Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures

David Harel
Weizmann Institute of Science
Rehovot, Israel
david.harel@weizmann.ac.il

Guy Katz
Weizmann Institute of Science
Rehovot, Israel
guy.katz@weizmann.ac.il

ABSTRACT

Behavioral programming (*BP*) is a decentralized scenario-based paradigm for the programming of reactive software, geared towards incremental and intuitive development. In this work we apply the principles of BP to a large, real-world case-study: a web-server. We discuss the conclusions learned from our attempt and propose several extension idioms to BP, aimed at improving the framework's scalability. Specifically, we propose extending BP with a timeout idiom for handling various time constraints, program-specific execution strategies, dynamic thread creation for efficiently allocating system resources, and support for parameterized events to handle inputs with infinite domains. Our extensions and case-study are implemented in a new framework for behavioral programming in C++.

Keywords

Behavioral programming, reactive systems, time constraints, TCP, HTTP, C++.

1. INTRODUCTION

Scenario-based programming [21] is a recently proposed approach to the development of decentralized reactive systems. At its core is the notion of programming through the specification of *scenarios*, each of which corresponds to a certain aspect of the system's behavior, not necessarily restricted to a particular component. When composed together according to a certain set of rules, the scenarios yield cohesive system behavior.

One of the main difficulties in this type of design is to ensure that the large variety of scenarios, each with their own unique characteristics and viewpoints, inter-operate correctly [28]. In particular, race conditions and unpredicted interweaving of scenarios can result in incorrect system behavior. Thus, it is desirable to define inter-scenario interfaces that are sufficiently simple to minimize these effects, but which are still powerful and expressive enough to be of practical use.

In this work, we focus on *behavioral programming* (*BP*) [24] — a particular form of scenario-based programming that originated from *live sequence charts* (*LSCs*) [12, 21]. Behavioral programs consist of a collection of special *behavior threads*, each describing a certain facet of the system's behavior. When run in parallel, the threads repeatedly synchronize with each other at predetermined points. Each synchronization point is then resolved by an *event selection mechanism* (*ESM*) that selects an event for triggering based on information gathered from all the threads. Thus, all components are consulted in choosing the global course of action, yielding the desired composite behavior.

The specific structure of behavioral programs, and in particular the simple yet strict thread interaction rules, has beneficial effects: programs can be written in a "natural" way, i.e., with modules that are aligned with the specification [14, 24]; it greatly reduces the amount of (unexpected) interleaving, hopefully making race conditions more scarce; it renders the program more amenable to incremental development; and it facilitates the application of program analysis methods to behavioral systems [20] (for more details, see Section 2). Consequently, it has been conjectured that behavioral programming is suitable for the development of large systems. However, to the best of our knowledge, this hypothesis has not yet been put to the test. Here we set out to do just that, by attempting to implement a real-world system in BP. Apart from checking the feasibility of the task, we were interested in assessing whether the aforementioned traits of BP, such as incremental development and the alignment of code with the specification, could indeed scale-up in a large system.

For our case-study we chose to implement protocol stacks of two common and elaborate protocols, TCP and HTTP. TCP (*Transmission Control Protocol*) is a connection-oriented protocol, used mainly for the transfer of data across the internet. The principal feature of TCP is its reliability: it guarantees that data arrives without errors and in the order in which it was sent. HTTP (*Hypertext Transfer Protocol*) is a request-response protocol, aimed at allowing clients to retrieve information from remote hosts. We have implemented and combined these two protocol stacks, creating a working web-server.

Our motivation in choosing this particular project was its volume, and also our desire to test BP's effectiveness in handling the large variety of coding situations the project entails: handling timeouts, string manipulation, file access, checksum calculations, handling multiple inputs, mandatory and forbidden user behaviors, etc.

As we began constructing the system we noticed that the existing, "traditional" idioms of BP were inadequate for dealing conveniently with certain programming tasks. Some of these tasks could be performed by BP but required employing ad-hoc solutions that bypassed the built-in infrastructure, whereas others (e.g., those related to time) were downright inexpressible using traditional BP, and required incorporating external mechanisms into the program [25] — in both cases, defeating the purpose of BP's simple and intuitive interfaces.

As the development of our case-study progressed, we were able to classify the difficulties we faced, placing them in four categories. For each of these, we were able to come up with an extension idiom to BP that allowed convenient programming solutions. In defining these new idioms, we attempted to retain as much of the simplicity and intuitiveness of the original framework as possible.

The first and foremost difficulty we encountered was the issue of time: traditional BP assumes all transitions in the systems take "zero time" (as per the *synchrony hypothesis* [10]), and does not provide a mechanism for bounding the flow of time between events. The TCP protocol makes abundant use of timers and timeouts, and it was unclear to us how to implement it in BP. To overcome this difficulty we extended BP with the notion of *timeouts* which, as demonstrated later in the paper, allowed us to express the required time constraints for our program.

