program Hot Cycle

A cycle where all synchronization points contain hot statements, but each b-thread passes through at least one synchronization point where it is non-hot. In these cycles, the b-program complies with all the liveness requirements modeled by individual b-threads. However, as a whole, the b-program must advance infinitely, since staying at each of the cycle's synchronization points indefinitely will violate at least one liveness requirement.

B-Thread Hot Cycle

A cycle where at least one b-thread is hot throughout. An infinite run that follows a b-thread hot cycle violates the liveness properties represented by the requirement b-threads that are always hot at that cycle.

Hot Termination

Technically not a cycle, and in a technical sense does not contain any liveness violations. But it does demonstrate an interesting case, where liveness terminology is useful for describing a safety requirement. Subsection [4.4.6](#) elaborates further on this point.

Given a liveness requirement, it is possible to detect its violation by finding an appropriate hot cycle or termination in the verified b-program state-space graph. The idiomatic case for this would be a b-thread hot cycle, where

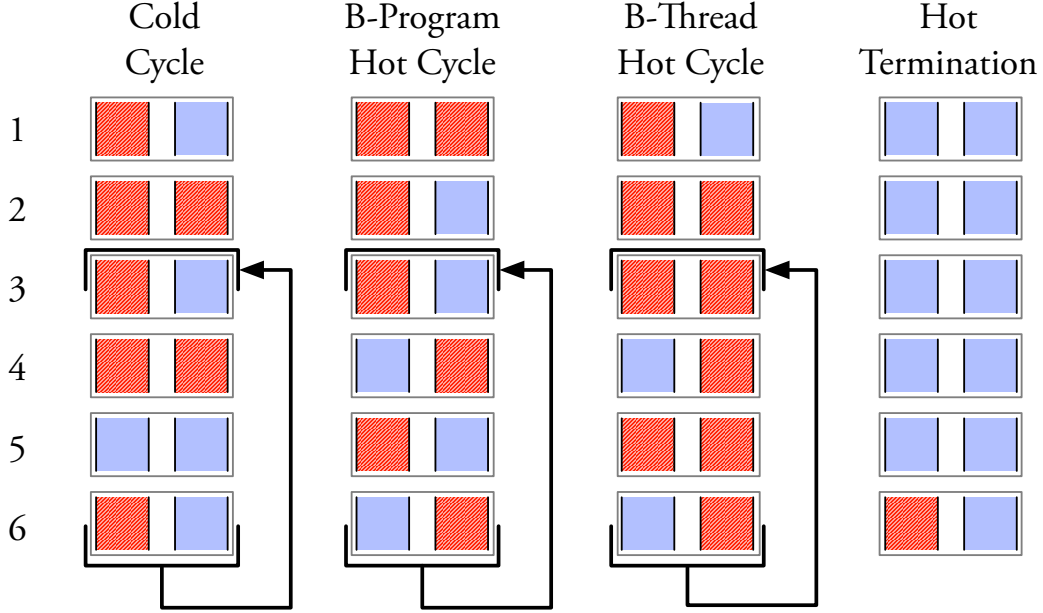


Figure 4.3: State-space graphs of four b-program, demonstrating different structures involving hot synchronization points. All b-programs contain two b-threads throughout. Progression is from top to bottom, where each row is a synchronization point. At each point, hot b-threads appear in dark red, and non-hot b-threads appear in light blue. *Cold Cycles* contain at least a single synchronization point where all b-threads are non-hot. These cycles do not breach any liveness requirements. *B-Program Hot Cycles* are composed of synchronization points that all have at least one hot b-thread (individually, each b-thread may be cold at some synchronization points during the cycle). These cycles may or may not violate liveness requirements, depending on the context and the requirement definition. *B-Thread Hot Cycles* consist of synchronization points in which at least one b-thread is always hot. If said b-thread represents a liveness requirement, a program execution that forever follows this cycle violates the requirement that b-thread models. *Hot termination* is a violation of a safety property, as the b-program run is finite. However, the requirement being violated may be described better in liveness terms.

the hot b-thread is a requirement b-thread modeling the violated requirement directly. An example of such b-thread is **MustAddBluberries**, in Listing 4.7, which simply hot-waits for the **BLUEBERRIES** event. A b-program hot cycle may also signal the violation of a liveness requirement, albeit more implicitly.

Formally, a run $\langle s_1^{(0)}, \dots, s_n^{(0)} \rangle \xrightarrow{e_1} \langle s_1^{(1)}, \dots, s_n^{(1)} \rangle \xrightarrow{e_2} \dots$ of a b-program $\langle \{S_i, E_i, \rightarrow_i, \text{init}_i, R_i, W_i, B_i, H_i\}_{i=1}^n \rangle$:

- Contains a b-thread hot cycle if $\exists i \in [1 \dots b]$ and $t_0 > 0$ such that $H_i(s_i^{(t)}) = 1$ for all $t > t_0$.
- Contains a b-program hot cycle if $\exists t_0 > 0$ such that $\sum_{i=1}^n H_i(s_i^{(t)}) > 0$ for all $t > t_0$.
- Contains a b-program cold cycle if for any $t_0 > 0$ there exists $t > t_0$ such that $\sum_{i=1}^n H_i(s_i^{(t)}) = 0$.
- Contains a hot termination if it is of finite length l and $\sum_{i=1}^n H_i(s_i^{(l)}) > 0$.
- Contains a cold termination if it is of finite length l and $\sum_{i=1}^n H_i(s_i^{(l)}) = 0$.

We now describe each of the structures that may contain violations in detail.

4.4.4 B-Program Hot Cycles

During a b-program hot cycle, each b-thread is non-hot infinitely often. However, at each synchronization point in the cycle, at least one b-thread is hot. Thus, the b-program as a whole must always advance. Consider, as an example, the code for balancing blueberries and kale (Listing 4.8). Here, **KaleAdder** and **BlueberryAdder** add a single dose of their respective ingredients when they detect that the extras ratio is off. They are both hot when they request the addition. If we start the b-program with the addition of half a dose of, say, kale, then these two b-threads will begin to add blueberries and kale to the mix and will never stop, because the fruit index will veer from 0.5 to -0.5 and back indefinitely. Indeed, if we tell the BPjs verifier to search for b-program hot cycles, we will get the report in Listing 4.13.

Listing 4.13: Verification log of the Blueberry/Kale balancer (Listing 4.8), when adding a 0.5 dose of kale. Adding b-threads indefinitely try to balance the respective quantities of blueberry and kale, but to no avail. The b-program as a whole gets into a hot cycle, but each b-thread is also non-hot during this cycle. Some lines have been omitted for brevity.

```

1 # Verification completed.
2 # Found Violation:
3 # Hot cycle violation: returning to index 1 in the trace
4   because of event [BEvent name:ADD_EXTRAS
5                     data:{blueberries=>0.0,kales=>1.0}]
6 # Counter example trace:
7 # [BEvent name:ADD_EXTRAS data:{blueberries=>0.0,kales=>0.5}]
8 # [BEvent name:ADD_EXTRAS data:{blueberries=>1.0,kales=>0.0}]

```

As described here, this infinite addition of ingredients is a bug. However, one can think of similar cases where this is a desired behavior. Examples include the guidance system of a rover, where b-threads correct its course as it follows a target, indefinitely; or a traffic lights system, where some b-threads must ensure that the round-robin queue of the traffic lights progresses indefinitely.

4.4.5 B-Thread Hot Cycle

During a b-thread hot cycle, at least one b-thread is hot during the entire cycle. If said b-thread models a requirement — which is often the case — a run where a b-program executes this cycle forever violates the a requirement modeled by that b-thread.

The pancake server in Listing 4.6 shows an interesting example of this. Composed with the range arbiter and the blueberries b-threads (Listings 4.4 and 4.5 respectively), it may or may not add blueberries at each cycle. If we want to ensure that blueberries are added at least once, we can add the b-thread in Listing 4.14, and verify the composed b-program for liveness properties. Indeed, the verification process finds a b-thread hot cycle, which enables an infinite run where blueberries are never added².

²To ensure that blueberries are added with each cycle, it will be necessary to add a b-thread asserting that a BLUEBERRIES event is selected between each two selections of the

Listing 4.14: A b-thread modeling the requirement that blueberries are eventually added to the pancake batter.

```

1 bp.registerBThread("MustAddBlueberries", function(){
2   bp.hot(true).sync({waitFor:BLUEBERRIES});
3 });

```

Listing 4.15: Verification output for the pancake server (Listing 4.6) composed with the blueberries code (Listing 4.5) and the must-add-blueberries b-thread above. The verification process found a cycle where blueberries are never added to the batter. During this cycle, the `MustAddBluberries` b-thread, whose requirement is being violated, is always hot.

```

1 # Verification completed.
2 # Found Violation:
3 # Hot b-thread cycle violation: b-threads
4   MustAddBlueberries can get to an infinite hot loop.
5   Cycle returns to index 0 because of event [BEvent name:RELEASE_BATTER]
6 # Counter example trace:
7 # [BEvent name:ADD_WET]
8 # [BEvent name:ADD_DRY]
9 # [BEvent name:ADD_DRY]
10 # [BEvent name:ADD_WET]
11 # [BEvent name:ADD_WET]
12 # [BEvent name:ADD_DRY]
13 # [BEvent name:ADD_DRY]
14 # [BEvent name:ADD_DRY]
15 # [BEvent name:ADD_WET]
16 # [BEvent name:ADD_WET]

```

4.4.6 Hot Termination

Hot termination happens when a b-program terminates while one or more of its b-threads are hot. As noted above, liveness properties do not apply with finite runs. Thus, a hot termination is a violation of a safety property. However, describing requirements using liveness terms may allow the verification process to detect the requirements that are being violated at a higher level.

Consider the finite run of the plain batter b-program, composed with the blueberries addition logic. We used this program in our discussion about deadlocks (Subsection 4.4.2). Here, we add the **MustAddBlueberries** b-thread (Listing 4.14) to the b-program, and verify it (Listing 4.16). This time, instead of detecting a deadlock, the verifier returns a counter example, where **MustAddBlueberries** hot-terminates. This is a significant improvement, as a deadlock can explain how a requirement was violated, but does not identify the violated requirement. Hot termination detection, on the other hand, focuses on the requirement that was violated, without considering the technicalities regarding how the violation occurred.

It may be possible to intuit why hot terminations are better phrased as liveness violations by trivially extending finite runs to infinite ones, as follows: Let r be a finite b-program run $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \dots \xrightarrow{e_{n-1}} s_n$. We extend it to an infinite run by adding a self loop at s_n , using a trivial event τ . This make r become $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \dots \xrightarrow{e_{n-1}} s_n \xrightarrow{\tau} s_n \xrightarrow{\tau} s_n \dots$. Extended this way, r becomes a run with a b-thread hot cycle. This cycle violates the requirement violated by the hot termination, but does so as a liveness violation: an infinite run where an event that should happen never does.

Hot-termination analysis is useful from an engineering perspective, as it allows developers to declare that a certain event should be selected without specifying who should request it. This is especially useful for BP, where programs are composed of multiple b-threads and are thus prone to mis-configurations, such as the omission of program parts. In cases where the missing parts are those responsible for requesting an essential event, adding a b-thread that

RELEASE event, and before the first time that it is selected. This, however, is a safety property and not a liveness one.

Listing 4.16: Verification log detecting the hot termination of a blueberry batter mixer b-program, with an added `MustAddBlueberries` b-thread. The counter example returned mentions the violated requirement. This is an improvement over the verifier output in Listing 4.11, which found a deadlock. Both outputs describe the same problem, but the report here explains *what* requirement was violated.

```

1 # Verification completed.
2 # Found Violation:
3 # Hot Termination - The following b-threads were
4   hot when the b-program ended: MustAddBlueberries
5 # Counter example trace:
6 # [BEvent name:ADD_DRY]
7 # [BEvent name:Thickness data:1.0]
8 # [BEvent name:ADD_WET]
9 # [BEvent name:Thickness data:0.0]
10 # [BEvent name:ADD_DRY]
11 # [BEvent name:Thickness data:1.0]
12 # [BEvent name:ADD_DRY]
13 # [BEvent name:Thickness data:2.0]
14 // abbreviated ...

```

hot-waits for that event ensures that mis-configured programs will not pass verification.

Hot termination violations can be phrased as safety properties by adding a special `PROGRAM_DONE` event, which is only selected if there are no other selectable events. Then, we can add a safety property forbidding that event from being selected before the hot-requested event has been selected. e.g. in:

```

1 var e = bp.sync({waitFor:[PROGRAM_DONE,BLUEBERRIES]});
2 bp.ASSERT( e != PROGRAM_DONE );

```

While this description is intuitive, it raises important concerns. First, adding a `PROGRAM_DONE` event introduces a special case (event) into a system that does not yet have any special cases. Second, in order to raise the failed assertion, the listed b-thread must run after the `PROGRAM_DONE` event has been selected. This, in turn, makes the b-threads waiting for this event special cases too, and raises questions such as “Can a b-thread synchronize after receiving a `PROGRAM_DONE` event?”, “Can the `PROGRAM_DONE` event be requested by a regular b-thread?”, and “Can the `PROGRAM_DONE` event be blocked?”.

All in all, this transformation holds; but it muddies the semantics of events

and b-threads, and thus can be considered “non-elegant”.

4.5 Related Work

The work presented here draws its main concept — a *hot* state that a program must eventually leave — from Live Sequence Charts (LSC) [23, 48]. LSC is a variant of Scenario Based Programming, that extends Message Sequence Charts with modalities. Among other things, LSC allows for a message to be labeled as *may be passed* or as *must be passed*. Messages that must be passed are called *hot messages*. Locations where lifelines receive or send hot messages are called *hot locations*. A lifeline at a hot location must, at some point, advance beyond that location.

For live copies of live sequence charts, LSC defines the notion of a *cut*, which maps each of the chart’s lifelines according to its current location. If any of these locations is hot, then the cut is considered as hot as well. This is the equivalent of the concept of *b-program hot cycles* presented here. LSC does not, however, offer a distinction similar to that made here between b-thread and b-program hot cycles.

LSC forbids exiting a chart while its cut is hot. The hot termination concept presented here is similar to this requirement.

In [55], Harel and Segall define the concepts of *local justice satisfaction* and *global justice satisfaction*. A system is said to satisfy an LSC specification in the local justice sense if, in all its possible runs, each of its LSCs are inactive infinitely often. A system is said to satisfy an LSC specification in the global justice sense if, in all its possible runs, all its LSCs are simultaneously inactive infinitely often. The justice concepts are comparable to *b-thread hot cycle* (local) and *b-program hot cycle* (global), even though they do not address liveness directly.

In [68], Klose, Toben, Westphal, and Wittke noted that LTL formulas describing LSCs become prohibitively large for LSCs of moderate size. They proceeded to identify two sub-classes of LSCs that can be verified efficiently: bonded LSCs, which can be verified using an observer automaton and a small

liveness property; and bonded and time-bound LSCs, which can be verified using reachability analysis.

LSC can be verified for liveness properties by translation to LTL, automata, or SMV modules [45]. We are not aware of any way to directly verify an LSC specification. A means of translating LSC to BPjs programs is proposed in [9]. Building on this translation, the work presented here opens another path for verifying the liveness properties of LSCs through model transformation.

The methodology for verifying liveness properties by marking synchronization points as hot and then searching for hot cycles in the b-program state-space graph was sketched by Harel, Lampert, Marron, and Weiss in their work on BPMC, the behavioral programming model checker [46]. This sketch has never been implemented, though. Unlike the present work, BPMC does not distinguish between b-thread hot cycles and b-program hot cycles, and does not offer an equivalent to the hot termination concept presented here.

BPMC and BPjs detect deadlocks in the same way, but differ in their interpretation of a state with no requested events. BPMC identifies this as a deadlock, as it is a state with no successors. BPjs, on the other hand, does not identify it as a deadlock, since no b-thread is blocked. This distinction, however, only holds for safety properties. If a b-program gets to a state where no events are requested but at least one b-thread is hot, BPjs will declare this to be a liveness violation (b-thread hot cycle). Moreover, through its modular nature, BPjs supports BPMC’s deadlock interpretation.

BPC (see Section 2.3) supports indirect program verification by translating b-programs to SPIN models [60].

As presented here, we simulate system environment by adding *environment b-threads* to a greater b-program. These are regular b-threads, and BPjs is not aware of their environment semantics. However, it is possible to distinguish them from the rest of the b-threads using a specialized event selection strategy, e.g. one that selects an event requested by the environment only if there is no selectable event requested by a system b-thread. In [78], Maoz and Sa’ar take a different approach — they extend the LSC language with *assume-guarantee scenarios*. These scenarios allow developers to enrich an LSC specification

with liveness and safety assumptions about the system’s environment.

To the best of our knowledge, the work presented here is the first to offer a method for the direct verification of the liveness properties of behavioral programs. It is also the first to present a distinction between hot b-program cycles and hot b-thread cycles.

4.6 Conclusion

We have presented here a method for verifying the safety and liveness properties of b-programs. The methodology is based on inspecting the state space of a b-program. To detect safety violations, we propose searching for nodes (synchronization points) that contain violations; to detect liveness violations, we propose marking b-threads as *hot* at certain synchronization points, and then searching for hot cycles. We have identified two types of possibly problematic hot cycles: *b-thread hot cycles*, which are always a violation of a liveness property; and *b-program hot cycles*, which may or may not be a violation. We further identified *hot termination* — a case where phrasing the safety requirement of a finite program using liveness terms allows the verification process to identify violated requirements at a higher level of abstraction.

Chapter 5

Semantic Variations using BP

The work described here was published in [9].

Diagrammatic modeling languages hold great promise for software engineering, given their ability to depict — literally — structural and behavioral specifications. While some diagrammatic languages have been adopted in documentation and high-level design, their overall promise still remains largely unrealized with regard to describing executable models. Almost 30 years after Harel first presented StateCharts [38], diagrammatic languages are still considered “doodles” by many practitioners [53].

This chapter proposes a methodology for describing the executable semantics of diagrammatic modeling languages, together with an execution engine based on this definition. In the proposed methodology, languages are defined by pairs of queries and mappers. The queries, defined by a language’s diagrammatic syntax, return language constructs. These constructs are mapped by the mappers to behavioral programming-based models. The resultant definition is executable, can inter-operate with similar definitions of other languages, and can be accessed by practitioners who read code but are less familiar with transition formulae. We demonstrate our approach by defining and creating an execution engine for a subset of the LSC language.

Factors hindering the adoption of diagrammatic languages for execution include the lack of an accessible definition of executable semantics, and the

absence of runtime engines that can inter-operate with text-based code. Our proposed methodology creates language definitions accessible to anyone who can read procedural formulation. Because the diagrams are translated into BP code, they can be embedded in a traditional software system easily — as described in Subsection 3.3.2 (and, in greater detail, in Chapter 6).

The chapter is organized as follows: Sections 5.1 and 5.2 introduce the concept of querying diagrams, and discusses the process of mapping them to BP-based code. Sections 5.3 and 5.4 apply this approach to a subset of the Live Sequence Chart language (LSC). Section 5.5 describes a runtime tool for LSC, implemented in line with these definitions. Section 5.6 demonstrates how the proposed approach can be used to accommodate semantic variation points. Section 5.7 considers related work, and Section 5.8 presents our conclusions.