The second problem we faced was the need for program-specific strategies in BP. The BP infrastructure dictates that at every synchronization point of the execution the program may have to choose between several legal execution paths, but it does not specify which should be chosen. Various schemes have been suggested in the past, which allow the user to partially prioritize between these paths, each scheme with its unique advantages and disadvantages. In the later stages of the development of our case-study, when the system contained numerous modules running in parallel, we observed that several requirements — especially those pertaining to prioritization between the modules — could be naturally enforced through a more intelligent path selection method. Unfortunately, the existing mechanisms were too restrictive. We consequently extended the framework to allow programmers to supply their own path selection strategies, per program, in a way that provided sufficient flexibility for our needs.

The third type of difficulty we encountered was that of dynamic thread creation. We wanted our web-server to be able to handle different volumes of activity: that is, to allocate more resources when traffic was high and to release them when it was low. A natural approach was to employ dynamic creation and destruction of threads, which is not allowed in the traditional BP semantics. In [20,23] a limited variant of this concept was proposed, in order to regulate external input to a behavioral program. We opted to generalize and formalize this idea, allowing it to be used anywhere within the program, and we then used it in our implementation to dynamically allocate more threads according to traffic.

Finally, the last issue we encountered was that of *parameterized inputs and events.* Previous work on scenrio-based programming and BP revolved around programs with a small bounded pool of inputs, whereas the input domain of a web-server is practically infinite. In order to implement our example, we thus generalized traditional BP to support parameterized inputs.

Having added these idioms to the BP framework, we were able to accomplish our implementation goals. We implemented our case-study in a new framework for behavioral programming in C++, termed *BPC*, which we present here, and which supports the extensions described above. The framework and code for the case-study are available online [2].

Returning to our original question of whether or not BP is suitable for the development of large systems, we believe that our case-study answers this affirmatively — provided that the above idioms, or similar ones, are made available. As for our secondary goal, namely to check whether BP's naturalness and incrementally would scale-up in a large system: while these properties are inherently difficult to quantify, the conclusions from our case-study seem to support an affirmative answer here, as well. We discuss this matter in later sections.

The remainder of the paper is organized as follows. We begin by describing the traditional semantics of BP in Section 2. In the succeeding sections, we discuss the difficulties we encountered in our case-study, one by one, and the new idioms used to overcome them. Each of the sections includes a small illustrative example and a discussion of where the difficulty occurred in the web-server application: time constraints are discussed in Section 3, customizable strategies in Section 4, dynamic thread creation in Section 5 and parametrized events in Section 6.

Sections 7 and 8 are dedicated to the BPC framework and the details of the implementation of our case-study, respectively. Related work appears in Section 9, and we conclude in Section 10.

The rigorous semantics of our extended variant of BP is available as supplementary material in [3].

2. BEHAVIORAL PROGRAMMING

2.1 Overview

In this section we describe the existing principles and idioms of behavioral programming, which, in the following sections, we refer to as "traditional BP".

A behavioral program consists of modules, each representing a different facet of behavior in the system. These modules, termed *behavior threads* (abbr. *b-threads*), are interwoven at run time: they all synchronize at certain points throughout the execution, and together steer the course of the system. Threads may also perform local computation between synchronization points.

At every synchronization point, each b-thread declares sets of events to be considered for triggering (*requested events*) and events whose triggering it forbids (*blocked events*), and then pauses. When all threads have submitted their declarations, an *event selection mechanism* (abbr. *ESM*), resolves the synchronization point: it computes the set of *enabled events* — events that are requested by at least one thread and blocked by none — and selects one of them for triggering. As soon as this event is selected, all threads that requested it are notified and are allowed to resume their execution. Threads may also react to triggered events that they did not request, in which case they are said to be *waiting-for* those events. The model disallows inter b-thread communication that does not occur through the synchronization mechanism.

Scenario-based programming in general, and BP in particular, share many traits with the actor model [5]: behavioral threads, like actors, are intended to be small pieces of code, aware only of their specific interests and goals. Indeed, we regard BP as complementary to the actor-oriented and agent-oriented paradigms. The main motivation behind BP, however, is that through its strict and simple synchronization mechanism — all threads repeatedly synchronize, and interact only indirectly, through requested and blocked events — it facilitates incremental, non-intrusive development, by writing threads that are aligned with the specification [24]. This is demonstrated in the example of Fig. 1, borrowed from [17]. Further, studies indicate that BP is *natural* in the sense that it is easy to learn, and that it fosters abstract programming [6, 14].

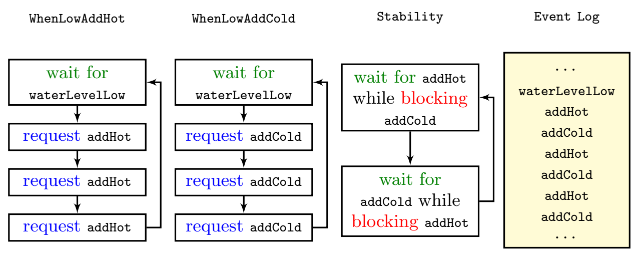

Figure 1: (From [17]) An example of the incremental development of a system for controlling water level in a tank with hot and cold water sources. At first, b-thread *WhenLowAddHot* is created; it repeatedly waits for waterLevelLow events and requests three times the event addHot. It is then discovered that adding just three water quantities for every sensor reading is insufficient, and b-thread *WhenLowAddCold* is added. It performs a similar action to that of *WhenLowAddHot*, but with event addCold. Then, when *WhenLowAddHot* and *WhenLowAddCold* are executed simultaneously, the run may include three consecutive addHot events followed by three addCold events. A new requirement is thus introduced, to the effect that water temperature should be kept stable. We incrementally add the b-thread *Stability* to enforce the interleaving of addHot and addCold events, without modifying existing code.