5.1 Semantic Mapping to BP

We propose translating diagrammatic languages to BP code by using a set of $\langle query, mappers \rangle$ pairs, and a set of BP events which we call *Source-Semantic Events*. The queries, defined using the source language’s terms and semantics, traverse the diagram being translated, and collect diagram elements that fit their criteria. These elements are constructs of the diagrammatic language, and as such have well defined semantics. The mappers then take the collected diagram elements/language constructs, and generate a set of b-threads. We call these b-threads *construct agent b-threads*, or *CABs*, for short.

CABs act on behalf of their construct during program execution (hence their name). Source Semantic Events are used to signal events in the original program, e.g. “message passed” for LSCs, or “step completed” in an Activity Diagram [88]. This mapping process is described in Figure 5.1.

The language that we used in the robot-in-a-house (Subsection 3.3.2) is another example of this approach. There, we used an ASCII drawing, mapping each character to 0 or more b-threads.

The set of query-mapper pairs define the executable semantics of the diagrammatic language. A b-program generated by a set of query-mapper pairs

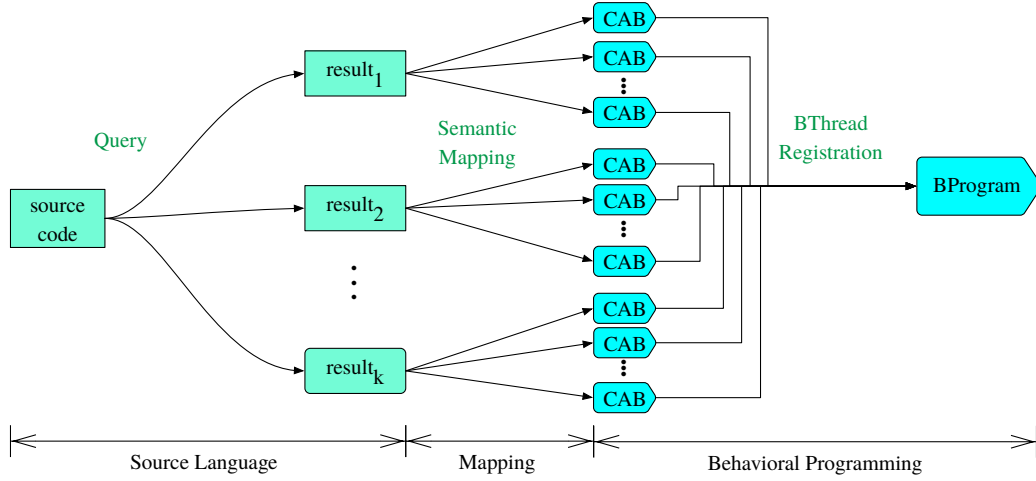


Figure 5.1: Defining executable semantics for diagrammatic languages using querying and mapping. Queries select constructs from a diagram. Selected constructs are mapped to one or more *Construct Agent B-threads (CABs)*. Run together, those CABs generate a valid execution of the original program.

is a valid representation of the mapped source program, in that the order of source-semantic events, in all its possible runs, are in accordance with the definitions of the defined diagrammatic language.

From a BP point of view, there is nothing special about a CAB or a source-semantic event. While both carry special semantics for the diagrammatic program, these semantics are only present during the interpretation of the b-program’s event trace — not during the execution of the b-program.

To summarize the process: first, queries and mappers take a program in a diagrammatic language, and translate it to BP code. The source semantic events in the b-program’s execution trace then map the execution of the b-program to the execution of the diagrammatic program.

5.2 Discussion

Describing the semantics of a formal diagrammatic language by mapping its constructs to BP presents a number of benefits. The resultant definitions are both formal and accessible, as they use simple code snippets rather than

transition formulae. Furthermore, the definitions are executable and verifiable. This allows language developers to test and verify language constructs by using sample programs, which is an effective means of detecting the language inconsistencies that may result from a combination of constructs. Language users benefit from executable definitions, because they can write programs to test their understanding of the language, and experiment with it.

This combination of readability, formality, executability, and verifiability is an improvement on existing practices.

The structure of the resultant definition is intuitive and easy to navigate, because the query-mappers pair structure is similar to that of a language reference. In our experience, many CABs can be reused across multiple constructs. Because CABs define semantics, this is not just regular code reuse, but also concept and comprehension reuse.

Formality, executability, and verifiability are crucial for removing ambiguities. Obviously, non-formal definitions can be ambiguous (e.g. Chan et. al. about Java [20], and Fecher et. al. about UML2.0 [29]). But even fully formal semantics definitions using transition formulae are prone to errors, as shown by Klein et. al. in [67].

The proposed approach will allow language developers to mix and match semantic variations of language constructs independently, as changes to the semantics of a single construct is undertaken by changing the relevant mapper only. Adding and removing language constructs can be experimented with in similar fashion (See Section 5.6).

Finally, because BP is the common denominator for diagrammatic languages defined using the proposed approach, these executable definitions will allow for language interoperation.

The method proposed here is a form of *translational semantics*, using a categorization system proposed by Da Silva [99]. This is as opposed to *structural operations semantics*, which describe semantics through a set of inference rules.

Da Silva further proposed a 2-orthogonal space with formal/informal and executable/non-executable axes. Interestingly, modeling languages with non-

executable semantics, such as UML class diagrams, can still be translated to b-programs in order to impose structural constraints. For example, if a model requires that every car instance has four wheels, this rule can be imposed in BP by introducing a b-thread that listens to car instantiations. When a car is instantiated, the constraint b-thread will spawn a b-thread that hot-waits for the creation of four wheels, and then blocks the creation of additional wheels for that instantiation.

Our proposed approach is, of course, not perfect. When a language construct is mapped to a large number of CABs, the reader must keep in mind the state of those CABs, in order to fully comprehend that construct’s behavior. This challenge is alleviated somewhat by CAB comprehension reuse. Another acknowledged issue is that the definition relies on BP, which is still at an early adoption stage.

5.3 Case Study: Semantic Variations of LSC

We will now demonstrate our approach by defining the operational semantics of a subset of Live Sequence Charts (LSC). LSC is a diagrammatic programming language that extends classical message sequence charts, mainly through universal interpretation and must/may, monitor/execute modalities. The language was developed by Damm and Harel [23], and was first implemented in a tool called Play-Engine [48]. A UML compliant variant is implemented by the PlayGo tool [47]. The semantics of the PlayGo version, which differ slightly from those of Play-Engine, are described in [82].

An LSC system is comprised of scenarios and objects. Each scenario describes a facet of the system’s behavior, and is described in a live sequence chart (an LSC). Overall system behavior is the outcome of the concurrent execution of all of the LSCs the it contains. A model LSC is shown in Figure 5.2

Objects, which appear in LSCs as lifelines, can send messages either to each other or to themselves. Messages are depicted in the charts by horizontal arrows between lifelines. Messages can be tagged as must occur (“hot”, red) or may occur (“cold”, blue), and as executed (“execute”, solid) or waited for

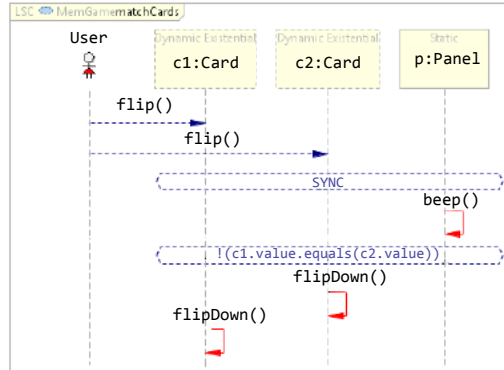


Figure 5.2: LSC describing a basic move in a card memory game. After the user flips two cards, a beep is emitted. If the cards are different, they are flipped back face-down. The first two events may or may not happen, and are thus *cold* (blue). The three subsequent events must take place once the first two events have occurred, and are thus *hot* (red). The **Sync** construct forces the **Beep** to occur following the second **click**.

(“monitor”, dashed). The Play-Engine variant supports both synchronous and asynchronous messages; in PlayGo, all messages are synchronous.

Each LSC consists of lifelines and messages. LSCs may also contain variable assignments and flow-control elements, such as loops and conditional execution. Special lifeline represent the user and the environment. Conditional guards, shown as elongated hexagons, specify statements that must be true in order for the execution to continue. A special condition called *SYNC*, always evaluates to true, and is used to synchronize lifelines.

The point where a lifeline intersects with a message, a condition, or any other language construct is called a *location*. During execution, lifelines proceed along their locations in descending vertical order. The collection of all current lifeline locations in an LSC, called a *cut*, is the equivalent of a program counter in traditional code.

The execution of an LSC consists of a series of events, such as message passing or condition evaluations. An event is considered *enabled* when all of its preconditions have been met — involved lifelines have arrived at their respective locations, variables have been bound, etc. At runtime, the LSC

engine repeatedly selects an enabled event for execution. Lifelines then move to their next locations, and the chart’s cut is updated.

The system avoids the execution of forbidden events whenever possible. Forbidden events can be specified in a number of ways. If a scenario of events ends in a hot false condition, it is considered forbidden. When an LSC is “strict”, all of the events that appear in it but are not enabled by its cut are forbidden. Finally, the Play-Engine variant allows, in some designated circumstances, for the tagging of individual events as forbidden. But if a forbidden event is nevertheless executed, then an exception, called *violation*, occurs.

LSC is an interesting language with which to demonstrate our proposed approach, since it is a real-world diagrammatic language with non-trivial semantics. Additionally, it has multiple semantic variants, which allows us to mix and match the semantics of specific constructs.

This chapter focuses on the operational semantics of a single LSC, and on a subset of its language constructs. A criteria for the selection of the construct subset was that it contained examples for all construct types, and thus could be extended intuitively.

5.4 A Visual Dictionary for LSC

In this section, we present a visual dictionary listing the syntactic query and BP-mapping for each LSC construct. Query matches are highlighted with a yellow background. For example, in the Sync definition (Subsection 5.4.2), the SYNC hexagon and the intersection of its upper edge are matched, and are labeled `snc` and 1_1 to 1_n for the purpose of the BP-mapper pseudo-code that follows. The CABs that compose the BP-mapping for the construct matched by each query appear after the query’s diagram. Re-used CABs are referred to by name, and are listed at Subsection 5.4.11. This graphical representation of the queries, which may allow for the intuitive specification of language constructs, is an idea for future implementations. In the current implementation, we use a textual query language; the graphical representation

is compiled manually, for the sake of readability.

All CABs exit when an exit event from their parent chart is triggered. This is done using the BP idiom of *interrupting events*: b-threads terminate when an event, which is a member of their interrupting event set, is triggered (see Subsection 3.2.1).

BP allows blocking and waiting for abstract sets of events. The model in this chapter uses two such event sets: `VisibleEvents`, which contains all the messages passed; and `ExitEvents(chart)`, which contains all the events signaling the execution termination of a chart or a subchart.

Code listings in this chapter serve both as an implementation and as a specification. Thus, we invite the reader to read them as both imperative and declarative. As an imperative code, `bp.sync(waitFor:E, block:VisibleEvents)` reads as “wait for E, and until then block all visible events”. But when read declaratively, it says “No visible event can be selected until E has been selected”.

5.4.1 Lifeline

A lifeline (see Figure 5.3 for visual representation) represents an object in a chart. Lifeline CABs are responsible for advancing the chart’s cut. When started by its parent chart, a `lifelineCAB` begins by waiting for its parent chart’s start event (first `bp.sync`). It then advances along its locations, requesting repeatedly to enter and leave each one in turn. During execution, a `lifelineCAB` blocks its parent chart from ending normally.

`lifelineCABs` do not enter subcharts. During the execution of subcharts, such as Loop (Subsection 5.4.10), they wait for the subchart to be completed; inside the subchart, new `lifelineCABs` act on their behalf. This is achieved by the `if` statement at the top of the iteration loop.

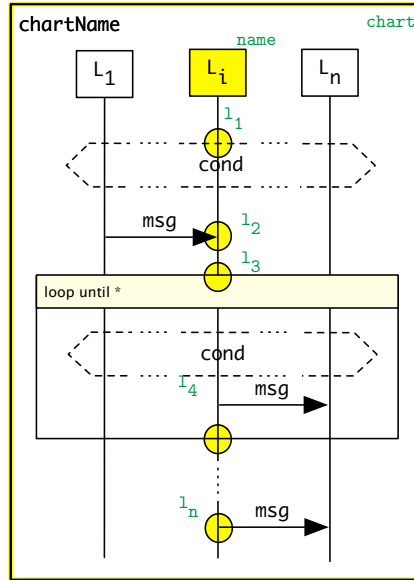


Figure 5.3: Visual Dictionary: an LSC Lifeline

- Lifeline CAB:

```

1 lifelineCAB(chart, l1, ... ln):
2   bp.sync({waitFor: ChartStart(chart)})
3   for ( i ∈ [1..n] ) {
4     if ( li is at bottom of subchart ) {
5       bp.sync({waitFor: Done(subchart),
6               block: ChartEnd(chart)})
7     }
8     bp.sync({request: Enter(li),
9             block: {ChartEnd(chart)} ∪ VisibleEvents})
10    bp.sync({request: Leave(li),
11            block: ChartEnd(chart)})
12  }

```

5.4.2 Sync

Visual representation: See part I of Figure [5.4](#)

- For each $i \in \{1, \dots, n\}$ instantiate a BlockUntilCAB, blocking Enabled(sync) until Enter(l_i) is selected.

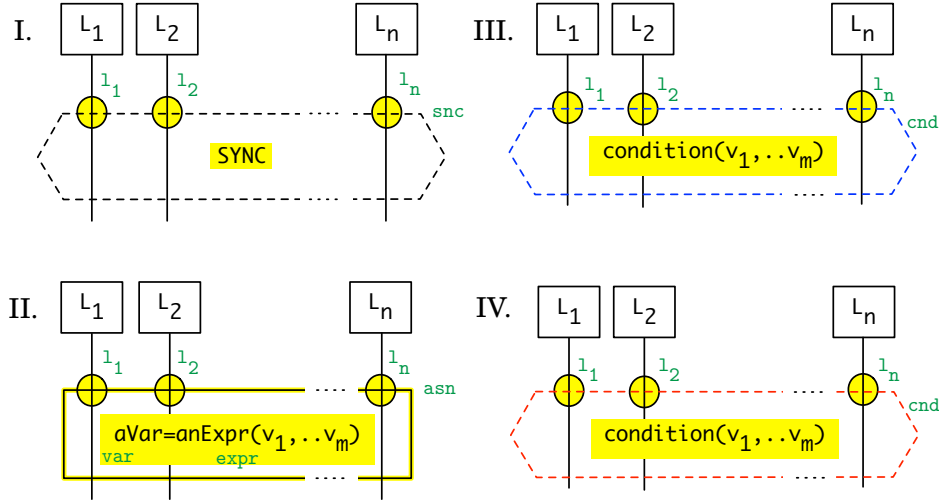


Figure 5.4: Visual dictionary: Synchronization constructs. (I) SYNC, (II) Assignment, (III) Cold Condition, (IV) Hot Condition

- For each $i \in \{1, \dots, n\}$ instantiate a `BlockUntilCAB`, blocking `Leave(li)` until `snc` is selected.
- Instantiate the following CAB:

```

1 syncCAB(snc):
2   bp.sync({request:Enabled(snc), block:Sync(snc)})
3   bp.sync({request:Sync(snc), block:VisibleEvents})

```

This CAB enforces the SYNC to be enabled before it is triggered. By blocking `VisibleEvents` in the second `bp.sync`, this CAB ensures that once enabled, the SYNC will be triggered prior to any visible event.

5.4.3 Assignment

Visual representation: See part II of Figure 5.4. Like SYNCs, Assignments are synchronization points for participating lifelines. Additionally, they may require that values are bound.

- For each $i \in 1, \dots, n$ instantiate a `BlockUntilCAB` with parameters `Enabled(asn)` and `Enter(li)`.

- For each $i \in 1, ..n$ instantiate a BlockUntilCAB with parameters `Leave(li)` and `Assignment(asn)`.
- For each $i \in 1, ..m$. instantiate a BlockUntilCAB with parameters `Enabled(asn)` and `Bound(vi)`.
- Instantiate a single `assignmentCAB` (below) with the matched parameters.

```

1 assignmentCAB( asn, l1, ...ln ):
2   bp.sync({request: Enabled(asn),      block: Assignment(asn)})
3   value = evaluate(expr)
4   bp.sync({request: Bound(var, value), block: VisibleEvents})
5   bp.sync({request: Assignment(asn),   block: VisibleEvents})

```

The `assignmentCAB` forces the assignment event to be enabled before it is triggered. Additionally, this CAB binds the variables assigned to it in the original LSC chart, by evaluating the expression in the assignment construct and publishing the result in an event. This requirement, that assignments occur as early as possible after being enabled, is enforced by this CAB blocking `VisibleEvents` in the latter two `bp.syncs`.

5.4.4 Cold Condition

Visual representation: See part III of Figure [5.4](#).

- For $i \in \{1..n\}$, instantiate a BlockUntilCAB, blocking `Enabled(cnd)` until `Enter(li)` is selected.
- For $i \in \{1..n\}$, instantiate a BlockUntilCAB, blocking `Leave(li)` until `Condition(cnd)` is selected.
- Instantiate a single `coldConditionCAB` (below) with the matched parameters.

```

1 coldConditionCAB( cnd ):
2   bp.sync({request: Enabled(cnd), block: Condition(cnd)})
3   if ( evaluate(cnd) ): resultEvent = Condition(cnd)
4   else: resultEvent = ColdViolation(cnd)
5   bp.sync({request: resultEvent, block: VisibleEvents})

```

5.4.5 Hot Condition

Visual representation: See part IV of Figure 5.4.

- Same as cold condition (Subsection 5.4.4), except that `hotConditionCAB` requests a `HotViolation` event:

```

1 hotConditionCAB( cnd ):
2   bp.sync({request:Enabled(cnd), block:Condition(cnd)})
3   if ( evaluate(cnd) ): resultEvent = Condition(cnd)
4   else: resultEvent = HotViolation(cnd)
5   bp.sync({request:resultEvent, block:VisibleEvents})

```

Messages share a number of aspects with conditions and assignments. They require participating b-threads to synchronize, and may require that variables are bound. Because these aspects are captured independently by CABs, message definitions are mostly CABs that we have encountered in previous sections.

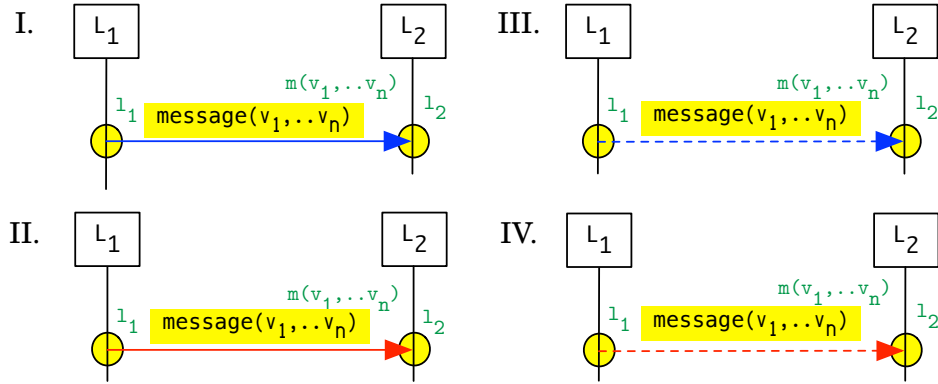


Figure 5.5: Visual Dictionary: Message types. (I) Cold, executed message. (II) Hot, Executed message. (III) Cold, monitored message. (IV) Hot, monitored message.