Behavioral programming can be regarded as a kind of "grand" design pattern, and thus can be implemented on top of high-level programming languages; examples include Java, Erlang and Google Blockly, among others. See [24] and references therein, and also the behavioral programming webpage [34].

Behavioral programming has been successfully used to develop various small applications. In [23], for instance, the incremental development of an application for playing Tic-Tac-Toe is described. The threads in that application are perfectly aligned with the specification of the game: turn alternation is enforced by a thread that repeatedly waits for all X moves while blocking O moves, and then waits for all O moves while blocking X moves; for each square there is a dedicated thread that waits for that square to be marked, and then prevents further markings; other threads detect wins by either of the players; etc. See [23] for further details. Other examples include, e.g., a maze solving robot [23], a rocket-landing game [32] and a quadrotor flight simulator [25].

2.2 Semantics

As inter-thread communication is allowed only through the event selection mechanism, we can regard a thread as being "in state" only when it is at a synchronization point. Given a global set of events E, we define a thread BT to be the tuple $BT = \langle Q, q_0, \delta, R, B \rangle$ where Q is the set of states (i.e., synchronization points), q_0 is the initial state, $\delta \subseteq Q \times E \times Q$ is a transition relation, $R : Q \to 2^E$ is a labeling function that maps a state to the set of events requested in that state, and $B : Q \to 2^E$ is a labeling function that maps a state to the set of events blocked in the state. Observe that there is no labeling function for waited-for events: the notion of waiting is expressed via the transitions between states. We use $\delta(q, e)$ to denote the states reachable from state q by event e, i.e., $\delta(q, e) = \{\tilde{q} \mid \langle q, e, \tilde{q} \rangle \in \delta\}$.

A behavioral program P is a collection of threads $\{BT^1, \ldots, BT^n\}$, where $BT^i = \langle Q^i, q_0^i, \delta^i, R^i, B^i \rangle$. A *configuration* γ of the program is a tuple $\langle q^1, q^2, \ldots, q^n \rangle$ where $q^i \in Q^i$; the initial configuration is $\gamma_0 = \langle q_0^1, q_0^2, \ldots, q_0^n \rangle$. A configuration $\tilde{\gamma} = \langle \tilde{q}^1, \ldots, \tilde{q}^n \rangle$ is the successor of configuration γ with respect to event $e \in E$, denoted $\gamma \xrightarrow{e} \tilde{\gamma}$, if and only if conditions 1 and 2 hold:

$$e \in \bigcup_{i=1}^{n} R^i(q^i) - \bigcup_{i=1}^{n} B^i(q^i) \tag{1}$$

$$\forall_{1 \le i \le n}, \quad \begin{cases} \delta^i(q^i, e) \neq \emptyset \implies \langle q^i, e, \tilde{q}^i \rangle \in \delta^i \\ \delta^i(q^i, e) = \emptyset \implies \tilde{q}^i = q^i \end{cases} \tag{2}$$

That is, event e is *enabled* (i.e., requested and not blocked) in configuration γ, and all threads react to event e as defined by their respective transitions rules: if an edge for event e exists, then such an edge is traversed. Otherwise, the threads stay in the same state. Intuitively, the existence of an edge implies that the thread waited for event e.

An execution ρ of P is a sequence of consecutive configurations starting at the initial configuration, $\rho = \gamma_0 \xrightarrow{e_0} \gamma_1 \xrightarrow{e_1} \ldots$. The execution can be infinite, or finite if it ends in a *terminal* configuration — a configuration with no successors. The *run* of ρ is the event sequence $e_0, e_1, \ldots$; and the set of runs of all valid executions of P is called the *language* of P, and is denoted $\mathfrak{L}(P)$.

2.3 Handling Input

The synchronous nature of BP dictates that the system may only progress when all threads have synchronized. Consequently, a thread that is performing some blocking *read* operation would render the rest of the system unable to process any events. This makes it difficult to design *sensor threads* — threads that wait for user input and then request events to notify other threads of this input. A naive way to design such a thread appears in Fig. 2.