5.4.6 Cold, Executed Message

Visual representation: See part I of Figure 5.5.

- Instantiate two `BlockUntilCABs`, blocking `Enabled(m)` until `Enter(li)` is selected (for $i \in \{1, 2\}$). If the sender lifeline is also the receiver, it is possible to omit one of these CABs.
- Instantiate two `BlockUntilCABs`, blocking `Leave(li)` for $i \in \{1, 2\}$ until `Message(m)` is selected.
- For each variable v not affected by m , instantiate a `BlockUntilCAB`, blocking `Enabled(m)` until `Bound(v)` is selected. These CABs prevent m from being enabled until all of the variables that it depends upon are bound.
- For each variable v affected (bound) by m , instantiate a `BindFromCAB`, binding v when `Message(m)` is selected. These CABs announce the binding made by m for v .
- Instantiate a single `ceMessageCAB` (shown below):

```

1 ceMessageCAB(m):
2   bp.sync({request:Enabled(m), block:Message(m)})
3   bp.sync({request:Message(m)})

```

This CAB forces the message passing event to be enabled prior to being triggered. The concept of an event being “enabled” is part of the LSC language, and so `Enabled(m)` serves as a source semantic event.

5.4.7 Hot, Executed Message

Visual representation: See part II of Figure 5.5. A “hot” message must be executed once it has been enabled (unlike a “cold” message, which may or may not occur). This difference in behavior is achieved by adding a single CAB, as shown below. The other CABs are reused.

- Same CABs as for the cold, executed message.
- A `TriggeredBlockUntilCAB`, where the trigger event is `Enabled(m)`, after which the event set `ExitEvents(chart)` is blocked until `Message(m)` is selected.

5.4.8 Cold, Monitored Message

Visual representation: See part III of Figure 5.5.

- Except for the `ceMessageCAB`, all CABs used for the cold, executed message (Subsection 5.4.6) are reused “as is” for the cold, monitored one.
- Instantiate a single `cmMessageCAB` (below). This CAB is identical to the `ceMessageCAB`, except for the second `bp.sync` call which waits for the message passing event rather than requests it.

```
1 cmMessageCAB(m):  
2   bp.sync({request:Enabled(m), block:Message(m)})  
3   bp.sync({waitFor:Message(m)})
```

5.4.9 Hot, Monitored Message

Visual representation: See part IV of Figure 5.5.

- Same CABs as for the cold, monitored message.
- A `TriggeredBlockUntilCAB`, as for the hot, executed message.

5.4.10 Loop

Visual representation: See part I of Figure 5.6. An LSC loop is a type of subchart: It can contain any LSC construct, including other loops, but uses the lifelines of its parent chart. A cold violation of a loop will terminate it, but will not terminate its parent chart.

For every loop construct detected by querying an LSC chart, instantiate the following CABs:

- `BlockUntilCAB`, waiting for `Leave(li,1)`, while blocking `Enabled(loop)`.
- A `loopCAB` (see below) with the matched parameters.

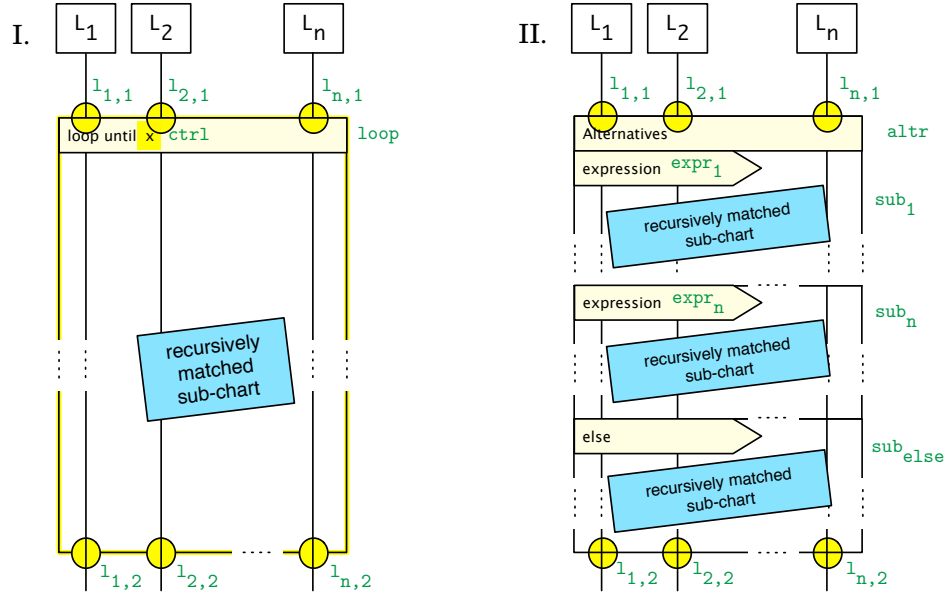


Figure 5.6: Visual Dictionary: Sub-charts. (I) Loop. (II) Alternatives.

```

1 loopCAB( loop, ctrl ):
2     bp.sync({request: Enabled(loop)})
3     loopIteration(loop, ctrl)
4     bp.sync({request: Done(loop), block: VisibleEvents})
5
6 loopIteration( loop, ctrl ):
7     if ( ctrl > 0 ):
8         startCABs(loop)
9         bp.sync(request: ChartStart(loop), block: VisibleEvents)
10        event = bp.sync({request: ChartEnd(loop),
11                          waitFor: ExitEvents(loop)})
12        if ( event is ChartEnd(loop) ):
13            loopIteration(ctrl-1, loop)

```

A `loopCAB` commences by requesting that its loop be enabled. It then runs the loop repeatedly, using the sub-routine `loopIteration`. After its final iteration, it announces that the loop is completed by requesting a `Done(loop)` event. Blocking `VisibleEvents` ensures that once the loop is completed, it is declared as such before any message is passed. The `loopIteration` sub-routine starts by checking whether a new iteration is required. If so, it creates the CABs for the `loop` subchart, and requests that its start event be fired. This

event is a signal for the lifeline CABs of the subchart to start advancing along their location list. When the subchart execution is completed, `loopIteration` checks whether the subchart ran to completion. If so, another iteration is attempted¹. While the loop construct executes, the lifelines of the parent chart `waitFor` it to end before entering their next location, below the chart.

5.4.10.1 Alternatives

Visual representation: See part II of Figure 5.6. Alternatives, the LSC equivalent of an `if-then-else` or `switch` statement, is another type of a subchart. As with loops, it is a fully recursive construct, and can contain instances of itself as well as any other construct.

- For each of the n participating lifelines, a `BlockUntilCAB` with parameters `Enabled(altr)` and `Leave(li,1)` (where $i \in [1..n]$).
- A single `alternativesCAB` (listed below).

```

1 alternativesCAB(altr, l1, ...ln, expr1, ...exprm, sub1, ...subm, subelse ):
2   bp.sync({request: Enabled(altr)})
3   if (subelse ≠ ⊥):
4     toExecute = subelse
5   else:
6     toExecute = ⊥;
7
8   for ( i ∈ [1..m] ):
9     if ( evaluate(expri) ):
10      toExecute = subi
11      break
12
13  if ( toExecute ≠ ⊥ ):
14    startCABs(toExecute)
15    bp.sync({request: ChartStart(toExecute), block: VisibleEvents})
16    bp.sync({request: ChartEnd(toExecute),
17             waitFor: ExitEvents(toExecute)})
18
19  bp.sync({request: Done(altr), block: VisibleEvents})

```

¹For the special case of control value `*`, meaning an unbounded amount of iterations, we define `*-1=*`.

The `alternativesCAB` starts in the usual manner, requesting that its construct is enabled. Once enabled, the `alternativesCAB` selects the subchart to execute, by evaluating the boolean expression guarding each subchart and finding the first expression that evaluates to `true`. If it does not find any such expression, then the CAB defaults to the `sub_else` chart, if present. If a subchart is selected, then the `alternativesCAB` starts its CABs, requests its start event (which, when selected, activates the subchart's lifelines) and then `waitFor` it to terminate. Finally, the `alternativesCAB` announces that the construct has completed execution by requesting a `Done` event.

5.4.11 Reused CABs

The following CABs, used by multiple definitions above, capture common aspects of the LSC language constructs. The fact that we can capture common construct aspects allows for more than just code re-use. It also formally defines underlying construct commonalities — which, in turn, allows for comprehension reuse, a desirable trait when presenting a language (or any other complex set of definitions) to a new audience.

5.4.11.1 BlockUntilCAB

This CAB blocks an event until another one is triggered. Used in all constructs to enforce correct execution order, the code for this CAB consists of a single `bp.sync` call.

```
1 blockUntilCAB( blocked, waitedFor ):
2     bp.sync({waitFor:waitedFor, block:blocked})
```

5.4.11.2 TriggeredBlockUntilCAB

This CAB waits for an event, then blocks a set of events until another event is selected.

```
1 triggeredBlockUntilCAB(trigger, blocked, waitedFor):
2     bp.sync({waitFor:trigger})
3     bp.sync({waitFor:waitedFor, block:blocked})
```

5.4.11.3 BindFromCAB

This CAB is responsible for binding a single variable to a value extracted from a message. It waits for the message to be sent, then requests a binding event announcing the new binding.

```
1 bindFromCAB( message, variable ):  
2     selected = bp.sync({waitFor:message})  
3     value = selected.message.get(variable)  
4     bp.sync({request:Bound(variable, value), block:VisibleEvents})
```

5.5 Implementation

To test our approach, we implemented an LSC runtime engine. The engine takes an XML description of an LSC as its input. It then uses XQuery[96] to both query the source code and to generate BP code. The BP code is then executed normally, using BPjs.

Input LSCs are described using straightforward XML-based format. Similar to a verbal description of an LSC, the format includes nodes such as `<lifeline>` and `<message>`. Figure 5.7 shows a simple LSC and its XML representation, in the format we used in this project.

Queries and BP-Mappers Queries regarding the XML files are done using XQuery. Here, we use the BaseX[37] query engine, which implements the XQuery 3.1 W3C standard [96] without any modifications. As LSCs contain recursive structures, the XQuery program consists of a top-level recursive query, which is used to query the top-level chart. It then recurses down the chart containment hierarchy, querying over each construct it finds and mapping it to BP code. Construct queries look very much like the construct definitions listed above, phrased in XQuery (see Listing 5.1).

Using this engine, LSCs described using our XML format can be directly executed. The code is available at [8].

```

<lsc id="lscWithLoop" name="Sample Loop">
  <lifeline name="A" location-count="3">
    <subchart-bottom subchart-id="theLoop" loc="3"/>
  </lifeline>
  <lifeline name="B" location-count="3">
    <subchart-bottom subchart-id="theLoop" loc="3"/>
  </lifeline>
  <message from="A" fromloc="1" to="B" toloc="1"
    content="hello" temperature="cold" exec="monitor" />
  <loop id="theLoop" control="3" locations="A@2,B@2">
    <lifeline name="A" location-count="1" />
    <lifeline name="B" location-count="1" />
    <message from="B" fromloc="1" to="A" toloc="1"
      content="world" temperature="hot" exec="execute" />
  </loop>
</lsc>

```

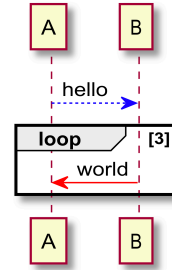


Figure 5.7: A simple LSC with a loop, and its XML representation.

Listing 5.1: XQuery code for detecting `<message>` nodes and generating the BPjs-based Javascript code that implements them. Compare this to the definition of cold, executed message (Subsection 5.4.6). Methods called by this query, such as `lsc:messageCAB`, return Javascript code.

```

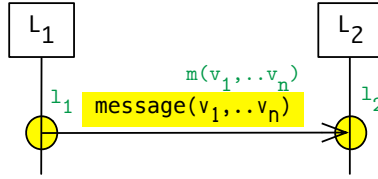
1 declare function local:message( $msg as node() ) as xs:string {
2   (: Generate the JS for the passed message XML node. :)
3   let $fromLoc := lsc:loc($msg/@from, $msg/@fromloc)
4   let $toLoc := lsc:loc($msg/@to, $msg/@toloc)
5   let $content := $msg/@content
6   let $msgEvent := lsc:Message($fromLoc, $toLoc, $content)
7   let $msgEnabled := lsc:Enabled($msgEvent)
8   let $chartId := lsc:chartId($msg/..)
9   return string-join((
10     lsc:blockUntilCAB( $msgEnabled, lsc:Enter($fromLoc, $chartId) ),
11     lsc:blockUntilCAB( $msgEnabled, lsc:Enter($toLoc, $chartId) ),
12     lsc:messageCAB( $fromLoc, $toLoc, $content ),
13     lsc:blockUntilCAB( lsc:Leave($fromLoc, $chartId), $msgEvent ),
14     lsc:blockUntilCAB( lsc:Leave($toLoc, $chartId), $msgEvent )
15   ), $newline )
16 };

```

5.6 Semantic Variations

We will now demonstrate how the proposed mechanism allows for adding, removing, and altering the semantics of each construct independently, along with adding new constructs or removing them altogether.

5.6.1 Asynchronous Message



The messages described in Section 5.4 are synchronous. We now add an *asynchronous* message, which will allow the sending lifeline to advance beyond the send location without having to wait for the receiving lifeline to receive the message. This type of message only exists in the Play-Engine variant; it is not supported by PlayGo.

Instantiate the following CABs:

- BlockUntilCAB, blocking Enabled(m), waiting for Enter(l1).
- BlockUntilCAB, blocking Received(m), waiting for Enter(l2).
- BlockUntilCABs, blocking Received(m), waiting for Sent(m).
- BlockUntilCAB, blocking Leave(l1), waiting for Sent(m).
- BlockUntilCAB, blocking Leave(l2), waiting for Received(m).
- For each variable v not affected by m , a BlockUntilCAB, blocking Enabled(m), waiting for Bound(v).
- For each variable v affected (bound) by m , a BindFromCAB, blocking Sent(m), waiting for v .

- A single `asyncMessageSendCAB` (shown below), forcing the message sending event to be enabled prior to being triggered.

```

1 asyncMessageSendCAB(m):
2   bp.sync({request:Enabled(m), block:Sent(m)})
3   bp.sync({request:Sent(m)})

```

- A single `asyncMessageReceiveCAB`, requesting that the message is received.

```

1 asyncMessageReceiveCAB(m):
2   bp.sync({request:Received(m)})

```

As there are `blockUntilCABs` blocking the message from being received prior to being enabled, fully bound, sent, and having its receiving lifeline in location to receive it, the `asyncMessageReceiveCAB` does not need to handle these preconditions and their possible orderings.

The mapping for asynchronous messages can be used either as a replacement for the synchronous message mapping, making all messages asynchronous (e.g. in a “what if we made all messages asynchronous” scenario), or as a mapping for a new type of specification idiom.

5.6.2 Strict vs. Tolerant Modes

An LSC can be either *strict* or *tolerant*. A Strict LSC is violated if an event that appears in it occurs when it is not enabled. An event of this nature will not cause a violation in a tolerant LSC. In PlayGO, all of LSCs are strict. In Play-Engine, both modes are allowed. Strictness can be imposed by adding b-threads to the mapping of a chart, one for each message appearing in the LSC.

Each b-thread is initialized with the events that are present in its respective chart. The function `cut(chart)` returns the cut of `chart`, which is the set of all of the locations that its lifelines are currently located in. Obtaining the cut of a given chart does not require direct communication with that chart or its lifelines — the cut can be obtained by waiting for the `Enter` events of that chart’s locations. A cut is considered **HOT** if at least one of its locations is **HOT**.

Listing 5.2: Enstrictor, a b-thread that makes an LSC strict

```

1 chartEvents = events_of( chart )
2 nonBlocked = {}
3 repeat:
4     event = bp.sync({waitFor: AllEvents,
5                     block: chartEvents \ nonBlocked})
6     if ( event is Enabled(x) ):
7         nonBlocked = nonBlocked ∪ {x}
8     else if ( event ∈ nonBlocked ):
9         nonBlocked = nonBlocked \ {last_event}
10    else:
11        if ( isHot(cut(chart)) ):
12            bp.sync({request: HotViolation, block: VisibleEvents})
13        else:
14            bp.sync({request: ColdViolation, block: VisibleEvents})

```

5.6.3 Adding a Type System by Blocking

The LSC implementation presented has used dynamic typing so far, since it does not verify that the receiving lifelines are implementing the messages they have received. Using event blocking, we can block messages unimplemented by their receiver in a purely incremental manner, by adding a b-thread.

BP allows b-threads to block sets of events by passing a predicate to `bp.sync`. `InvalidMessages`, defined in Listing 5.3, is a predicate valid for all messages unimplemented by their receiver. In order to prevent these messages, the `typeSystemBThread` can be added to the system.

Listing 5.3: Type System Event Set and BThread. This code assumes each lifeline has a list of defined operations

```

1 InvalidMessages( msgEvent ):
2     return
3     msgEvent.message ∉ msgEvent.receiver.definedMessages
4
5 typeSystemBThread:
6     bp.sync({block: InvalidMessages})

```

Traditionally, when typing constraints are imposed on a program, they are imposed across the program as a whole. Our proposed approach, of imposing typing constraints, offers a greater degree of flexibility, in that it can be lifted or imposed without making changes to the rest of the code. It can be limited to parts of the code by altering the predicate; alternatively, it can pass the invalid

method call to a special handler equipped to perform any arbitrary operation. This is similar to SmallTalk’s `doesNotUnderstand:` method, which is invoked when an object receives a message it does not have an implementation for.

5.7 Related Work

The notion of describing semantics by mapping one domain onto another is not new. AToM3 [70], for example, a tool for creating and transforming meta-models, uses graph-based meta-models and transformations to achieve this objective. UML [88] has its own metamodel, used to describe its diagrams. Executable UML [86, 92] adds executable semantics to a subset of UML’s diagrams. Our work differs from both in that it does not use a meta-model per se — the queries extract data from the source language, but the result is presented in the source language, and not in a metamodel.

In [71], Latombe et. al. presented a way of coping with semantic variations in domain specific modeling languages. Their method uses a 2-tier structure, where the top tier lists all options according to a set of available semantics, and the lower level selects the correct option according to the desired semantic variant. Our approach differs in that it uses changes in queries and mappers to vary the semantics. Thus, options not available to the selected variant are not generated.

Multiple semantic variants of the LSC language have developed over the years. Cohen and Maoz [22] use a feature model (containing 19 features) in order to consolidate them under a single, configurable semantic interpretation. The solution presented there uses model transformation, taking an LSC specification and a semantic configuration, and generates an automaton that is later analyzed using an external tool (e.g. GOAL). Our proposed approach is comparable, with the set of model queries and semantic mappers taking the role of the feature model and translation algorithm, and the resultant b-program replacing the generated automaton. Compared to our queries and semantic mappers, feature models offer a more structured way of comparing semantic variants. For the same reason, queries and mappers allow for more agility and

experimentation, which are important features during language development.

This project was also partly motivated by [31]’s call for endowing the conventions behind complex diagrams (biological ones, in the case of [31]) with explicit formal semantics.

5.8 Conclusion

By querying the syntax of a diagrammatic language and mapping the result to BP-based code, we can formally define the operational semantics of said language. The resultant definition is executable, accessible to practitioners who read code but are less familiar with state change formulae, and allows the language developer to experiment with different semantics of language constructs independently. We have demonstrated the proposed approach using a subset of the LSC language, and presented a working tool based on this definition.

Chapter 6

BP Model Driven Engineering

The work described in this chapter was previously published in [35] and [5].

Model-Driven Engineering (MDE) is a well-known approach for building software systems. A model is an abstraction of a system that one wishes to study. It contains the system’s relevant aspects, omitting aspects irrelevant for the study that the model was made for. As such, models facilitate communication between technical and non-technical stakeholders [25, 99]. Modeling is used in many engineering domains, including Chemistry, Physics, and Economics. In software design, modeling has been used since the late 1960s as a tool to facilitate reasoning about behavior and correctness, beginning with research by Floyd [30] and Hoare [59]. Models were subsequently used in design and documentation, OMG’s Unified Modeling Language [88] being the most prominent example.

While many modeling languages have formal semantics, practitioners often think of them as generally helpful doodles, rather than as formal languages. In [53], Harel calls this “the doodling phenomenon”. Other researchers have advocated for embracing models with their full semantics as well (e.g. Seidewitz [98] and Lee [72]).

In recent years, models have become more central to the software building process, with new techniques and methodologies using models not only for documentation, but also in execution and testing [99]. These practices are

known as “Model-Driven Engineering (MDE)”. When using MDE, engineers develop a system by creating an executable model that describes it, using modeling languages such as SysML [56], or Umple [73]. Models are often integrated with the rest of the system through code generation — the modeling tool uses the model to generate code in an “ordinary” language (such as Java or C#). The generated code is then compiled and packaged with the system, like traditionally-developed code. Other integration methods have also been explored; for example, LSC models developed in PlayGO [47] can be integrated with ordinary Java code, using aspect-oriented programming [76, 77].

Model-driven engineering allows for describing the developed systems at a higher level, as compared to regular code. It also allows for the simulation and verification of certain model properties, which help detect requirement and design issues at an early stage in the development process.

It is no surprise that the modeling language used in a given MDE project can be crucial for that project’s success. A survey conducted by Whittle et al. [101] found that the formal semantics and theoretical foundations of modeling languages are key to their subsequent adoption, as well as their conformance with human abstractions.

Because BP is well-founded, has clear semantics (see Sections 2.2 and 4.3), and aligns well with human logic processes and thought [33], it is a natural candidate for creating models in MDE setting. Other important properties of BP include its executability, amenability to formal verification (see Chapter 4), alignment with stated requirements, and developmental incrementality (see Chapter 2). These properties allow engineers to translate behavioral concerns from system requirements directly into operational software. This translation further enables the straightforward tracing of bugs to specification inconsistencies.

BP allows for incremental extensions and refinements, which makes it possible for system developers to begin the modeling process at an abstract level, and to add implementation details later. It supports a trial-and-error style development that is often essential for developing embedded software. Models can integrate with traditional systems in multiple ways, including the afore-

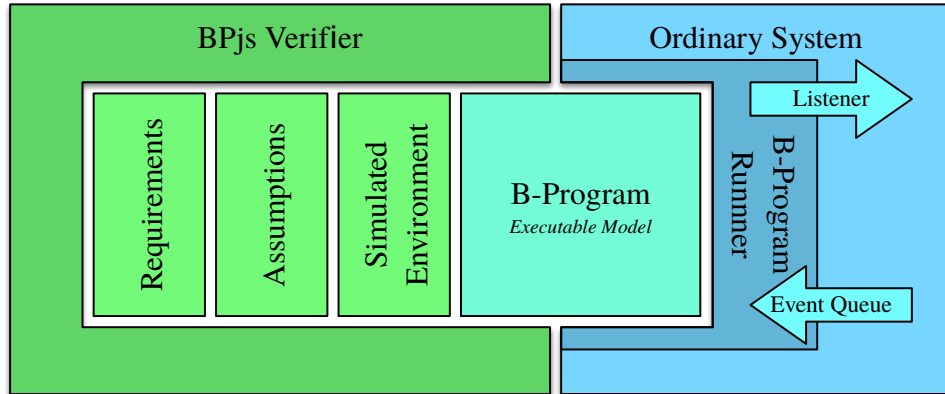


Figure 6.1: A b-program used as a model in Model-Driven Engineering. During verification (left), the model is analyzed by a verifier. Additional b-threads may be added for simulating the system environment, adding assumptions in order to limit verification search space, and adding additional requirement b-threads not already contained in the model. During runtime (right), the b-program is put in a b-program runner, which serves as an adapter between the model and an ordinary software system. The same b-program is used for both runtime and verification, which eliminates the bugs that may emerge during model translation.

mentioned aspects, and an Observer interface. Finally, code-based models are inherently testable, and can be formally analyzed, e.g., by model checking.

During model testing and verification, BP’s incrementality can be used to simulate a system’s environment, to formally take assumptions into account, and to formally describe required system properties. Our proposed design for embedding b-programs in traditional software systems allows the use of the same model during both verification and runtime, thereby eliminating — by design — the bugs and inconsistencies that might otherwise creep into the system during a model translation process. Figure 6.1 shows a b-program used as a model, during verification and runtime.

Not every program written using BP is a model. In particular, if b-threads interact with their environment or with each other directly (rather than through events), the b-program they partake in cannot be translated to a transition system as described in Section 4.3. As a result, that b-program cannot be formally analyzed or verified using the techniques presented in this

work, and thus cannot be considered “a model” under the definitions we propose. Examples of such direct interaction include using shared memory structures (even those supporting concurrent access), or directly accessing external resources such as a database or a file system.

For a b-program to be a model, it has to use “pure BP”: All its b-threads must interact with each other and with their environment using events only. When a b-thread has to interact with its environment, it should do so through the b-program’s host program, with which it can communicate using events. The host listens to events selected by the b-program and can take actions according to these events. When a host program needs to send data to a “pure” b-program, it can only do so by enqueueing events in its external event queue. As an example, if a b-thread needs to access the contents of a file, it can request a `READ_FILE` event, and then wait for a `FILE_CONTENTS` event. The host would intercept the `READ_FILE` event, read the file contents, wrap it in a `FILE_CONTENTS` event, and send it back to the b-program.

From an engineering perspective, the fact that not all b-programs are pure by definition is a strong point for BP, as it allows software engineers to decide if they want to program, model, or use a hybrid approach. This chapter discusses the use of “pure” BP as a model in Model-Driven engineering, to examine whether BP is usable under the constraints this purity imposes. This examination is done through two examples: a simulated autonomous rover, and a satellite.

This chapter is organized as follows: Section [6.1](#) presents a simulated autonomous rover, designed using MDE with a BP model. Section [6.2](#) presents a cube satellite designed and tested using MDE and BP. Section [6.3](#) considers related research studies, and Section [6.4](#) presents our conclusions.

6.1 Tracking Rover Control

This section presents a solution to a challenge published by the 2018 edition of MDETools workshop, which called for MDE-based implementations of control software for a simulated rover. Said rover was to track another rover (“leader”)

at a safe distance — not too far away, but not too close. The lead rover could change direction and speed, and the tracking rover had to respond to these changes, in order to remain within safe range¹

Our proposed model for the tracking rover’s control software is a b-program, consisting of a set of b-threads, each corresponding to a specific requirement. The model development process relies on BP’s incrementality: we start by defining b-threads for specific requirements. Then, we refine the model, based on results obtained from verification and trial and error.

We initially focus on these two requirements:

1. The rover has to follow the leader
2. The rover should always stay at a safe distance from the leader

Both requirements can generate instructions for the tracking rover’s wheels. These requirements may conflict with one another; for example, when the leading rover has steered to one side, and the tracking rover must both go forward and turn in order to maintain its position. Listing 6.1 illustrates how blocking can be used to integrate such competing requirements, when isolated in separate b-threads.

The first line imports a Java package which abstracts from the interaction between the scenario-based model and the (simulated) rover. Specifically, the Java host application sends `Telemetry` events that carry the rovers’ telemetry data. The host also translates actuation events `TURN_LEFT`, `TURN_RIGHT`, `GO_TO_TARGET`, and `GoSlowGradient(power)` into the commands sent to the rovers’ wheels. Behavioral abstraction layers of this nature allow models to separate key problem aspects from platform-specific technical details.

Line 3 defines an event set, that is used to wait for all `Telemetry` events. Since these events differ in their data, a b-thread cannot list all of them individually. Instead, it can wait for a set of events which satisfy a given predicate.

¹The full challenge can be found here:

<https://mdetools.github.io/mdetools18/challengeproblem.html>

Listing 6.1: Follower Rover Control program in BPjs (abbreviated for brevity)

```

1 importPackage(Packages.il.ac.bgu.cs.bp.leaderfollower.events);
2
3 var AnyTelemetry = bp.EventSet("Telemetries", function (e) {
4     return e instanceof Telemetry;
5 });
6
7 var esFBwardEvents = bp.EventSet("FBwardEvents", function (e) {...});
8
9 bp.registerBThread("Go", function () {
10     while (true) {
11         bp.sync({waitFor: AnyTelemetry});
12         bp.sync({request: StaticEvents.GO_TO_TARGET});
13     });
14
15 bp.registerBThread("SpinToTarget", function () {
16     while (true) {
17         var et = bp.sync({waitFor: AnyTelemetry});
18         var degToTarget = compDegToTarget(et.LeadX, et.LeadY,
19                                         et.RovX, et.RovY, et.Compass);
20         while (Math.abs(degToTarget) > 4) { // must correct rover orientation
21             if (degToTarget > 0) {
22                 bp.sync({request: StaticEvents.TURN_RIGHT, block: esFBwardEvents});
23             } else {
24                 bp.sync({request: StaticEvents.TURN_LEFT, block: esFBwardEvents});
25             }
26             et = bp.sync({waitFor: AnyTelemetry, block: esFBwardEvents});
27             degToTarget = compDegToTarget(et.LeadX, et.LeadY,
28                                         et.RovX, et.RovY, et.Compass);
29         });
30     });
31
32 var tooClose = 12.5;
33 var tooFar = 15;
34
35 bp.registerBThread("NotTooClose", function () {
36     while (true) {
37         var lastTelemetry = bp.sync({waitFor: AnyTelemetry});
38         while (lastTelemetry.Dist < tooFar) {
39             if (lastTelemetry.Dist >= tooClose-(tooFar-tooClose)) {
40                 var slowDownPower=Math.round((lastTelemetry.Dist-tooClose)/
41                                             (tooFar-tooClose)*100);
42                 bp.sync({waitFor: [StaticEvents.TURN_RIGHT,StaticEvents.TURN_LEFT],
43                         request: GoSlowGradient(slowDownPower),
44                         block: StaticEvents.GO_TO_TARGET});
45             } else {
46                 bp.sync({waitFor: [StaticEvents.TURN_RIGHT,StaticEvents.TURN_LEFT],
47                         request: GoSlowGradient(-100),
48                         block: StaticEvents.GO_TO_TARGET});
49             }
50             lastTelemetry = bp.sync({waitFor: AnyTelemetry,
51                                     block: StaticEvents.GO_TO_TARGET});
52         });
53     });
54
55 function compDegToTarget(xL, yL, xR, yR, CompassDeg) {...}

```


Using BPs’ `bp.EventSet`, the predicate returns `true` if and only if the event `e` is an instance of class `Telemetry`. This technique is also used in line 7, for waiting for all forward/backwards movement events.

Lines 9 to 13 are a b-thread modeling the requirement that the tracking rover should drive forward, to follow the leader. It is a two-step scenario: it waits for a telemetry event, and reacts by requesting the `GO_TO_TARGET` event.

Lines 15 to 27 are a b-thread for the requirement that the tracking rover orient itself in relation to the leader before driving towards it. The scenario uses a helper function `compDegToTarget`, which computes the angle in degrees between the tracker and the leader. If the value of angle falls outside a defined range, the b-thread requests the event of turning right (or left), while blocking all forward/backward movement events. The abstraction layer changes wheel speed to actuate the turn, and the orientation is checked again. When the angle falls within the allowed range, i.e., the tracker is oriented in the direction of the leader, forward movement is no longer blocked. Note that the b-thread is modeled so as to be as self-contained as possible.

Process-wise, we started by running the two b-threads described above, to see how they performed. After a short debug process, we realized that these b-threads were not enough. While the tracker did indeed follow the leader, it sometimes got too close to it, violating a requirement that had not yet been implemented. We could, of course, adjust the existing b-threads such that the `GO_TO_TARGET` event could not be requested if the distance between the tracker the leader was below a certain threshold — which is how such issues are usually dealt with in standard modeling and programming languages. However, as we were using SBM, and this could be viewed as a separate behavioral concern, we decided instead to implement this requirement in a separate b-thread, called `NotTooClose`. This b-thread, in lines 34 to 51, is an “anti-scenario” — a b-thread that uses the BP event blocking idiom to forbid the `GO_TO_TARGET` event (which means traveling at maximum speed) if the distance of the tracker from the leader is too short. Instead of the blocked event, this b-thread uses feedback control logic to request a `GoSlowGradient(power)` event, which adjusts wheel speed in relation to the tracker’s distance from the leader. The parameter

`power` can be positive or negative, allowing the tracker to drive in reverse if necessary.

B-thread `NotTooClose` uses the *break-upon* idiom (see Section 5.4) to ensure that the `GoSlowGradient` events it requests are based on current data. Section 6.1.1 presents a way for automatically verifying that this is indeed the case.

While small and simple, this model already demonstrates how blocking can be used to integrate positive and negative specifications if new requirements emerge during the development process. As demonstrated, this is especially useful for command and control software, where typical trial-and-error creates new requirements, which are better implemented in separate b-threads for maintenance and readability purposes.

The code for the rover is available at [97]. A video of the model in action is available online at <https://vimeo.com/299312428>.

6.1.1 Rover Formal Analysis

A key advantage of SBM is that the models it creates are amenable to formal analysis, which can reveal conflicts and omissions between system requirements and design decisions. Such analysis can be performed from the earliest stages of prototype development.

Model analysis is enabled mainly by abstracting away sensors and actuators, and focusing on scenarios that represent basic requirements (as well as exceptions and refinements thereof), and on the interaction and composition of these scenarios. During verification, a b-program cannot interact with its environment, since its execution is non-linear, and the environment (which is not aware of the verification) will not be synchronized with the verified program once backtracking begins. However, relevant aspects of environment behavior can be modeled as additional scenarios. This also presents an opportunity for abstracting much of the detail of the real-world environment. Such environment scenarios can also monitor real environment behavior at run time, and give advance warning of situations that the system has not formally been

prepared for.

We performed model verification using BPjs’ built-in verifier (see Chapter 4). We added environment b-threads that simulated a leader moving, the generated telemetries based on geometry alone (that is, we only took into consideration wheel power and orientation, not friction or elevation). We also added b-threads responsible for raising false assertions when safety requirements are violated. Because this type of problem — two rovers driving on an infinite plane — has a state space of infinite size, we limited the depth of the DFS state space exploration to 500 events. Without this limitation, the verifier would follow a single program execution; because it uses DFS for traversing the b-program state-space, this algorithm would lead it infinitely deeper into the simulation b-program state-space graph.

This simple setting allowed us to detect, for example, that if the tracker’s speed cannot exceed 120% of the leader’s speed, then the distance between the vehicles will extend beyond the threshold. We also detected the absence of the “break upon” part, as discussed above, by adding a b-thread that waits for `GoSlowGradient` events and confirms that their power field is consistent with the latest telemetry data. The result was an automatically generated trace of the system, with a `GoSlowGradient` appearing at a wrong time. We used this trace to generate Figure 6.2². After fixing `NotTooClose` (by having it wait-for turn events, as described above), we used verification to prove that the problem had indeed been solved.

6.1.2 Discussion and Conclusion

The follower rover use-case presented here demonstrates the advantages of using BP for model-driven engineering: the intuitiveness of b-threads, alignment of b-threads with requirements and design documents, weaving-in of new b-threads with little or no change to existing ones, formal verification capabilities, and direct executability of the model, without transformation or synthesis. These advantages carry over to SBP in general [35].

²Figure was generated by processing the event trace to GraphViz’s Dot language. Annotations and enlargements were added manually.

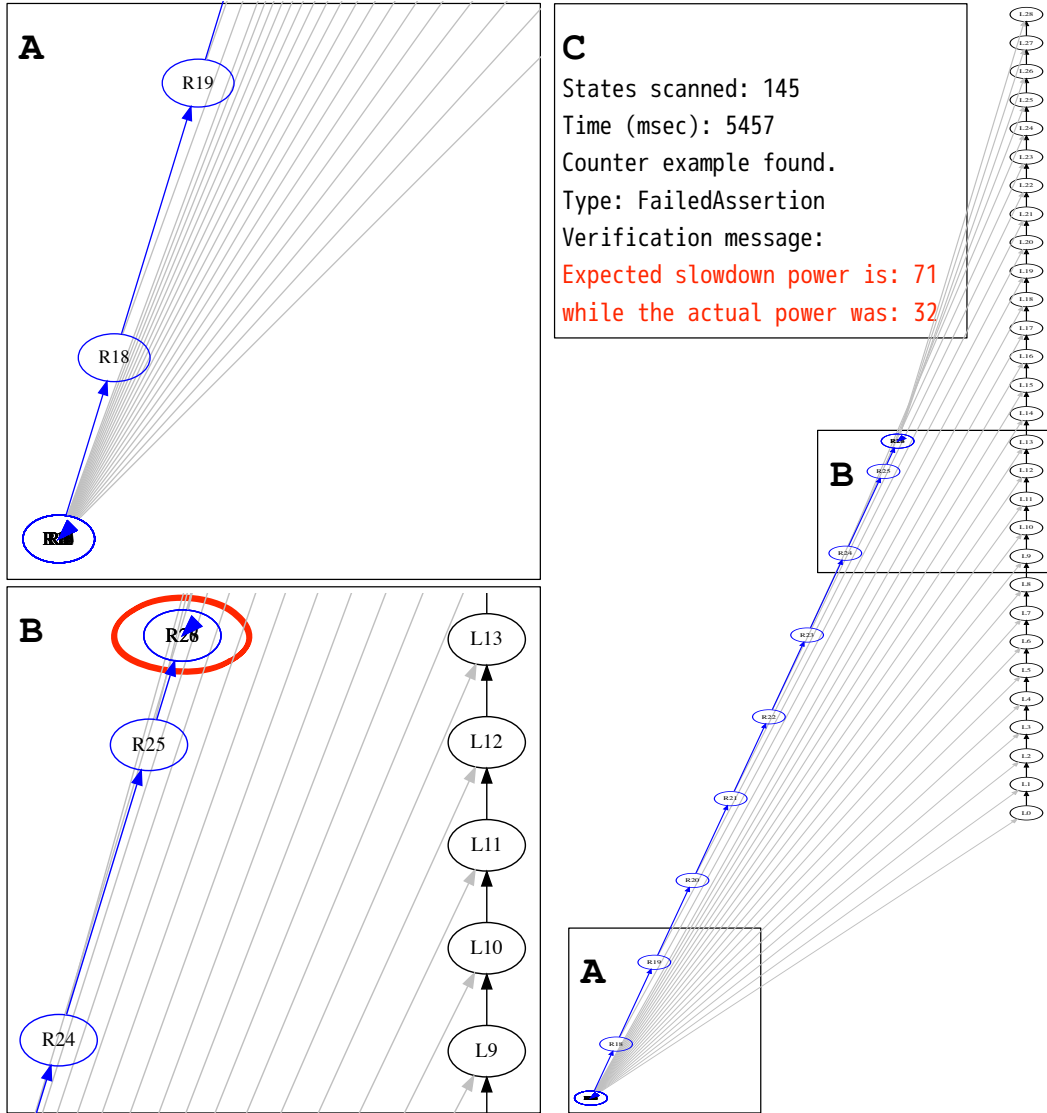


Figure 6.2: A race condition caught by verification. The leader rover (black ovals) moves forward at a steady pace. The faulty tracker rover (blue ovals) follows it. To align itself behind the leader, the tracker must move closer to it, and change its orientation. It starts by turning (frame A), and then moves forward at full speed. When it gets close enough to the leader (frame B), it must both slow down and turn. At this point, the `NotTooClose` b-thread requests a `GoSlowGradient` event, while the `SpinToTarget` b-thread requests a turn. The b-program selects the turn event first. By the time the `GoSlowGradient` event is selected, the leader is further away from the rover, and so the power field of the `GoSlowGradient` event is smaller than expected (frame C).