In order to overcome this difficulty we use the solution proposed in [16], denoted *eager execution*. The idea is that if a thread that has not yet synchronized is known to never block an event, and that event is enabled with respect to the other threads, then it may be immediately triggered without waiting for the delayed thread. Events triggered this way are then stored in a dedicated queue, and when the delayed thread finally catches up it processes them. As a particular case, sensor threads can always be declared to block no events, allowing the rest of the system to operate normally. See [16] for more details.

```
while( true ) {
    waitForButtonClick();    // Returns only on click
    bSync({buttonClicked}, ∅, ∅);
}
```

Figure 2: Pseudocode for a naive implementation of a sensor thread. The thread runs in an infinite loop, in each iteration waiting for input via the blocking call `waitForButtonClick` and then requesting a buttonClicked event. The `bSync` call is the synchronization API method: its first parameter (marked in blue) is the set of requested events, the second (green) is the set of waited-for events, and the third (red) is the set of blocked events. `bSync` only returns when an event that was requested or waited-for has been triggered. In particular, this enforces the convention that a thread implicitly waits for every event that it requests — though it may also wait for additional events, that it did not request. Here, the only requested event is buttonClicked, and the other two event sets are empty. When waiting for a button click, this thread prevents the entire system from triggering any events.

3. SUPPORT FOR TIME CONSTRAINTS

The traditional BP semantics suffices when the system in question is time oblivious — i.e., when its response to external inputs takes negligible time and there are no constraints on the instance in time in which events may be triggered. But what happens when that is not the case? Consider, for instance, a behavioral program for a railway crossing. Suppose that a sensor thread signals the approach of a train by generating a lowerGate event. Also, say the gate is to remain down for 30 seconds, after which another thread is to request a raiseGate event. The simplest approach is to encode this thread as depicted in Fig. 3.

```
while( true ) {
    bSync(∅, {lowerGate}, ∅);
    sleep(30);
    bSync({raiseGate}, ∅, ∅);
}
```

Figure 3: A thread that waits for lowerGate, sleeps for 30 seconds, and then requests a raiseGate event.

Unfortunately, this thread pauses the execution of the entire system during its sleeping periods.

One way to tackle this difficulty is to delegate timing responsibilities to a non-behavioral component, and have the system communicate with it using *external events* [25]. However, this solution requires that the programmer goes beyond the scope of BP.

Another approach is to use the eager execution mechanism, as was the case with sensor threads. However, eager execution has limited applicability: consider, for instance, the stronger variant in which the programmer wishes to *block* the raiseGate event for the 30 seconds following a lowerGate event, to negate any accidental requests made by other threads. This stronger requirement is difficult to accommodate using the eager synchronization mechanism, as there is no way to readily inform the blocking thread that 30 seconds have passed.

Programming tasks in which time plays a role appeared frequently in our TCP stack implementation. The TCP protocol guarantees that data arrives without errors and in the order in which it was sent. To accomplish this, the end parties acknowledge the reception of each TCP segment using a scheme of agreed-upon sequence numbers; segments that are lost or arrive corrupt are not acknowledged, and are then retransmitted. Thus, a TCP stack needs to keep track of the time passed since sending each outgoing TCP segment, and retransmit it unless an acknowledgment is received within a certain time window.

To support these requirements, we extended BP with a *timeout* idiom. This idiom allows each thread to declare, at every synchronization point, a timeout value — in addition to the requested, waited-for and blocked events. The timeout value indicates the maximal number of seconds the thread is willing to wait, in synchronized state, for a requested or waited-for event to be triggered, after which it "withdraws" its synchronization and associated event declarations. In practice, this means that the synchronization call returns and the thread resumes. The programmer can check if the call returned due to a triggered event or because of a timeout.

Using the timeout parameter, the thread depicted in Fig. 4 guarantees that the railway crossing system does not generate an early raiseGate event.

```
while( true ) {
    bSync(∅, {lowerGate}, ∅, ∞);
    bSync(∅, ∅, {raiseGate}, 30);
}
```

Figure 4: An implementation using the timeout parameter. Event raiseGate is blocked for 30 seconds.

The thread waits for a lowerGate event, and then spends the successive 30 seconds blocking raiseGate events. Counting these 30 seconds begins the instant the thread synchronizes, and does not depend on other threads. Since the thread neither requests nor waits for any events, the blocking is guaranteed to continue for the full 30 seconds, after which the call returns, and the thread waits for additional lowerGate events. The special ∞ symbol passed as the timeout parameter implies that the thread is willing to wait indefinitely; in fact, using this value produces the same result as the traditional synchronization interface.

The timeout idiom can also be used to implement a TCP retransmission scheme, as shown in Fig. 5. The thread transmits the packet, and then waits for an acknowledgment for 2 seconds. If the acknowledgment is not received, the second synchronization call returns due to a timeout, and the process repeats. We describe our implemented retransmission mechanism in greater detail in Section 8.

```
do {
    bSync({sendSegment}, ∅, ∅, ∞);
    bSync(∅, {acknowledgment}, ∅, 2);
} while ( timeoutInLastSync() )
```

Figure 5: A retransmission scheme. The thread waits for an acknowledgment for 2 seconds, and if it fails to arrive — retransmits the segment.

The proposed idiom appears to be natural and intuitive, and thus compatible with the rest of BP's idioms. Indeed, one of our goals was to stick to simple and intuitive idioms whenever possible. The timing idiom is also quite expressive, allowing a variety of behaviors that were previously beyond the direct scope of BP, such as "block for x seconds" (as in the railway example), "request for x seconds and then default", or just "sleep for x seconds" without delaying the system.