The model presented here relies on blocking-based scenario composition. The **Go** scenario is a default behavior, and causes the tracker rover to drive forward at full speed. This behavior is blocked when there are overriding concerns. The **SpinToTarget** b-thread blocks it when the tracker is not oriented in the direction of the leader; the **NotTooClose** b-thread blocks it when the tracker is too close to the leader, and needs to slow down. Other composition styles are also possible. For example, a wait-for-based composition may have a b-thread wait for a telemetry event, determine whether the rover needs to turn or go forward, and publish an event accordingly. Two other b-threads may wait for forward and turn events, calculate the required amount (wheel power and degrees, respectively), and request events that instruct the wheels to act accordingly (Greenyer et al. [35] presents a similar model, implemented using IOSM-K). These composition styles are not mutually exclusive.

6.2 On-Board Satellite Control and Testing

We now turn to another example of model-driven engineering using BP (and BPjs) — an on-board control software for a satellite. But first, some background on space missions.

A common method of increasing the efficiency and productivity of satellites and other space missions is to make them more autonomous, by improving their on-board control and troubleshooting capabilities. As a result, these machines can be expected to only become more software intensive. Since the 1990s, NASA and others have been exploring artificial intelligence approaches and algorithms for ground control systems and for avionics. The goal of these examinations was to automate aspects of space missions, in order to reduce the number of experts needed to control and coordinate missions, as well as to improve system performance and resilience. This was followed by research into agent-based methods and control theory concepts, directed towards improving self-management and survivability in the harsh environments in which space missions operate. While software-based systems responsible for complex decisions are common in other fields, introducing these advanced features to the

field of space missions is a challenge, because space systems must maintain an uncompromisingly high standard of reliability. In a sense, general software engineering is moving away from the culture that defines space-mission software development. While techniques such as agile programming, adaptive planning, evolutionary development, early delivery, and continuous integration have many benefits, they cannot be directly implemented in the context of space mission software development, due to the strict safety and robustness requirements expected from space software.

The strict requirement for safety and robustness has nudged the space industry into relying heavily on systems with “heritage” — that is, systems that have already been successfully deployed in space missions. Understandable though this may, this approach slows down system development progress, because new technologies must be introduced into secondary components, and be launched into space, before they can be used in primary components. However, we have seen that BP can be used to create reactive systems whose robustness and safety can be verified. Thus, using BP for developing space software can accelerate and modernize space software engineering, and without sacrificing the important safety quotient.

In this section, we propose a novel approach to satellite-software development that allows for modularity and formal verification. These, in turn, enable safer and more robust satellite software. Specifically, we propose to use *scenario-based programming*, where software components (modules) represent different aspects of mission and housekeeping scenarios and anti-scenarios (things that must not happen). We present examples of how specifications for flight management can be translated into code artifacts, representing them in a direct and intuitive way. We support this approach by presenting a development environment we are designing for creating *on-board mission software*. Our environment includes an automatic model-checking tool for verifying the developed software against a formal specification. Unlike traditional testing, this tool allows for an exhaustive analysis of the code, producing formal guarantees of quality. Moreover, the system can generate complex realistic test scenarios, by composing simple ones. This improves testing efficiency and cov-

erage by increasing the amount of system states being tested. Subsection [6.2.2](#) demonstrates verification on specific parts of a model, on specific logical layers of an application, and on an entire model at a specific abstraction level. This allows for the introduction of modular design processes, where modules, layers, and aspects of behavior can be tested and verified in isolation as soon as their code is ready.

Additionally, we describe a “hybrid laboratory” for the advanced testing of mission software. Our laboratory uses a novel approach, which allows for the automatic generation of test scenarios using scenario-based programming. While the examples we provide here have been simplified for the purpose of this section, our group at Ben-Gurion University is presently developing a complete on-board mission software suite for a satellite. The experience that has been yielded by this project shows that the development environment, along with the hybrid laboratory, provide a viable tool-set for the development of reliable satellite software.

6.2.1 Managing Satellite Energy and Altitude Using BP

We will now demonstrate how BPjs can be used for developing software modules to manage two satellite sub-systems. These modules are part of an on-board satellite control software suite that we are currently in the process of developing. When completed, this software will control all aspects of a satellite in orbit, including interaction with its sub-systems, and mission and maintenance tasks.

The first subsystem is *EPS* (Electric Power System). For this example, we assume that our EPS can switch between three operational modes: **good**, **low**, and **critical**, depending on the battery voltage. The second subsystem is *ADCS* (Attitude Determination and Control System); again, we assume it can switch between three attitude control modes:

1. **detumbling**: used for reducing angular rates, usually after deployment of the satellite
2. **sun pointing**: used for pointing the satellite’s solar panels towards the

sun, in order to charge its batteries

3. **payload pointing**: used for pointing the satellite’s payload toward a desired set point

The ADCS is activated when a payload pass is scheduled. For this example, we assume that both **sun pointing** and **payload pointing** require low angular speed rates to be active. We also take into account the fact that both the EPS and ADCS have dedicated hardware, responsible for low-level requirements (e.g., control of actuators and sensors), and that the on-board software we are focusing on sends only high-level commands (e.g., switch to operational mode ‘X’). Assuming this system architecture, the on-board computer possesses all the necessary cross-system knowledge, and is thus expected to make the cross-system decisions.

Our on-board b-program software has a set of dedicated b-threads for each sub-system, implementing its control logic. This logic is based on data from actual satellite missions, such as [24, 95]. The program, as well as additional documentation, are available at [4].

6.2.1.1 The EPS

The EPS logic is specified by the b-thread presented in Listing 6.2. We initially wait for an EPS telemetry event, containing current telemetry received from the EPS board. In this example, we assume that this includes the battery voltage and the current EPS mode. According to the battery voltage field in the EPS telemetry, an initial EPS mode is requested using the **setEPSMode** event. After this initialization, each time an EPS telemetry is received (i.e., an **EPSTelemetry** event is triggered), this b-thread requests an EPS-mode change based on the battery’s current voltage.

The code in Listing 6.2 avoids stale requests that rely on old telemetries by using the break-upon idiom (see Section 5.4): whenever it requests an EPS-mode change, it also waits for EPS-telemetry events. If a new telemetry arrives before the set-mode request has been granted (that is, an **EPSTelemetry** event is selected by the b-program before the requested **setEPSMode** event has

been), the set-mode request is cancelled and the b-thread has a chance to re-evaluate it. The same technique was used by the `NotTooClose` b-thread in the tracker rover examples (Section 6.1).

6.2.1.2 The ADCS

The ADCS logic is specified by the b-thread presented in Listing 6.3. It is initialized by requesting a `SetADCSModeDetumbling` event. Each time an `ADCSTelemetry` event is selected, the appropriate `SetADCSMode` event, based on the current ADCS mode, the current angular rate, and on whether the satellite is performing an active pass or not. This b-thread, too, avoids stale set requests by waiting for telemetry events while requesting ADCS changes.

6.2.1.3 A Cross-System Requirement

Over and above the logic of each subsystem, an additional logic is required for handling behaviors involving a small number of subsystems. Consider for example the following cross-system requirement:

The ADCS should not remain in, or switch to, payload-pointing mode, while the EPS mode is low or critical.

Whether this requirement was provided originally, or after the first two had been written and tested, the BP paradigm encourages us to add a new b-thread for blocking this newly undesired behavior. The additional b-thread is given in Listing 6.4.

6.2.2 Formal Verification of B-Programs

In order to verify that the on-board satellite control software works, we use BPjs' analysis engine for verification. Consider, as an example, the requirement that the ADCS should not switch to payload-pointing mode when the EPS mode is low or critical. To achieve this objective, we use the `bp.ASSERT()` method for marking such possibility as *invalid* (see Subsection 4.4.1). The assertion is given in a new b-thread, presented in Listing 6.5.

Listing 6.2: EPS (Electric Power Supply) b-thread

```

1  var EPSTelem = bp.EventSet("EPSTelem", function(e){
2      return e instanceof EPSTelemetry;
3  });
4
5  var LOW_MAX = 70;
6  var GOOD_MIN = 60;
7  var CRITICAL_MAX = 50;
8  var LOW_MIN = 40;
9
10 bp.registerBThread("EPS - Turn ON/OFF logic", function () {
11
12     /* Init */
13     var ePSTelem = bp.sync({waitFor: EPSTelem});
14     if (ePSTelem.vBatt >= GOOD_MIN) {
15         bp.sync({waitFor: EPSTelem,
16             request: StaticEvents.SetEPSModeGood});
17     } else if (ePSTelem.vBatt >= LOW_MIN) {
18         bp.sync({waitFor: EPSTelem,
19             request: StaticEvents.SetEPSModeLow});
20     } else {
21         bp.sync({waitFor: EPSTelem,
22             request: StaticEvents.SetEPSModeCritical});
23     }
24     delete ePSTelem;
25
26     // ongoing control loop
27     while (true) {
28         var ePSTelem = bp.sync({waitFor: EPSTelem});
29         switch ( ePSTelem.mode ) {
30             case EPSTelemetry.EPSMode.Good:
31                 if (ePSTelem.vBatt < GOOD_MIN) {
32                     bp.sync({waitFor: EPSTelem,
33                         request: StaticEvents.SetEPSModeLow});
34                 }
35                 break;
36
37             case EPSTelemetry.EPSMode.Low:
38                 if (ePSTelem.vBatt > LOW_MAX) {
39                     bp.sync({waitFor: EPSTelem,
40                         request: StaticEvents.SetEPSModeGood});
41                 }
42                 if (ePSTelem.vBatt < LOW_MIN) {
43                     bp.sync({waitFor: EPSTelem,
44                         request: StaticEvents.SetEPSModeCritical});
45                 }
46                 break;
47
48             case EPSTelemetry.EPSMode.Critical:
49                 if (ePSTelem.vBatt > CRITICAL_MAX) {
50                     bp.sync({request: StaticEvents.SetEPSModeLow});
51                 }
52                 break;
53         }
54     }
55 });

```

Listing 6.3: ADCS b-thread.

```

1 var ADCSTelem = bp.EventSet("ADCSTelem", function (e) {
2   return e instanceof ADCSTelemetry;
3 });
4
5 bp.registerBThread("ADCS Mode Switch logic", function () {
6   // Init Deployment
7   bp.sync({request: StaticEvents.SetADCSModeDetumbling});
8
9   // ongoing
10  while (true) {
11    var aDCSEvent = bp.sync({waitFor: ADCSTelem});
12    switch ( aDCSEvent.mode ) {
13      case ADCSTelemetry.ADCSMode.Detumbling:
14        if (aDCSEvent.angularRate == "Low" && aDCSEvent.isActivePass) {
15          bp.sync({waitFor: ADCSTelem,
16                request: StaticEvents.SetADCSModePayloadPointing});
17        } else if (aDCSEvent.angularRate == "Low") {
18          bp.sync({waitFor: ADCSTelem,
19                request: StaticEvents.SetADCSModeSunPointing});
20        }
21        break;
22      case ADCSTelemetry.ADCSMode.SunPointing:
23        if (aDCSEvent.angularRate == "Low" && aDCSEvent.isActivePass) {
24          bp.sync({waitFor: ADCSTelem,
25                request: StaticEvents.SetADCSModePayloadPointing});
26        } else if (aDCSEvent.angularRate == "High") {
27          bp.sync({waitFor: ADCSTelem,
28                request: StaticEvents.SetADCSModeDetumbling});
29        }
30        break;
31      case ADCSTelemetry.ADCSMode.PayloadPointing:
32        if (aDCSEvent.angularRate == "Low" && !aDCSEvent.isActivePass) {
33          bp.sync({waitFor: ADCSTelem,
34                request: StaticEvents.SetADCSModeSunPointing});
35        } else if (aDCSEvent.angularRate == "High") {
36          bp.sync({waitFor: ADCSTelem,
37                request: StaticEvents.SetADCSModeDetumbling});
38        }
39        break;
40    }
41  }
42 });

```

Listing 6.4: Integrator b-thread.

```

1 bp.registerBThread("EPS & ADCS Integrator", function () {
2   while (true) {
3     var ePSTelem2 = bp.sync({waitFor: EPSTelem});
4     while ( ePSTelem2.currentEPSMode == "Low" ||
5            ePSTelem2.currentEPSMode == "Critical" ) {
6       if ( ePSTelem2.isActivePass ) {
7         bp.sync({waitFor: ADCSTelem,
8                request: StaticEvents.PassDone,
9                block: StaticEvents.SetADCSModePayloadPointing
10              });
11       }
12       var adcSEvent2 = bp.sync({waitFor: ADCSTelem,
13                              block: bp.Event("SetADCSModePayloadPointing")});
14       if (adcSEvent2.currentADCSMode == "PayloadPointing") {
15         bp.sync({waitFor: ADCSTelem,
16                request: StaticEvents.SetADCSModeSunPointing,
17                block: StaticEvents.SetADCSModePayloadPointing
18              });
19       }
20       var ePSTelem2 = bp.sync({waitFor: EPSTelem,
21                              block: StaticEvents.SetADCSModePayloadPointing
22              });
23     }
24   }
25 });

```

Listing 6.5: A safety requirement b-thread, generating a false assertion when a request to switch to the payload-pointing mode is made, while the EPS mode is low or critical.

```

1 bp.registerBThread("NeverPointingOnLow", function () {
2   var relevantEvents = [EPSTelem, USE_PAYLOAD];
3   /* Init */
4   var canPoint;
5   var evt = bp.sync({waitFor: relevantEvents});
6   if (EPSTelem.contains(evt)) {
7     canPoint = evt.mode.equals(EPSTelemetry.EPSMode.Good);
8   }
9   /* ongoing verification */
10  while (true) {
11    var pointingRequested = false;
12    var evt = bp.sync({waitFor: relevantEvents});
13    if (EPSTelem.contains(evt)) {
14      canPoint = evt.mode.equals(EPSTelemetry.EPSMode.Good);
15    } else if (USE_PAYLOAD.equals(evt)) {
16      pointingRequested = true;
17    }
18    bp.ASSERT(!(pointingRequested && !canPoint));
19  }
20 });

```

During verification, BPjs traverses a b-program’s state-space and searches for various violations, including invalid program states (marked by false assertions). If such a state is found, BPjs reports the sequence of program states and events that led to the discovery of the invalid state.

This type of verification has some advantages over testing. First, it rigorously covers concurrent programs, as it looks at all possible event selection orderings. Tests, by comparison, only look at a single event selection order. This is a major problem when testing b-programs, because they are concurrent by design, and b-programmers strive to limit event selection order as little as possible. Moreover, when a bug happens only when events are selected in a specific order, a test may or may not detect it — depending on the event selection order randomly selected during a specific test run. This may lead developers to believe that a bug has been fixed when, in reality, a different event selection order was used.

Another advantage of verification over testing is that verification returns program traces, which contain data about how the program ended in an invalid state. Tests, on the other hand, normally return stack traces, which only contain data about the program at the point when the violation was detected. Consider, for example, the common error of setting an object reference to `null` (e.g. because a sub-procedure failed to return a valid object reference), and then attempting to invoke methods through this reference. A unit test may detect this issue, but will complain at the point of the invocation, i.e., where the problem was discovered. The stack trace will not contain data about the reference assignment, where the problem actually occurred.

The size of a B-program’s state space depends on the number of b-threads it is composed of, and on the range of values they store. B-program state spaces can be very large, and may even be infinite. An example of a b-program with an infinite state space is the tracking rover verification b-program (see Section 6.1), where one of the environment b-threads contains the unbounded coordinates of the leading rover. This state explosion problem can be alleviated by reducing the model size, e.g. by storing only required data at the right abstraction level, by breaking one model into a number of sub-models, or by

blocking some events during verification on the basis of reasonable assumptions. It is also possible to limit the length of the traversed path, an essential feature for verifying infinite graphs.

Another option is to use modern multi-core/many-core servers and long verification times to traverse large portions of the program state graph. While this type of program analysis does not cover the entire program state-space, it does scan a large amount of program states (depending on time and resources allotted), and thus can provide some statistical guarantees of program correctness.

The b-program analysis approach presented here differs from “exhaustive testing” in two major aspects. From a semantic perspective, it traverses the system’s state-space rather than changes the system’s inputs. Thus, in addition to detecting safety violations, it can detect liveness violations, caused by “bad” cycles that would result in infinite runs that violate the system’s specifications. Moreover, the analysis engine can be used to perform other state-space analyses, such as drawing graphs and generating automata. From a performance perspective, the analysis process re-uses states by back-tracking the system’s state space. This makes the analysis process more efficient than performing multiple runs, and allows the engine to detect more than a single violation per run.

6.2.3 Simulating Environment For Verification

As previously noted, a b-program is verified by looking at all of its possible runs. The verification is done on the b-program only, excluding actual system interactions with the environment. Therefore, events emanating from the system’s environment (telemetry events, for example) have to be simulated during the verification process.

To this end, we create b-threads that act on behalf of the environment. Conveniently, using BP to simulate the environment gives us greater control over the events that can be requested by the environment, and how it responds to model decisions.

Requirement b-threads, such as the one presented in Listing 6.5, generate false assertions when a b-program gets into an illegal state, and do not affect normal program execution. Thus, they have to be a part of the verified program, but may be omitted from the deployed b-program. When full verification is possible, it is acceptable to omit them, as the verification proved that the conditions they assert will never be **false**. At the semantic level, successful full verification proves the requirements that these b-threads represent are never violated. In cases where full verification is not possible (e.g., due to the lack of resources/time), and if the system can be allowed to drop into a “safe mode”, it is possible to include asserting b-threads in the deployed version.

We start with a b-thread that randomly generates all of the possible environment events (shown in Listing 6.6). This b-thread requests **EPSTelemetry** and **ADCSTelemetry** events in order; however, each event is arbitrarily chosen from all possible events of each type. For example, an **EPSTelemetry** event can be any of the 606 possible combinations of active pass (yes/no), battery level (0 to 100 inclusive), and EPS mode (good/low/critical).

This type of environment is very easy to write, and does a good job of finding problems in early versions of the model. Moreover, its execution graph contains all possible environment behaviors. Therefore, if a b-program passes verification in this environment, it can survive any environment. However, such environments generate huge execution graphs, with a large number of outgoing edges from nodes representing the synchronization points where the controller waits for environment events. Thus, full verification using fully non-deterministic environments is often unrealistic. Additionally, this type of environment simulation may generate many false positives, because the real environment is often more constrained.

We used these b-threads to verify our system. First, as a baseline, we removed the cross-system b-thread (Listing 6.4), and successfully verified that the system indeed violates the cross-system requirement. Next, we tried to verify the system including the cross-system requirement. The verification process scanned approximately 650K possible states, using 1.3M iterations, without finding any violations. Since we chose the events randomly, it cannot

Listing 6.6: A b-thread that randomly generates all possible EPSTelemetry and ADCSTelemetry events.

```

1  var ACTIVE_PASSES = [false,true];
2  var EPS_MODES = EPSTelemetry.EPSMode.values();
3  var possibleEPSes = [];
4
5  for ( var vBatt=0; vBatt <= 100; vBatt++ ) {
6    for ( var modeIdx in EPS_MODES ) {
7      for ( var activePassIdx in ACTIVE_PASSES ) {
8        possibleEPSes.push( EPSTelemetry(vBatt, EPS_MODES[modeIdx],
9                                ACTIVE_PASSES[activePassIdx]) );
10   }}}
11
12 // Generate all possible ADCS Telemetries
13 var ADCS_MODES = ADCSTelemetry.ADCSMode.values();
14 var ANGULAR_RATES = ADCSTelemetry.AngularRate.values();
15 var possibleADCses = [];
16 for ( var adcsModeIdx in ADCS_MODES ) {
17   for ( var angularRateIdx in ANGULAR_RATES ) {
18     for ( var activePassIdx in ACTIVE_PASSES ) {
19       possibleADCses.push( ADCSTelemetry(ADCS_MODES[adcsModeIdx],
20                                           ANGULAR_RATES[angularRateIdx],
21                                           ACTIVE_PASSES[activePassIdx]) );
22   }}}
23
24 bp.registerBThread("Environment", function(){
25   while ( true ) {
26     bp.sync({request:possibleEPSes}); //Choose a random EPSTelemery event
27     bp.sync({request:possibleADCses}); //Choose a random ADCSTelemery event
28   });

```


Listing 6.7: A trace of the events that led to a violation of the system that does not include the the cross-system b-thread.

```

1 [BEvent name:SetADCSModeDetumbling]
2 [BEvent name:EnvSetAngularRateLow]
3 [BEvent name:PassDone]
4 [EPSTelemetry vBatt:1 currentEPSMode:Good activePass:false]
5 [BEvent name:SetEPSModeCritical]
6 [ADCSTelemetry currentADCSMode:Detumbling angularRate:Low activePass:false]
7 [BEvent name:SetADCSModeSunPointing]
8 [EPSTelemetry vBatt:2 currentEPSMode:Critical activePass:false]
9 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
10 [EPSTelemetry vBatt:3 currentEPSMode:Critical activePass:false]
11 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
12 [EPSTelemetry vBatt:4 currentEPSMode:Critical activePass:false]
13 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
14 [EPSTelemetry vBatt:5 currentEPSMode:Critical activePass:false]
15 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
16 [EPSTelemetry vBatt:6 currentEPSMode:Critical activePass:false]
17 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
18 [EPSTelemetry vBatt:7 currentEPSMode:Critical activePass:false]
19 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
20 [EPSTelemetry vBatt:8 currentEPSMode:Critical activePass:false]
21 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
22 [EPSTelemetry vBatt:9 currentEPSMode:Critical activePass:false]
23 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
24 [EPSTelemetry vBatt:10 currentEPSMode:Critical activePass:false]
25 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:false]
26 [BEvent name:ActivePass]
27 [EPSTelemetry vBatt:11 currentEPSMode:Critical activePass:true]
28 [ADCSTelemetry currentADCSMode:SunPointing angularRate:Low activePass:true]
29 [BEvent name:SetADCSModePayloadPointing]

```

be promised that we covered all possible runs.

A more realistic environment simulation generates the telemetry events according to previous selected events. For example, whenever a `SetEPSModeGood` event is selected, the following `EPSTelemetry` event will represent this mode switch. The complete code for this simulation can be found in [4]. We verified the system using this environment, again in two steps — initially without the cross-system b-thread, and then with it.

As before, the first run resulted with a trace of the events that led to a violation, presented in Listing 6.7. For all possible runs of up to 200 consecutive selected events, the second approach was able to verify that there were no violations — and in less than 2.5 minutes. Runs of a greater depth (i.e., longer) can also be verified using this approach.

6.2.4 Hybrid Lab Testing Environment

In this subsection we describe our on-going work on a novel hybrid laboratory for the advanced simulation and testing of satellite-mission software, where scenario-based programming is used to automatically generate complex test scenarios. This lab is already being used for integrating and testing a full-scale satellite on-board software suite by our group at Ben-Gurion University.

The hybrid lab consists of a simulation computer (desktop PC), and an *on-board computer* (OBC), which runs the on-board software. The simulation computer runs MATLAB, STK, and custom software for simulating satellite sensors (e.g., temperature, sun, magnetometers), actuators (e.g., magnetotors, wheels, heaters), and all the other satellite subsystems that the OBC interacts with (e.g., EPS, communications, payload). MATLAB is used for evaluating and running the satellite’s dynamics model. STK is used for evaluating and running the environment model (e.g., gravity and magnetic fields, and earth coverage).

The hybrid lab’s layout is depicted in Figure [6.3](#). The simulation computer interacts with the OBC through the native electrical on-board connectors used by the real hardware that the PC simulates. Thus, from the OBC’s perspective, the hybrid lab setup is identical to the setup used in orbit.

The presented hybrid lab allows developers to test on-board software in various scenarios. Examples of such scenarios are hardware failures, specific orbits, and environmental conditions written as a scenario-based test software. Additionally, the lab’s layout makes it possible to gradually replace simulated devices with their actual counterparts. In turn, this facility for gradual replacement allows for testing the integration of hardware components with the simulator, the OBC, and the on-board software.

We note that this hybrid laboratory design allows for testing on-board satellite software not necessarily written using scenario-based programming.

While all programs may (and often, do) contain bugs, the use of BP and high-level languages such as Java and JavaScript introduces new capabilities to on-board mission software debugging. Using our hybrid lab and Java debugging and profiling tools, we are able to test non-BP sections of the satellite

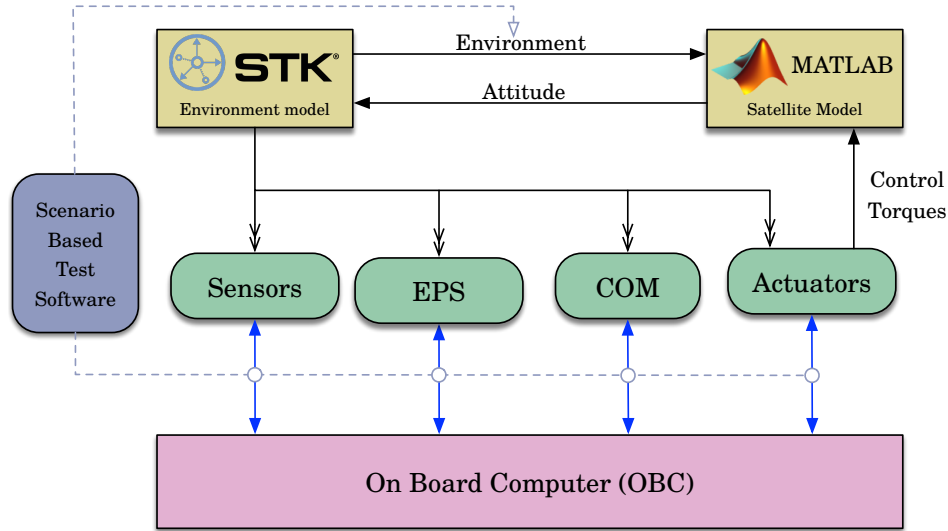


Figure 6.3: The hybrid lab layout. MATLAB and STK, running on a simulation computer, simulate the satellite’s dynamics, environment, and various hardware components. The OBC and the simulation computer interact using the native electrical on-board connectors used by the real hardware that the PC simulates.

software developed. These include: the software components responsible for listening to events and sending commands to the actual hardware; components that translate sensor data and communications into external events pushed to the b-program; and the BPjs runtime system itself. As an example, with this lab set-up, we were able to efficiently detect and fix a memory leak bug within the BPjs tool. Detecting and fixing similar bugs in a C program, as usually used in satellite software, may prove to be a difficult task. Moreover, the fixed software will need to go through the (expensive) testing process again.

6.2.5 Conclusion

This section present three contributions to the field of on-board satellite software programming and testing:

Software. A novel approach for programming satellites software, using BP.

Verification. A tool and methodology for verifying the correctness of the

proposed on-board satellite software, which operates by looking at all of its possible runs, ensuring that they all comply with a set of formal requirements. This verification can be performed on specific parts of a model, on specific logical layers of an application, or on an entire model at a specific abstraction level.

Testing. A novel hybrid laboratory for advanced simulation and testing of satellite-mission software, where scenario-based programming is used to automatically generate complex test scenarios.

While the provided examples have been simplified for the purpose of this paper, we are confident that the approaches outlined in this chapter can be applied to full-scale on-board satellite software. Our group at Ben-Gurion University is currently working on such a case study, which we hope will validate the scalability of our proposed approach.

6.3 Related Work

The main property of SBM distinguishing it from standard programming, executable modeling in, say, fUML [86] and Alf [87], rule-based or publish-subscribe systems [28], or highly intuitive visual languages like Statecharts or UML-RT [92], is its compositional semantics. Specifically, in SBM, independently specified threads of behavior running in parallel can be composed *automatically* by the infrastructure, with support for forbidden behavior, strict event ordering, or satisfying multiple concurrent requests with a single event triggering — to mention a few examples. This also enables alignment of the software structure with its requirements.

This advantage also holds in comparisons with systems geared towards parallel synchronized executions and optional voting, like SystemC [89], Esterel [11], or multi-agent systems. SBM can be distinguished from the separation of concerns in Aspect Oriented Programming (AOP) [66] and similar techniques in that there is no distinction between base and modifications or exceptions thereto, and that the anchors of specifications are usually meaningful system events rather than particular pre-existing method names. Scenarios

also manage states conveniently; state management in AOP join points requires non-trivial programming. The BIP (behavior, interaction, priority) [13] language has similar goals and terminologies; however, it focuses on the development of a system that is correct-by-construction, while SBM concentrates on programming in a natural way, and turns to techniques like model-checking to discover design issues.

Another approach for combining scenarios for test generation, based on synthesis and LSC, is proposed in [74]. Under that proposed approach, tests are strategies for generating environment events. Tests are created by detecting cases where an environment can cause different activated LSCs to refer to common objects. This approach allows for a minimal set of tests that allow maximal coverage of cases where multiple scenarios interact.

One common concern in system testing and analysis is external side effects. In the current case study, these include the effect of turning on a satellite heating unit, or rotating a rover and activating its wheel engines at 50% effort. Other domains may include writes to databases and file systems, or sending and receiving data through a network. The solutions presented here use environment simulation b-threads to incorporate the side effects in the verified model. Other systems, such as NASA’s JavaPathFinder [57], use a similar approach, by defining “native peer” classes that model these side effects.

We proposed using lengthy analysis runs in order to cover significant portions of program state-space, when that space is too large to fully traverse and verify. Microsoft applies a similar approach in its SAGE system [14]. SAGE uses fuzz testing [12] to test products. But unlike regular fuzz testing, which generates inputs at random, SAGE analyses the verified binaries for execution paths. It then generates the fuzz inputs on the basis of this analysis, in order to maximize executable coverage.

6.4 Conclusion

In this chapter, we presented two examples of systems designed using the Model-Driven Engineering approach. Both used BP for creating their model.

By using BPjs as the BP platform, we were able to easily embed the models in Java host applications. Additionally, we were able to directly verify these models, including environment simulation where needed.

Model-Driven Engineering is a promising approach for the creation of complex software. Using BP and BPjs for the model layer allows for the design of highly reliable models, and consequently reliable software and hybrid systems.

Chapter 7

Conclusions and Additional Proposed Research

This chapter discusses general conclusions and possible directions for future research.

In this dissertation we assessed the potential uses of Behavioral Programming across a range of engineering and research contexts. We started by defining BP in a modular, parameterized way. This led us to define *synchronization statements* and modular *event selection strategies*, that can be used both during runtime and during program analysis. It also enabled us to view b-program execution as a traversal of a state-space graph, where the nodes are the running b-program at a given synchronization point, and the edges are events that, at said point, have been requested and not blocked. While earlier research has touched on some of these topics, ours is the first BP definition to include all these aspects, and to support them with pluggable, modular runtime and analysis software framework — BPjs.

The conceptual work done while developing our notion of a running b-program is reflected in code. As with most BP runtime implementations, BPjs initially had a `BThread` class, the runtime representation of a b-thread. As we progressed, we realized that while b-threads are a useful concept for the BP programmer, this is less the case for the BP runtime, which is only interested at the state of the b-thread *during synchronization points*. Thus, we

removed the `BThread` class, and replaced it with `BThreadSyncSnapshot`, which captures the state of a b-thread immediately after it submits a synchronization statement.

Based on this definition of BP, we created a bi-directional communication protocol that can be used to embed a running b-program in a traditional software system. This makes model-driven engineering using BP easier, as it allows system designers to leave low-level tasks to the host application, and to focus their BP model on high-level decision making. This separation is important for two reasons. First, it facilitates the re-use of models, because they now no longer need to take technical, platform-specific details into account. Second, it makes models smaller, and thus easier to analyze and verify.

The proof of a pudding is in the eating; thus, it would be interesting to see whether BPjs, and our definition of BP, gain traction as more people give it a byte. So far, BPjs have been used in a number of university classes and student projects (in addition to the usages described in Chapter 6). It is also used as the main BP research platform for our BP research group at Ben-Gurion University of the Negev — three graduate students using it in their research projects.

7.1 Possible Future Research Directions

Future extensions of the work presented in this dissertation can be directed in two possible directions: inward and outward.

7.1.1 Inward

Inward work will involve extensions and alterations to our definition of BP. One such direction is the implementation of new event selection strategies. Possible examples of this approach include strategies that impose fairness by limiting starvation; strategies optimized towards advancing the maximal number of b-threads; and strategies that harness machine learning or planning to optimize some aspect of program execution. Meta-strategies, that combine multiple

strategies and adapt to current b-program’s needs, are another interesting path for future research. Once enough strategies are available, we will be able to compare and contrast them. More importantly, we will be able to curate them for re-use, tailoring a strategy to a given b-program based on its properties and the resources available for its execution.

Resource allocation is another interesting topic. Specifically, as BPjs have broken the connection between an OS thread and a b-thread, we can now try different strategies for matching OS threads to b-threads. Should we use thread pools of fixed size, or create and destroy threads during program execution? Can we predict the optimal thread pool size for a b-program? How is OS thread handling affected by changes to the size or number of CPUs, memory, or average amount of b-threads advancing between synchronization points? Robust answers to these questions will allow b-programs to run faster and more efficiently, thereby enabling the use of larger, more complex models.

B-program verification offers many research opportunities and open questions. We currently use depth-first search; whilst this does work, there is clearly room for improvement. As an example: we could incorporate heuristics for selecting the events that the verifier should explore first on entering a new node. This could speed up the process of bug discovery.

DFS is inherently single-threaded: at each point, only a single node in the state-space is explored. Parallelizing the verification process — perhaps using a breadth-first search with heuristics — may reduce verification time. A natural extension of this would be the distribution of the b-program analysis process across a large number of machines.

One specific direction in which parallelism and distribution can be taken is with the integration of graph-based databases into the process. In such cases, a large number of machines will traverse a b-program’s state-space, adding the nodes and edges discovered to a central graph database. The b-program could then be analyzed by querying the database.

A future direction we plan to explore is that of adding fairness support to the verifiers presented here (see [3, C3.5]). In some cases, liveness violation analysis can return a valid, but nevertheless very unlikely counter example.

These counter examples can be ignored safely; a fairness parameter offers an accurate way of filtering them out.

7.1.2 Outward

Outward work will use BP as a foundation for building other systems. One interesting example would be extending the work described in Chapter 5 to define additional languages. A further extension could be in integrating language definitions, thereby creating a heterogeneous modeling environment whose underlying semantic layer is defined and implemented using BP. Model heterogeneity offers a compelling alternative to model generality [72]. The Ptolemy Project [27] explores heterogeneous modeling for concurrent, real-time embedded systems using an actor-based approach; it will be interesting to compare and contrast it with BP-based heterogeneous modeling.

Another interesting research direction is the use of BP in model-driven engineering (MDE), for creating systems in various domains. This would allow the community to determine the domains best served by BP. A comparative analysis of such projects will give useful pointers regarding what makes a domain a good fit for BP. It is always interesting to see a theory meeting the real world — will the problems that will be faced be caused by the paradigm, or by technical limitations such as b-thread count and performance? Will we identify reoccurring design patterns in these systems' BP code? Will we be able to re-use b-threads across domains?

Finally, a task that is not purely academic: we would like to see BPjs gain non-academic traction, through adoption by general software engineering practitioners. This is a task that other BP implementations — and model driven systems in general — are yet to achieve¹. MDE tools are traditionally cited as a barrier for MDE adoption (see [101] for research overview). However, social and organizational barriers are often as important as the technical ones [19, 101]. Thus, adopting MDE is not just a technical decision, as it would involve significant business considerations.

¹This may be a bit of “my thesis will have some effect on the world” text; I hope the reader will forgive this self-indulgence, in the final few paragraphs of a dissertation.

Whittle et al. [101] conclude that tool makers should “match tools to people, not the other way around.” They offer a taxonomy of the tool-related hurdles that prevent practitioners from adopting MDE. These include versioning, chaining, flexibility, tool and language complexity, usability, conformance with human abstractions, theoretical foundations and formal semantics, training, quality of generated code, upfront investment (migration, tool integration), and organizational inertia.