In our view, a call to `bSync` that returns due to a timeout is not considered an error — rather, it is just another possible outcome of the synchronization attempt. Thus, timeouts are not regarded as exceptions, but as a mechanism for coping with thread transitions that are not immediate: if a thread is delayed in reaching its synchronization point, other threads may react (when their timeouts expire) and change their event declarations.

In [20], the authors present a model-checker for behavioral programs, capable of handling safety and liveness properties. An interesting aspect of this model-checker is that it receives the property in question in the form of a b-thread: a thread that waits for unwanted event sequences and marks states as bad (safety), or a thread that waits for good event sequences and marks states as good (liveness). We observe that the addition of timeouts allows us to express, e.g., time-related safety properties. For instance, consider a system in which every e_1 event is always followed by an e_2 event, and suppose that we wish to verify that at most 2 seconds pass between every e_1 and e_2 pair. This property can be expressed by the b-thread depicted in Fig. 6.

```
while( true ) {
    bSync(∅, {e1}, ∅, ∞);
    bSync(∅, {e2}, ∅, 2);
    if ( timeoutInLastSync() )
        bSync({error}, ∅, ∅, ∞);
}
```

Figure 6: A thread that requests an error event if event e_2 is not triggered within 2 seconds of event e_1.

This sort of constraint can be useful for model-checking (by extending the tool of [20] to support timeouts); and it can also give rise to a lookahead mechanism that influences the choice of triggered events so as to satisfy the constraint, similarly to the smart play-out mechanism of [18, 19]. Both directions are left for future work.

Technically, thread timeouts are triggered by the event selection mechanism, similarly to the way regular events are handled. Further details appear in Section 7.2 and as supplementary material in [3].

4. CUSTOMIZABLE EVENT SELECTION

BP's traditional semantics dictates that in every synchronization cycle one event that is requested and not blocked is triggered. However, if there is more than one viable event for triggering, it is unspecified which of those will be selected.

In practice, however, it is often useful to have events selected using a certain strategy. For instance, consider the following system that manages a smart door lock. When a person approaches, he/she must place the appropriate identification card on a reader, and a behavioral program decides whether or not to let the person through. A simple design for the program is to have a dedicated thread handle the lock; and whenever an id card is scanned, have it request an open event, which is translated by an actuator thread to opening the actual lock.

Next, suppose some people should be denied passage — e.g., if they do not appear in the white list, if they appear in the black list, or if their access card has expired. We assume that the open event has an *id* parameter, that identifies the person, and that other parts of the system may block open events for certain *id*s. However, as the lock thread does not know in advance which events are blocked, it cannot just request the open event, or the system could get stuck if that event is blocked. One possible solution appears in Fig. 7: the thread requests, along with an open event, also an idle event. If the open event is blocked, the idle event is processed, and the thread can continue processing future requests.

```
while ( true ) {                                    // Door thread
    bSync(∅, {request}, ∅, ∞);
    bSync({open(lastEvent().id), idle}, ∅, ∅, ∞);
}

while ( true ) {                                 // Blocker Thread
    bSync(∅, ∅, {open | open.id is in black list}, ∞);
}
```

Figure 7: A sketch of the lock program. The door thread waits for requests, and tries to grant them. Other threads may block the open(id) event for certain values of the id parameter. As the door thread has no way of knowing whether a specific event is blocked or not, it also requests an idle event, to allow itself to ignore the request and move on to process future requests.

In order for this scheme to work properly and not deny entrance to authorized ids, we need to be certain that any open event will take precedence over the idle event. Thus, we need a way to enact a certain strategy for how enabled events are to be selected for triggering.

On several occasions we encountered similar, though more complex, situations in our case-study. In one case, during tests with multiple simultaneous connections, we observed that some clients would get starved. Hence, we wanted to enforce the requirement that higher priority be given to requests from starving connections. In another case, we wanted to ensure that segment-sending requests always received a higher priority than connection-termination requests, so that segments were never sent on closed connections.

Previous work discussed several event selection strategies, such as thread-based priority, round robin, random and arbitrary selection [23]. The BPJ framework, for instance, uses the thread-based priority scheme. Through our case-study and the specific requirements that it entailed regarding event selection, we came to recognize that no one strategy fitted all programs, and that it was useful to allow programmers to supply their own program-specific strategy.

Consequently, we extend BP's event selection in the following way. Let Γ denote the set of possible system configurations. In its most general form, an event selection strategy is a function $f_{es} : \Gamma^* \times \Gamma \to E$, which takes as input the history of previous system configurations and the current configuration, and chooses an event for triggering from among the enabled events. The selection function is supplied by the programmer and is considered part of the behavioral program rather than of the BP framework. Apart from subsuming the above mentioned mechanisms, this approach also allows event selection strategies that change over time, such as learning [13] or look-ahead algorithms.

Technically, the BPC framework allows the programmer to provide a callback object to manage event selection. In particular, modifying the selection strategy does not entail recompiling the framework. For more details on the specific strategy used in our case-study and its implementation, see Section 8.

5. DYNAMIC THREAD CREATION

A reactive application may, throughout the course of its run, have to deal with varying volumes of activity. It is desirable to have applications that adjust — that is, dedicate more computational resources — when activity is high, and free them when they are no longer needed. This goal may be difficult to achieve with traditional BP.

Consider, for instance, a mail client application, which takes as input an email address and the body of a message and then sends it. Further suppose that the application waits for an acknowledgment message for every email sent. It must thus keep track of previous mails that have yet to be acknowledged.

One possible design for such an application is to direct all requests and acknowledgments to a single thread, which can then keep track of traffic, using an internal database. A cleaner solution, however, is to have multiple threads, each in charge of sending a single message and tracking its acknowledgment. The resulting threads are simpler and do not require a database, and are thus less prone to error.

The question then arises of how many of these threads we should instantiate. Statically determining this number would raise the risk of not having enough resources when many requests are performed simultaneously, and the risk of wasting computational resources when traffic is low.

To resolve this issue, we propose to allow threads to dynamically spawn other threads, and similarly, to allow threads to terminate during execution. That way, new threads can be instantiated when needed, and can be terminated when they are no longer needed, freeing system resources. An implementation of the above program using dynamic thread spawning appears in Fig. 8.

```
while ( true ) {                          // Dispatcher thread
    bSync(∅, {mail}, ∅, ∞);
    new Sender(lastEvent().address, lastEvent().text);
}

do {                                          // Sender thread
    bSync({send(address,text)}, ∅, ∅, ∞);
    bSync(∅, {ack(address)}, ∅, 10);
} while( timeoutInLastSync() );
```

Figure 8: The *Dispatcher* thread waits for incoming mail requests. For every such request, it dynamically creates a new *Sender* thread and passes to it as parameters the address and text fields of the request. Each *Sender* instance deals with just one mail, which was passed to it during construction. Immediately upon its instantiation, the thread sends the mail and awaits an acknowledgment. If no such acknowledgment is received within 10 seconds, the mail is resent. As soon as the acknowledgment has been received, the thread terminates.

We faced similar situations in our case-study. In particular, the TCP message acknowledgment scheme is very similar to the above example, and indeed we implemented it by spawning dedicated threads that wait for message acknowledgment. Further, each active connection in the protocol stack requires some bookkeeping (e.g., the state of the connection, last received incoming sequence number, and last used outgoing sequence number), and we explored implementation variants, in which these bookkeeping threads were spawned per connection, in order to improve efficiency. More details appear in Section 8.

The reader may notice that, assuming that the address and text parameters in Fig. 8 are unbounded, there are infinitely many versions of thread *Sender* that may be created throughout the run. Indeed, the traditional definition of behavioral programs as a set of threads that exist through the program's run no longer applies in the face of dynamic thread creation. Instead, we associate a behavioral program with a set of *thread templates* — copies of which may be instantiated at different times throughout the run. See the supplementary material in [3] for a rigorous definition.

We point out that the concept of dynamic thread creation was already introduced in [23], where the authors used dynamically created *sensor threads*. These threads could only be spawned by non-behavioral components, and were only used to signal user input. In contrast, we allow the dynamic creation of general threads by other threads throughout the program (and use the mechanism of [16] to manage system input).

Dynamic thread creation is a useful feature, but it also has its tolls: in our BPC implementation, each behavioral thread is presently implemented as a POSIX thread, and so the creation of a large number of thread incurs a overhead. We plan to mitigate this problem by implementing of a more efficient, lightweight threading mechanism, similar to the one used, e.g., in Erlang [8].

6. PARAMETERIZED INPUT

In this section we examine another issue that may arise in applying the BP principles to a large system: the need to handle a pool of (practically) infinitely many possible inputs. This requirement is quite common; in particular, it arose in our case-study, where the system had to handle incoming TCP segments, i.e., byte sequences of unknown length.

For illustration, consider a simple program that takes as input a number x and checks whether it is a multiple of 3 and also ends with the digit 5. One approach to writing such a program would be to have a sensor thread wait for inputs, and then broadcast them to the rest of the system. One checker thread would then check whether x is divisible by 3, and another would check whether x ends with 5. Using event blocking, the two checker threads could then reach a combined decision on the final answer.

The input parameter x is unbounded, and as the BP framework dictates that threads only exchange information through the synchronization mechanism, this implies that the event set E of the program must be infinite. A convenient way to facilitate handling infinite event sets is to extend the definitions of traditional BP, and allow events with unbounded parameters. This is a generalization of an approach that appeared in examples described in [20]; there, the authors used events with bounded parameters to facilitate waiting-for or blocking finite sets of events. Pseudocode for the program described above, using parameterized events, appears in Fig. 9.

```
while ( true ) {                          // Sensor thread
    int x = readInput();
    bSync({check(x)}, ∅, ∅, ∞);
}

while ( true ) {                   // First checker thread
    bSync(∅, {check}, ∅, ∞);
    if ( ( lastEvent().x % 3 ) == 0 )
        bSync({good}, {bad}, {check}, ∞);
    else bSync({bad}, ∅, {good, check}, ∞);
}

while ( true ) {                  // Second checker thread
    bSync(∅, {check}, ∅, ∞);
    if ( ( lastEvent().x % 10 ) == 5 )
        bSync({good}, {bad}, {check}, ∞);
    else bSync({bad}, ∅, {good, check}, ∞);
}
```