The technological foundation we chose for building BPjs allows it to fit into the modern programming eco-system. Rather than using a unique workbench, BPjs is packaged as a library, its code arranged using the popular, IDE-agnostic Apache Maven system. This allows programmers to use their preferred tools, thus addressing usability (since the programmer can use her regular editor), tool integration, and training. It also reduces the inertial impact of adopting BPjs. Creating models in JavaScript allows engineers to use their existing knowledge for model creation. It is also a better fit for code versioning and review tools than node-based formats such as XML. The design choices enabled through the use of familiar design patterns such as Strategy and Observer flatten the learning curve, address issues such as chaining, tool complexity, and flexibility, and solves code-creation issues by design. BP itself, as we have seen, conforms with human abstractions, is well-founded theoretically, and provides intuitive formal semantics.

All these make BPjs the first “blue-collar BP platform”. The programming proletarians have nothing to lose but their chains.

Bibliography

- [1] Runtime Verification Conference Website. <http://www.runtime-verification.org/> (2001-2018). [Online; accessed 31-Jan-2018]
- [2] Aljamaan, H., Garzon, M., Lethbridge, T.: Umplerun: a dynamic analysis tool for textually modeled state machines using umple (2015)
- [3] Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
- [4] Bar-Sinai, M., Elyasaf, A., Sadon, A., Weiss, G.: Appendices for A Scenario Based On-board Software and Testing Environment for Satellites. IACAS 59 (2019). URL <https://github.com/bThink-BGU/59IACAS-Appendix>
- [5] Bar-Sinai, M., Elyasaf, A., Sadon, A., Weiss, G.: A scenario based on-board software and testing environment for satellites. In: Proceedings of The 59th Israel Annual Conference on Aerospace Sciences (2019)
- [6] Bar-Sinai, M., Weiss, G.: Code Appendix for “BPjs - A Behavioral Programming Tool Suite” (2018). URL https://github.com/michbarsinai/BPjs-SCP-OSP_CodeAppendix
- [7] Bar-Sinai, M., Weiss, G.: Code appendix for “verification with BPjs” (2019). URL <https://github.com/michbarsinai/FormaliSE2020CodeAppendix>

- [8] Bar-Sinai, M., Weiss, G., Marron, A.: Code Appendix for “of Diagrammatic Languages Using Behavioral Programming and Queries” (2016). URL <https://github.com/michbarsinai/BP-javascript-search>
- [9] Bar-Sinai, M., Weiss, G., Marron, A.: Defining semantic variations of diagrammatic languages using behavioral programming and queries. In: Proceedings of the 2nd International Workshop on Executable Modeling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 3, 2016., pp. 5–11 (2016). URL <http://ceur-ws.org/Vol-1760/paper1.pdf>
- [10] Bar-Sinai, M., Weiss, G., Shmuel, R.: BPjs: an extensible, open infrastructure for behavioral programming research. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018, pp. 59–60 (2018). DOI 10.1145/3270112.3270126. URL <https://doi.org/10.1145/3270112.3270126>
- [11] Berry, G.: The foundations of Esterel. In: Proof, Language, and Interaction, pp. 425–454 (2000)
- [12] Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. IBM systems journal **22**(3), 229–245 (1983)
- [13] Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. CONCUR (2008)
- [14] Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: Whitebox fuzz testing in production. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 122–131. IEEE Press (2013)
- [15] BPjs contributors: BPjs Javadoc Online (2019). URL <http://www.javadoc.io/doc/com.github.bthink-bgu/BPjs/>

- [16] Brooks Jr., F.P.: No Silver Bullet — Essence and Accidents of Software Engineering. Information Processing (1986). DOI 10.1109/MC.1987.1663532. URL <https://doi.org/10.1109/MC.1987.1663532>
- [17] Brooks Jr., F.P.: The Mythical Man-month (Anniversary Ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
- [18] Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 1020 states and beyond. Information and Computation **98**(2), 142 – 170 (1992). DOI [https://doi.org/10.1016/0890-5401\(92\)90017-A](https://doi.org/10.1016/0890-5401(92)90017-A). URL <http://www.sciencedirect.com/science/article/pii/089054019290017A>
- [19] Chami, M., Bruel, J.M.: A survey on MBSE adoption challenges. In: The Systems Engineering Conference of the Europe, Middle-East and Africa (EMEA) Sector of INCOSE (EMEASEC 2018) (2018)
- [20] Chan, J., Yang, W.: Ambiguities in Java. CTHPC **2**, 51–62 (2002)
- [21] Choirat, C., Honaker, J., Imai, K., King, G., Lau, O.: Zelig: Everyone’s Statistical Software (2019). URL <http://zeligproject.org/>
- [22] Cohen, B., Maoz, S.: Semantically configurable analysis of scenario-based specifications. In: Gnesi, S., Rensink, A. (eds.) Fundamental Approaches to Software Engineering, pp. 185–199. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
- [23] Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Journal on Formal Methods in System Design **19**(1), 45–80 (2001)
- [24] Deng, S., Meng, T., Wang, H., Du, C., Jin, Z.: Flexible attitude control design and on-orbit performance of the ZDPS-2 satellite. Acta Astronautica **130**, 147–161 (2017)
- [25] Edward, Y.: Modern structured analysis. Edward Yourdon Englewood: Prentice-Hall International **4** (1989)

- [26] Eitan, N., Harel, D.: Adaptive behavioral programming. IEEE International Conference on Tools with Artificial Intelligence (2011)
- [27] Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y., Neuendorffer, S.: Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE* **91**(1), 127–144 (2003). URL <http://chess.eecs.berkeley.edu/pubs/488.html>
- [28] Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)* **35**(2), 114–131 (2003)
- [29] Fecher, H., Schönborn, J., Kyas, M., de Roeper, W.: 29 new unclarities in the semantics of UML 2.0 state machines. *Formal Methods and Software Engineering* pp. 52–65 (2005)
- [30] Floyd, R.W.: Assigning meanings to programs (1967). DOI <https://doi.org/10.1090/psapm/019>
- [31] Fox Keller, E., Harel, D.: Beyond the gene. *PLoS ONE* **2**(11), 1–11 (2007). DOI 10.1371/journal.pone.0001231. URL <http://dx.plos.org/10.1371%2Fjournal.pone.0001231>
- [32] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education (1994)
- [33] Gordon, M., Marron, A., Meerbaum-Salant, O.: Spaghetti for the main course?: Observations on the naturalness of scenario-based programming. In: *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '12*, pp. 198–203. ACM, New York, NY, USA (2012). DOI 10.1145/2325296.2325346. URL <http://doi.acm.org/10.1145/2325296.2325346>
- [34] Gosling, J.: The feel of Java. *Computer* **30**(6), 53–57 (1997). DOI 10.1109/2.587548. URL <https://doi.org/10.1109/2.587548>

- [35] Greenyer, J., Bar-Sinai, M., Weiss, G., Sadon, A., Marron, A.: Modeling and programming a leader-follower challenge problem with scenario-based tools. In: Hebig, R., Berger, T. (eds.) Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018., *CEUR Workshop Proceedings*, vol. 2245, pp. 376–385. CEUR-WS.org (2018). URL http://ceur-ws.org/Vol-2245/mdetools_paper_8.pdf
- [36] Greenyer, J., Gritzner, D., Gutjahr, T., König, F., Glade, N., Marron, A., Katz, G.: ScenarioTools—a tool suite for the scenario-based modeling and analysis of reactive systems. *Science of Computer Programming* **149**, 15–27 (2017)
- [37] Grun, C.: Pushing XML main memory databases to their limits (2006)
- [38] Harel, D.: Statecharts: A visual formalism for complex systems. *Science of Computer Programming* **8**(3), 231–274 (1987)
- [39] Harel, D.: Biting the silver bullet: toward a brighter future for system development. *Computer* **25**(1), 8–20 (1992). DOI 10.1109/2.108047
- [40] Harel, D., Kantor, A., Katz, G.: Relaxing synchronization constraints in behavioral programs. In: *Logic for Programming Artificial Intelligence and Reasoning*, pp. 355–372 (2013). URL http://link.springer.com/chapter/10.1007/978-3-642-45221-5_25
- [41] Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., Weiss, G.: On composing and proving correctness of reactive behavior. *EM-SOFT* (2013). URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6658591

- [42] Harel, D., Katz, G.: Scaling-up behavioral programming: Steps from basic principles to application architectures. In: Proceedings of the 4th International Workshop on Programming based on Actors Agents and Decentralized Control, pp. 95–108. ACM (2014)
- [43] Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of reactive programs. pp. 3–12 (2012). DOI 10.1109/ICECCS.2012.25
- [44] Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of safety and liveness violations in reactive programs. Transactions on Computational Collective Intelligence XVI pp. 1–33 (2014). URL http://link.springer.com/chapter/10.1007/978-3-662-44871-7_1
- [45] Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-Out of behavioral requirements. In: International Conference on Formal Methods in Computer-Aided Design, vol. 2, pp. 378–398. Springer (2002)
- [46] Harel, D., Lampert, R., Marron, A., Weiss, G.: Model-checking behavioral programs. In: Proceedings of the Ninth ACM International Conference on Embedded Software, EMSOFT '11, pp. 279–288. ACM, New York, NY, USA (2011). DOI 10.1145/2038642.2038686. URL <http://doi.acm.org/10.1145/2038642.2038686>
- [47] Harel, D., Maoz, S., Szekely, S., Barkan, D.: PlayGo: towards a comprehensive tool for scenario based programming. In: ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010, pp. 359–360 (2010). DOI 10.1145/1858996.1859075. URL <https://doi.org/10.1145/1858996.1859075>
- [48] Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
- [49] Harel, D., Marron, A., Nissim, A., Weiss, G.: A software engineering framework for switched fuzzy systems. In: FUZZ-IEEE 2012, IEEE In-

- ternational Conference on Fuzzy Systems, Brisbane, Australia, June 10-15, 2012, Proceedings., pp. 1–9 (2012). DOI 10.1109/FUZZ-IEEE.2012.6251301. URL <https://doi.org/10.1109/FUZZ-IEEE.2012.6251301>
- [50] Harel, D., Marron, A., Weiss, G.: Programming coordinated scenarios in Java. In: Proc. 24th European Conference on Object-Oriented Programming (ECOOP), *Lecture Notes on Computer Science*, vol. 6183, pp. 250–274 (2010). URL http://dx.doi.org/10.1007/978-3-642-14107-2_12
- [51] Harel, D., Marron, A., Weiss, G.: Behavioral programming. *Communications of the ACM* **55**(7) (2012)
- [52] Harel, D., Pnueli, A.: Logics and models of concurrent systems. chap. On the Development of Reactive Systems, pp. 477–498. Springer-Verlag, Berlin, Heidelberg (1985). URL <http://dl.acm.org/citation.cfm?id=101969.101990>
- [53] Harel, D., Rumpe, B.: Meaningful modeling: What’s the semantics of “semantics”? *IEEE Computer Magazine* (2004)
- [54] Harel, D., Segall, I.: Planned and traversable Play-Out: A flexible method for executing scenario-based programs. In: Grumberg, O., Huth, M. (eds.) TACAS, *Lecture Notes in Computer Science*, vol. 4424, pp. 485–499. Springer (2007). URL <http://dblp.uni-trier.de/db/conf/tacas/tacas2007.html#HarelS07>
- [55] Harel, D., Segall, I.: Synthesis from scenario-based specifications. *Journal of Computer and System Sciences* **78**(3), 970 – 980 (2012). DOI <https://doi.org/10.1016/j.jcss.2011.08.008>. URL <http://www.sciencedirect.com/science/article/pii/S0022000011000870>. In Commemoration of Amir Pnueli
- [56] Hause, M.: The SysML modelling language. In: Fifteenth European Systems Engineering Conference (2006)

- [57] Havelund, K., Pressburger, T.: Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer* **2**(4), 366–381 (2000). DOI 10.1007/s100090050043. URL <https://doi.org/10.1007/s100090050043>
- [58] Henzinger, T., Kirsch, C., Sanvido, M., Pree, W.: From control models to real-time code using Giotto. *Control Systems Magazine, IEEE* (2003)
- [59] Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969). DOI 10.1145/363235.363259. URL <http://doi.acm.org/10.1145/363235.363259>
- [60] Holzmann, G.J.: The model checker SPIN. *IEEE Transactions on Software Engineering* **23**(5), 279–295 (1997). DOI 10.1109/32.588521. URL <https://doi.org/10.1109/32.588521>
- [61] Imai, K., King, G., Lau, O.: Toward a common framework for statistical analysis and development. *Journal of Computational Graphics and Statistics* **17**(4), 892–913 (2008). URL <http://j.mp/msE15c>
- [62] Katz, G.: On module-based abstraction and repair of behavioral programs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) *Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 518–535. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-45221-5_35
- [63] Katz, G., Marron, A., Sadon, A., Weiss, G.: On-the-fly construction of composite events in scenario-based modeling using constraint solvers. In: *Proc. 7th Int. Conf. on Model-Driven Engineering and Software Development MODELSWARD, 2019* (2019)
- [64] Keller, R.M.: Formal verification of parallel programs. *Commun. ACM* **19**(7), 371–384 (1976). DOI 10.1145/360248.360251. URL <https://doi.org/10.1145/360248.360251>

- [65] Kernighan, B., Plauger, P.: Software Tools. Addison-Wesley Publishing Company (1976)
- [66] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Longtier, J., Irwin, J.: Aspect-oriented programming. Euro. Conf. on Object-Oriented Prog. (ECOOP) (1997)
- [67] Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flat, M., McCarthy, J., Rafkind, J., Tobin-Hochstadt, S., Findler, R.: Run your research, on the effectiveness of lightweight mechanization. pp. 285–296. POPL (2012)
- [68] Klose, J., Toben, T., Westphal, B., Wittke, H.: Check it out: On the efficient formal verification of live sequence charts. pp. 219–233 (2006). DOI 10.1007/11817963_22
- [69] Kugler, H., Plock, C., Roberts, A.: Synthesizing biological theories. In: CAV, pp. 579–584 (2011). URL http://link.springer.com/chapter/10.1007/978-3-642-22110-1_46
- [70] de Lara, J., Vangheluwe, H.: ATOM3: A tool for multi-formalism and meta-modelling. In: Kutsche, R., Weber, H. (eds.) FASE, *Lecture Notes in Computer Science*, vol. 2306, pp. 174–188. Springer (2002)
- [71] Latombe, F., Crégut, X., Deantoni, J., Pantel, M., Combemale, B.: Coping with semantic variation points in domain-specific modeling languages. In: 1st International Workshop on Executable Modeling (EXE’15) (2015)
- [72] Lee, E.A.: Heterogeneous actor models (2011). URL <http://chess.eecs.berkeley.edu/pubs/866.html>. Invited Roadmap Talk, EM-SOFT, Taipei, Taiwan
- [73] Lethbridge, T.C., Mussbacher, G., Forward, A., Badreddin, O.: Teaching uml using umple: Applying model-oriented programming in the

- classroom. In: 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET), pp. 421–428 (2011). DOI 10.1109/CSEET.2011.5876118
- [74] Manna, V.P.L., Segall, I., Greenyer, J.: Synthesizing tests for combinatorial coverage of modal scenario specifications. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015, pp. 126–135 (2015). DOI 10.1109/MODELS.2015.7338243. URL <https://doi.org/10.1109/MODELS.2015.7338243>
- [75] Maoz, S.: Polymorphic scenario-based specification models: semantics and applications. *Software & Systems Modeling* **11**(3), 327–345 (2012). DOI 10.1007/s10270-010-0168-6. URL <https://doi.org/10.1007/s10270-010-0168-6>
- [76] Maoz, S., Harel, D.: From multi-modal scenarios to code: Compiling LSCs into AspectJ. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pp. 219–230. ACM, New York, NY, USA (2006). DOI 10.1145/1181775.1181802. URL <http://doi.acm.org/10.1145/1181775.1181802>
- [77] Maoz, S., Harel, D., Kleinbort, A.: A compiler for multimodal scenarios: Transforming lscs into aspectj. *ACM Trans. Softw. Eng. Methodol.* **20**(4) (2011). DOI 10.1145/2000799.2000804. URL <https://doi.org/10.1145/2000799.2000804>
- [78] Maoz, S., Sa’ar, Y.: Assume-guarantee scenarios: Semantics and synthesis. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) *Model Driven Engineering Languages and Systems*, pp. 335–351. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
- [79] Maoz, S., Sa’ar, Y.: Counter Play-Out: Executing unrealizable scenario-based specifications. In: Proceedings of the 2013 International Confer-

- ence on Software Engineering, ICSE '13, pp. 242–251. IEEE Press, Piscataway, NJ, USA (2013). URL <http://dl.acm.org/citation.cfm?id=2486788.2486821>
- [80] Marron, A.: Behaviroal programming website (2019). URL <http://www.b-prog.org>
- [81] Marron, A., Hacothen, Y., Harel, D., Mülder, A., Terfloth, A.: Embedding scenario-based modeling in statecharts. In: Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018., pp. 443–452 (2018). URL http://ceur-ws.org/Vol-2245/morse_paper_2.pdf
- [82] Marron, A., Szekely, S.: LSC Language Reference Manual. Department of Computer Science and Applied Mathematics Weizmann Institute of Science (2014)
- [83] Marron, A., Weiss, G., Wiener, G.: A decentralized approach for programming interactive applications with JavaScript and Blockly. In: Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, AGERE! 2012, pp. 59–70. ACM, New York, NY, USA (2012). DOI 10.1145/2414639.2414648. URL <http://doi.acm.org/10.1145/2414639.2414648>
- [84] Mellor, S., Balcer, M., et al.: Executable UML: A foundation for model-driven architectures. Addison-Wesley Longman Publishing Co., Inc. (2002)
- [85] Mozilla, individual contributors: Mozilla rhino JavaScript engine web-

- p>site (2019). URL
- <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino>
- [86] OMG: Semantics Of A Foundational Subset For Executable UML Models (FUML) v1.2.1 (2016). URL <http://www.omg.org/spec/FUML/1.2.1/>
 - [87] OMG: Action Language for Foundational UML (Alf), V. 1.1 (2017). OMG document `formal/17-07-04`
 - [88] OMG: Unified Modeling Language Superstructure Specification, v2.0 (Aug. 2005). URL <http://www.omg.org>
 - [89] OSCI: Open SystemC Initiative. IEEE 1666 Lang. Ref. Manual. <http://www.systemc.org> (2011)
 - [90] Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57 (1977). DOI 10.1109/SFCS.1977.32
 - [91] Pnueli, A., Sa’ar, Y., Zuck, L.D.: Jtlv: A framework for developing verification algorithms. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification, pp. 171–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
 - [92] Posse, E., Dingel, J.: An executable formal semantics for UML-RT. *Software & Systems Modeling* **15**(1), 179–217 (2016). DOI 10.1007/s10270-014-0399-z. URL <https://doi.org/10.1007/s10270-014-0399-z>
 - [93] Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)
 - [94] R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2013). URL <http://www.R-project.org/>

- [95] Ran, D., Sheng, T., Cao, L., Chen, X., Zhao, Y.: Attitude control system design and on-orbit performance analysis of nano-satellite—“Tian Tuo 1”. *Chinese Journal of Aeronautics* **27**(3), 593–601 (2014)
- [96] Robie, J., Dyck, M., Spiegel, J.: XQuery 3.1: An XML query language (2015). URL <http://www.w3.org/TR/xquery-31/>
- [97] Sadon, A., Bar-Sinai, M., Weiss, G.: Code Appendix for “BPjsLeaderFollower” (2018). URL <https://github.com/bThink-BGU/BPjsLeaderFollower>
- [98] Seidewitz, E.: What models mean. *IEEE Software* **20**, 26–32 (2003)
- [99] Rodrigues da Silva, A.: Model-driven engineering. *Computer Languages, Systems and Structures* **43**(C), 139–155 (2015). DOI 10.1016/j.cl.2015.06.001. URL <http://dx.doi.org/10.1016/j.cl.2015.06.001>
- [100] Trower, J., Gray, J.: Blockly language creation and applications: Visual programming for media computation and bluetooth robotics control. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE '15*, pp. 5–5. ACM, New York, NY, USA (2015). DOI 10.1145/2676723.2691871. URL <http://doi.acm.org/10.1145/2676723.2691871>
- [101] Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial adoption of model-driven engineering: Are the tools really the problem? In: Moreira, A., Schatz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) *Model-Driven Engineering Languages and Systems*, pp. 1–17. Springer Berlin Heidelberg (2013). URL https://link.springer.com/chapter/10.1007/978-3-642-41533-3_1
- [102] Wiener, G., Weiss, G., Marron, A.: Coordinating and Visualizing Independent Behaviors in Erlang. In: *Proc. 9th ACM SIGPLAN Erlang Workshop* (2010)

תקציר עברי

נושא העבודה: הרחבת גישת התכנות ההתנהגותי עבור תכנון מבוסס מודלים

מנחה: ד"ר גרא וייס

תכנות התנהגותי (BP - Behavioral Programming) היא פרדיגמה תכנותית המאפשרת לתאר מערכות בעלות התנהגות מורכבת על ידי אוסף התנהגויות סדרתיות (single threaded) הנשזרות באופן אלגוריתמי בזמן ריצה. כל התנהגות כזו מכונה b-thread, והיא נכתבת על ידי קוד סדרתי רגיל. ה-b-threads הנשזרים יכולים לא רק להוסיף פעולות למערכת, אלא גם לתאם פעולות על ידי המתנה, ולהחסיר פעולות על ידי חסימות. שזירת ה-b-threads מתבססת על נקודות סנכרון, בהן ה-b-threads יכולים לבקש, לחסום, או להמתין לאירועים. תיאור זה מתאים במיוחד למערכות תגובתיות (reactive systems) - מערכות המגיבות לאירועים בסביבתן כחלק ממהלך הפעולה הרגיל שלהן. מערכות כאלו נפוצות מאוד, וכוללות שרתי אינטרנט, רכבים אוטונומיים, וממשקי משתמש. לתיאור מערכת על ידי אוסף b-threads מספר יתרונות חשובים. ראשית, תיאור זה אינטואיטיבי מאוד, כיוון שהוא דומה לתיאור דרישות המערכת כפי שהן ניתנות בשפה אנושית. שנית, תאור זה מאפשר לשנות ולעדן (refine) את התנהגות המערכת הכוללת על ידי הוספה של b-threads חדשים (כאמור, הוספה של b-thread לא בהכרח מוסיפה התנהגות חדשה - היא יכולה גם למנוע התנהגות לא רצויה). שלישית, ניתן יחסית בקלות להתייחס לתיאור זה כאל מודל פורמלי של מערכת, לנתח אותו באופן ממוחשב ואף לאמת את נכונותו בייחס למפרט דרישות פורמלי (formal verification).

תכנות התנהגותי, שהוצג לראשונה ב-2010 על ידי הראל, מרון, ו-וייס, הוא סוג של תכנות מבוסס תסריטים. שיטה נוספת של תכנות מבוסס תסריטים היא Live Sequence Charts (LSC), שהוצגה על ידי דאם והראל ב-2001. שפת LSC מאפשרת אף היא לבצע הוספות וחסימות, אולם מתמקדת בהודעות העוברות בין אובייקטים, במקום באירועים (שהם אובייקטים נטולי מקור או יעד מוגדרים). הבדל נוסף בין LSC ל-BP הוא ששפת LSC מבוססת על שרטוטים ולא על טקסט. לאחרונה הוצגו גרסאות מבוססות טקסט ל-LSC, אולם גם הן מתמקדות בהעברת הודעות בין אובייקטים.

מספר מערכות להרצה וניתוח של תוכניות BP הוצגו בעבר, כולל ספריות הרצה ל-Java, C++, Blockly, JavaScript ו-Erlang. שתי מערכות מתקדמות יחסית הן BPJ, שאיפשרה תכנות התנהגותי בשפת Java, ו-BPC, המאפשרת תכנות התנהגותי בשפת C++. שתי הספריות איפשרו ניתוח מוגבל של תוכניות BP. גרסה אחת של BPJ אפשרה בדיקת תכונות בטיחות, אולם אינה שימושית יותר מסיבות טכניות. BPC מאפשרת ניתוח על ידי תרגום של התוכנית למודל וניתוח על ידי כלי חיצוני. שתי ספריות אלו נבנו כפלטפורמה שעליה ניתן לבנות תוכנות BP, אולם באופן שונה מהעבודה המוצגת כאן: ספריית BPJ ניתנה להרחבה ועידון על ידי שינוי הקוד שלה, ואילו BPC מתוכננת כמסגרת תוכנית (framework) ולא כספריה הניתנת להטמעה במערכות תוכנה מסורתיות.

תחום מערכות מידול תוכנה הוא תחום נרחב מאוד, וכולל כמובן גם פרדיגמות שאינן BP. אפשר לציין כאן תתי-קבוצות של UML בעלות סמנטיקה ביצועית (executable semantics) כדוגמת xUML ו-Foundational UML. שפת Umlple מאפשרת לשלב מודלים של UML בקוד רגיל על ידי מחוללי קוד: המתכנתת עובדת עם מודלים, והמערכת מחוללת מהם קוד רגיל (למשל ב-Java) לפני הרצת המערכת. גישה נוספת לאימות מערכות מומשה על ידי נאס"א בפרוייקט JavaPathFinder, המבצע וורייפיקציה על תוכנות Java אחרי קומפלציה (Java bytecode) על ידי בדיקת כל הריצות האפשריות של המערכת. בדיקה כזו כמובן יסודית מאוד, אולם דורשת זמן ומשאבים הגדלים באופן מעריכי עם סיבוכיות התוכנית. לכן גישה זו יקרה אפילו עבור תוכניות פשוטות, ועלולה להיות בלתי ישימה לתוכנות מסובכות.

עבודה זו מרחיבה את BP, ובוחנת את השימוש בו ככלי מרכזי בהנדסת תוכנה של מערכות תגובתיות, בדגש על תכנון מבוסס מודלים (model-driven engineering). במסגרת עבודה זו פותחו הגדרה מודולרית וניתן להרחבה של BP; פרוטוקול תקשורת בין מודל BP לתוכנות מסורתיות, המאפשר הטמעה של מודלי BP בתוכנות אלו; הגדרת ממשק עבור אלגוריתם בחירת האירועים המאפשרת להשתמש במימוש ספציפי של אלגוריתם גם להרצת תוכנית BP וגם לניתוח שלה; הוספת אידיומים חדשים ל-BP: מטא-דאטה לנקודות סנכרון, סימון הפרות של תכונות בטיחות וחיות, פיצול b-threads, והגדרת קבוצת אירועים שמסיימת את הרצת ה-b-thread; בנייה של מערכת אנליזה, הרצה, ואימות של תוכניות BP, הראשונה שיכולה לזהות הפרה של תכונות חיות ומתבססת על הרצה ישירה (בניגוד לתרגום למודלים אחרים); זיהוי שתי מחלקות של הפרות תכונות חיות ב-BP, ומקרה בו נוח יותר לנסח הפרה של תכונת בטיחות כהפרה של תכונת חיות; מתודולוגיה לכתובת מערכות תגובתיות על בסיס מודל BP; מתודולוגיה לתיאור סמנטיקה של שפות מידול בעזרת BP; תעוד נרחב של מערכת ההרצה והניתוח של BP שפותחה כחלק מהפרוייקט, כולל שני שימושים לדוגמא של המערכת; גישה חדשה לסריאליזציה והשוואת מצב של תוכניות BP (שאף הניבה תרומת קוד לפרוייקט קוד פתוח של Mozilla).

במסגרת עבודה זו אנו מציגים את BPjs - כלי להרצה, ניתוח, והטמעה של תוכניות BP במערכות תוכנה מסורתיות. מערכת זו מתייחסת לתוכניות BP כאל מודל ממוחשב, כך שהרצתן היא בעצם פעולה רגילה על מודל, ולא מקרה מיוחד. בנוסף, מערכת זו היא הראשונה המאפשרת ניתוח של תכונות בטיחות וחיות של תוכניות BP, ומבצעת זאת באופן ישיר - ללא תרגום למודל מסוג אחר וללא שימוש בכלים חיצוניים. בניגוד למערכות קודמות, מערכת BPjs תוכננה כפלטפורמה הניתנת להרחבה מודולרית ולהטמעה במערכות קיימות. לדוגמא, ניתן לשנות את האלגוריתם השוזר את ה-b-threads בקלות, להוסיף קוד המגיב לאירועים אותם המערכת בוחרת, או לבחור אלגוריתם לניתוח תוכניות BP (המערכת הוצגה עם מספר אלגוריתמים, אולם תוכננה כך שיהיה קל להוסיף חדשים). תכונות אלו הופכות את BPjs לתשתית מחקרית ואפליקטיבית לתחום ה-BP בפרט, ולתחום מערכות מבוססות מודלים בכלל.

אחד מהניתוחים החשובים שאפשר לעשות למערכת הוא אימות - בדיקה מלאה כי היא עומדת בדרישות פורמליות (למשל: "כל בקשה לשרת תקבל תגובה", או "אין להפעיל את ציוד התצפית כאשר רמת הטעינה של הבטריה מתחת לסף מסויים"). קיימות שתי קבוצות בסיסיות של תכונות פורמליות של מערכת: תכונות בטיחות (X לא קורה)

ותכונות חיות (במהלך הריצה יקרה Y). מחלקות אלו חשובות, כיוון שכל תכונה של המערכת המתייחסת לזמן באופן לינארי ניתן להביע על ידי שילוב של תכונות אלו. עבודה זו מאפשרת לבדוק את קיומן של תכונות משתי המחלקות האלו במערכת BP. לגבי תכונות חיות - למיטב ידיעתנו זוהי המערכת הראשונה המאפשרת בדיקה זו (ולכן, גם המערכת הראשונה המאפשרת בדיקת תכונות זמן לינארי כלליות).

על מנת לאפשר בדיקה של תכונות אלו, b-threads ב-BPjs יכולים לסמן כי א הם זיהוי הפרה של תכונות בטיחות, (ב) הם מצפים להתקדם מעבר לנקודה מסויימת בקוד. בעת אימות תוכנית BP, המערכת עוברת על כל ההרצות האפשריות של התוכנית. מעבר זה מתבצע על ידי הקפאה של התוכנית בכל נקודת סנכרון שלה, ואז בחינת כל ההמשכים האפשריים משם. למעשה, BPjs עוברת על גרף ההרצה של התוכנית, כאשר נקודות הסנכרון הן הצמתים ואילו האירועים הם הקשתות.

אם, בעת מעבר על גרף ההרצה, BPjs מזהה כי אחד ה-b-threads סימן כי תכונת בטיחות כלשהי הופרה, המערכת מחזירה את נתיב ההרצה שהוביל לשגיאה. נתיב זה כולל את האירועים שנבחרו, ואת מצבי התוכנית. אם, לעומת זאת, המערכת מזהה בגרף ההרצה מעגל שבכל צמתיו b-thread מסויים מסמן שהוא רוצה להתקדם, היא מדווחת על הפרה של תכונת בטיחות. זאת מכיוון שאם המערכת תרוץ לאורך מעגל זה, אותו b-thread לעולם לא יגיע למצב בו הוא יכול להשאר. כאן גם מצאנו הפרדה בין שני מצבים: מעגל בו b-thread יחיד חייב להתקדם כל הזמן (הפרה של תכונת החיות אותה ה-b-thread מייצג), ומעגל בו בכל צומת קיים לפחות אחד b-thread שחייב להתקדם, אבל לכל אחד מה-b-threads ישנו לפחות צומת אחד בו הוא מוכן להשאר. מצב כזה לא בהכרח מהווה הפרה של תכונת חיות – תלוי בתכנון המערכת.

בעת אימות תוכנית BP יש לפעמים להוסיף b-threads שידמו את סביבת התוכנית, או ייצגו דרישות מערכת מסויימות שלא מיוצגות באופן ישיר על ידי b-threads במערכת. לדוגמא, בעת אימות של תוכנית שליטה ברכב אוטונומי, יש להוסיף b-threads שידמו את סביבת הרכב על ידי בקשת אירועי טלמטריה, או ידמו תקלות מסויימות על ידי יצירת דיווחים אודותיהן. בנוסף, ניתן להוסיף b-threads המייצגים דרישות מערכת בסגנון "לבסוף הרכב מגיע ליעד". בנוסף, אם רוצים למקד את האימות בריצות מסוג מסויים, ניתן להוסיף b-threads שיחסמו את הריצות שאינן מסוג זה.

כאמור לעיל, קיימות שפות מידול פורמלי רבות, המתאימות לתיאור אספקטים שונים של מערכות. בפרט, בבניה של מערכת תגובתית מבוססת מודל, נרצה לעיתים להשתמש בשפה שאינה BP על מנת לתאר תכונות מסויימות של המערכת, בד בבד עם תיאור חלקים אחרים של המערכת ב-BP. בנוסף, נרצה להמשיך להשתמש בתשתית התיאורטית והטכנולוגית להטמעת מערכות BP במערכת המסורתית המשמשת לקידוד החלקים ה-"נמוכים" של המערכת (למשל, קריאת חיישנים והפעלת מנועים). לצורך זה, עבודה זו מציגה מתודולוגיה לתאור סמנטיקה של שפות מידול בעזרת BP. המתודולוגיה מתבססת על ביצוע שאילתות על קוד המקור של המודל (לאו דווקא טקסטואלי - יכול להיות גם מבוסס שרטוטים). התוצאה של כל שאילה היא אוסף מבנים בשפת המידול. לפי המתודולוגיה המוצעת כאן, מבנים אלו מתורגמים ל-b-threads, ומתווספים לתוכנית BP. לאחר ביצוע סדרה של שאילתות כאלו, נוצרת תוכנית BP שסמנטיקת ההרצה שלה לזו של המודל בשפה המקורית. אנו מדגימים מתודולוגיה זו על שפת LSC.

שיטת הגדרה זו מאפשרת לא רק לשלב מודלים משפות שונות במודלים של BP, אלא גם ליצור מנועי הרצה לשפות מידול בעלות סמנטיקה ביצועית, לבחון בקלות יחסית שינויים בסמנטיקה של שפות מידול, ולהציג את הסמנטיקה של שפות מידול באופן נגיש למתכנתים, שרגילים לקרוא קוד ופחות רגילים לקרוא נוסחאות מעברים מתמטיות. על מנת לבחון את השימוש ב-BP כשכבת המודל במערכות תגובתיות מבוססות מודלים, עבודה זו מציגה שני מקרי בוחן (case studies). הראשון בודק את המתודולוגיה על רכב אוטונומי שנדרש לעקוב אחרי רכב אוטונומי אחר; השני משתמש במתודולוגיה על מנת לכתוב תוכנת הרצה ללויין, ועל מנת לבדוק אותה. במקרה הבוחן של הרכב האוטונומי, מציג תוכנית שלטיה ברכב שמטרתו לעקוב אחרי רכב אחר, תוך שמירת מרחק בטוח (לא רחוק מדי ולא קרוב מדי). תכנון המערכת מתבסס על הטמעת מודל BPjs בתוכנת Java, כאשר המודל אחראי על החלטות המערכת (מתי לפנות, כמה להאיץ) ואילו שכבת ה-Java אחראית על קריאת החיישנים ומסירת המידע למודל מחד, והאזנה להחלטות המודל וביצוען מאידך. המודל עצמו נבנה בגישה של התנהגות דיפולטית עם החרגות, והורכב משלושה b-threads. ה-b-thread הראשון אחראי על ההתנהגות הדיפולטית ודורש לנסוע קדימה בשיא המהירות כל הזמן. שני b-threads נוספים אחראים על החרגות: הראשון מתקן את הכיוון של הרכב כאשר אינו פונה לכיוון הרכב המוביל, והשני נוסע קדימה לאט כאשר הרכב מתקרב למוביל יתר על המידה. בעזרת מנגנוני האימות, הצלחנו לזהות שגיאות במודל, כגון מרוץ (race condition) שגרם לרכב להתקדם במהירות לא נכונה בתנאים מסויימים.

במקרה הבוחן של הלויין - חלק מפרוייקט ארוך טווח לבניית מעבדה לתוכנות לוייניות מבוססות BP - נבנתה מערכת בקרה עם ארכיטקטורה דומה (מודל BPjs מוטמע בתוך תוכנית Java). במסגרת העבודה המוצגת כאן מומשו שני מודולים, האחד אחראי על ניהול האנרגיה והשני על בקרת הכוון, וכן אלגוריתם המתאם ביניהם. המערכת עצמה רצה על מחשב לווייני הנמצא במעבדה. על מנת לדמות תנאי טיסה אמיתיים, המחשב הלווייני מחובר לסנסורים, או למכשירים המדמים סנסורים ומכשירים שונים (למשל מדמי גופי חימום). מכשירים אלו מחוברים למחשב שולחני שקובע אילו נתונים הם ישלחו על מנת לדמות סביבת טיסה. נתונים אלו נקבעים על ידי תוכנת STK, המיועדת לסימולציה של משימות חלל, ועל ידי תוכנית MATLAB, המחשבת סימולציה של נתוני הלויין. גם במערכת זו אימתנו את המודל שאחראי על החלטות המערכת, על מנת לוודא כי אינו מכיל שגיאות.

ניתן להמשיך את העבודה המוצגת כאן לשני כיוונים: פנימה (חקר BP) והחוצה (שימוש ב-BP ככלי להנדסה מבוססת-מודלים של מערכות תוכנה). כלפי פנים, עבודה זו מניחה תשתית תיאורטית המגובה בכלים מעשיים לבדיקה של אלגוריתמי בחירת אירועים, אנליזה של תוכניות BP, סקירת גרפי ההרצה שלהן, וכן שימושים נוספים בתשתיות ההרצה והאנליזה של BPjs. כלפי חוץ, עבודה זו מדגימה איך ניתן להשתמש ב-BP ו-BPjs על מנת לבנות מערכות תגובתיות מבוססות מודלים. על בסיס ידע זה ניתן לבנות מודלים מסובכים יותר, לחקור תבניות תכן שימושיות (design patterns) של תוכניות BP, לחקור מודולריזציה ושימוש חוזר בקוד (בפרט, ב-b-threads), ולפתח כלים נוספים המאפשרים הנדסת מערכות תוכנה תחת מתודולוגיה זו.

מילות מפתח: תכנות התנהגותי, הנדסת תוכנה, תכנון מבוסס מודלים, אימות תוכנה, דרישות פורמלית, בקרה.