Figure 9: A program that takes as input a number x, and decides whether it is a multiple of 3 that ends with 5. The *Sensor* thread waits for exterior inputs, and translates them into a parameterized event check. This event is waited for by the two checker threads, each of which checks one of the two conditions. Both threads then proceed symmetrically: if the condition holds, they request event good; otherwise, they request event bad and block event good. Thus, event good is triggered if and only if both conditions hold for x. If either thread discovers that its respective condition does not hold, event good becomes blocked and cannot be triggered, resulting in the triggering of bad. Observe that when handling a previous request both threads block new check events, to delay new inputs until they can be processed.

Using this idiom in our case-study, we employed a sensor thread to read incoming segments, and for each segment to request a tcpSegmentReceived event — with the segment as its parameter. The segment could then be processed by other threads. Other parts of the system also made use of parameterized events; for instance, events associated with incoming HTTP requests carried ip and port parameters, indicating the address to which a response to the request needed to be sent. For additional details, see Section 8 of the paper.

We observe that in many cases, dealing with unbounded parameterized events calls for threads with infinitely many states — states that may depend on the parameters. For instance, consider the sensor thread in Fig 9. A "state" of the thread is a synchronization point with fixed requested, waited-for and blocked events. For any input x, the thread requests different events; and hence, it must have as many states as there are x's. Therefore, this extension calls for allowing infinite state sets in the rigorous semantics of BP; See supplementary material in [3].

Finally, once we allow unbounded parameterized events we should consider that threads may wish to wait-for or block infinitely many events (as in the example of Fig. 9). More complex cases include waiting-for or blocking events depending on their parameters. In BPC, we follow the example of the BPJ framework [23] for BP in Java, and allow parameter-dependent blocking or waiting-for events via *event predicates* — functions that take events and answer *true* or *false*. When a thread uses a predicate to indicate its waited-for events, it is notified of a triggered event if that event causes the predicate to evaluate to true, and similarly, if it uses a predicate to indicate its blocked events, an event can be triggered only if the predicate returns *false* for that event. As in BPJ, we require that the requested events be explicitly declared, so that set must be finite. This is required for the event selection process in the ESM.

7. THE BPC FRAMEWORK

In this section we present the BPC framework for behavioral programming in C++, which implements the extensions discussed in previous sections. It is available online at [1]. The framework is designed to allow the user to conveniently define and write behavior threads while using the full power of the C++ programming language. The synchronization and coordination mechanism is implemented as part of the framework, and is concealed from the user.

7.1 User Interface

Behavior threads are implemented as classes that inherit from the *BThread* class. They are customized to carry out particular behavior by overriding the `entryPoint` method, which the framework invokes when the thread starts. The interface provided by the parent class includes the `bSync` method to perform thread synchronization and the `lastEvent` method to retrieve the result of the last synchronization point — be it an event or a timeout. The `bSync` method pauses the thread until a requested or waited-for event is triggered, or until a timeout occurs.

Events in the system are instances of class *Event*. All events have a *type* (an integer), and additional parameters can be added by supplying classes that inherit from *Event*.

In order to run the application, the user instantiates the initial threads inside the `main` method of the program and calls a special `start` method provided by the framework.

Parts of the application that corresponds to the example in Fig. 1 appear in Fig. 10. Additional features of BPC, such as dynamic thread creation and customized event selection strategies, appear as parts of the case-study, described in Section 8. In later code snippets we sometimes omit parts of the C++ syntax and focus on the body of the threads.

```
enum { waterLevelLow, addHot, addCold };

class WhenLowAddHot : public BThread {
   void entryPoint() {
      while( true ) {
         Vector<Event> requested, waited, blocked;
         waited.append(waterLevelLow);
         bSync(requested, waited, blocked, NO_TIMEOUT);

         waited.clear(); requested.append(addHot);
         for( unsigned i = 0; i < 3; ++i )
            bSync(requested, waited, blocked, NO_TIMEOUT);
}}};
```

Figure 10: The events in the program have a *type* field with possible values waterLevelLow, addHot and addCold, and no parameters. The *WhenLowAddHot* class inherits from *BThread*, with the entryPoint method customized to carry out the specific thread behavior. The thread runs in an infinite loop, and in each iteration it waits-for event waterLevelLow and then requests event addHot three times. The bSync method takes three event vectors and a timeout parameter (here, set to the special value NO_TIMEOUT).

In order to facilitate the migration of existing behavioral code, BPC supports programs where the extension idioms that we propose in this work are not used. For instance, synchronization calls may contain just the first 3 parameters, ignoring the timeout parameter; the effect is the same as passing the NO_TIMEOUT value, which has the same semantics as a synchronization call in traditional BP. If no customized event selection strategy is defined, BPC uses a default, arbitrary selection scheme. Naturally, events without parameters and programs with just statically created threads are also allowed.

7.2 The Underlying Mechanism

Communication between threads and the event selection mechanism is performed using standard client-server sockets. Throughout the run, the ESM maintains a server socket which awaits new threads that might connect. For each currently active thread, the ESM maintains an active socket connection on which synchronization data and triggered event information are exchanged. Whenever a thread synchronizes with the ESM, the latter checks if the thread declared a timeout at this synchronization point; if so, it sets a timer to expire accordingly. If an event that the thread requested or waited-for is triggered before the timeout expires, the timer is reset. The pseudocode for the ESM appears in Alg. 1.

Algorithm 1 Event Selection Mechanism

```
1: $ActiveThreads \leftarrow \emptyset$,    $Synchronized \leftarrow \emptyset$
2: while true do
3:   wait for new threads, synchronizations and timeouts
4:   if new thread $bt$ connected then
5:     $ActiveThreads \leftarrow ActiveThreads \cup \{bt\}$
6:   else if timeout for thread $bt$ expired then
7:     $Synchronized \leftarrow Synchronized - \{bt\}$
8:     inform $bt$ of a timeout
9:   else if thread $bt$ synchronized then
10:      set the timeout timer for $bt$
11:      if $ActiveThreads = Synchronized$ then
12:        if exists event $e$ enabled for triggering then
13:          for every thread $bt'$ that requested/waited-for $e$ do
14:            send $e$ to $bt'$
15:            $Synchronized \leftarrow Synchronized - \{bt'\}$
16:            reset timeout timer for $bt'$
```

Line 12 of the algorithm does not specify which event e to choose in case there are multiple enabled events. The default option in BPC is arbitrary event selection. If the programmer has customized the event selection mechanism, he/she has provided an object that can take the list of enabled events and return the next choice, in which case that object is then invoked. The object may also store information from previous iterations and use it in the present selection iteration. An example appears in Section 8.

Another variant of BP that may be useful in a distributed setting includes a distributed version ESM [16]. In this variant, which is also supported in BPC, the threads are partitioned into sets — each of which is managed by a different *ESM agent*. The agents exchange information among themselves when needed. Distributing the ESM can be useful, e.g., when threads run on multiple machines, and communication between these machines is slow or costly.

8. CASE-STUDY: A WEB-SERVER

In this section we survey the architecture of our web-server case-study, dwelling in particular on the implementation of the examples discussed in Sections 3-5. Most of the threads and inter-thread interactions described in this section are displayed in Fig. 11. Apart from giving the technical details, throughout this section we also try to convey to the reader a sense of the interaction between the behavioral code and native C++ code in our implementation, and also of the incremental development process of a behavioral application.

8.1 The Implementation's Layout

Our application consists of two distinct sets of threads, one for the TCP layer and one for the HTTP layer. The two layers interact with each other via behavioral events, and each also has an additional source of input: the TCP layer reads TCP segments off a "raw socket", and the HTTP layer reads files from a given directory. These additional inputs are obtained by threads containing non-trivial native C++ code, and are then translated into behavioral events in order to be passed to other threads.

Internally, each layer is designed using a *dispatcher* architecture: a dispatcher thread handles each incoming segment, classifies it according to its attributes, and then passes it to specific *handler* threads via behavioral events. These handler threads can then request additional events in order to issue a reply and/or update other threads of the contents of the segment. Handlers typically perform local computation using native C++ code — e.g., calculating TCP checksums or reading files from the directory. Incoming TCP segments containing HTTP requests are passed between the layers, and the same happens to HTTP replies on their way to the client.

To exemplify the server's operation we describe in more detail the handling of TCP connection establishment requests (SYN segments). Initially, the *RawSocketReceiver* sensor thread reads the incoming segment from the socket. When it is received, the thread requests a tcpSegmentReceived event — with the segment as its parameter — in order to pass the segment to the *TcpDispatcher* thread.

The *TcpDispatcher* uses native C++ code to classify incoming TCP segments according to their attributes, and requests additional events accordingly. SYN requests, for example, are identified by reading the SYN flag from the TCP header of the segment. The thread then requests a tcpSynRequest event in order to notify *TcpSynHandler* — the specific handler for SYN requests.

The *TcpSynHandler* thread responds to each request by generating a tcpOutgoingSegment event, with a SYN-ACK segment as its parameter. Finally, this event then gets translated into a tcpSendSegment event — handled by the *RawSocketSender* sender thread, which actually sends the segment to the client.

We note that in order to construct the SYN-ACK segment, the *TcpSynHandler* thread must first acquire a fresh sequence number. This is performed by sending a request to, and receiving a response from, the *SequenceNumberAllocator* thread — the thread in charge of managing the sequence numbers of every TCP connection. *SequenceNumberAllocator* may handle simultaneous requests for sequence numbers (for the same connection) from multiple threads, and here the BP event selection mechanism guarantees that each outgoing segment has a fresh sequence number: *SequenceNumberAllocator* handles requests (represented by events) sequentially, and thus race conditions are avoided.

Apart from dispatcher and handler threads, additional

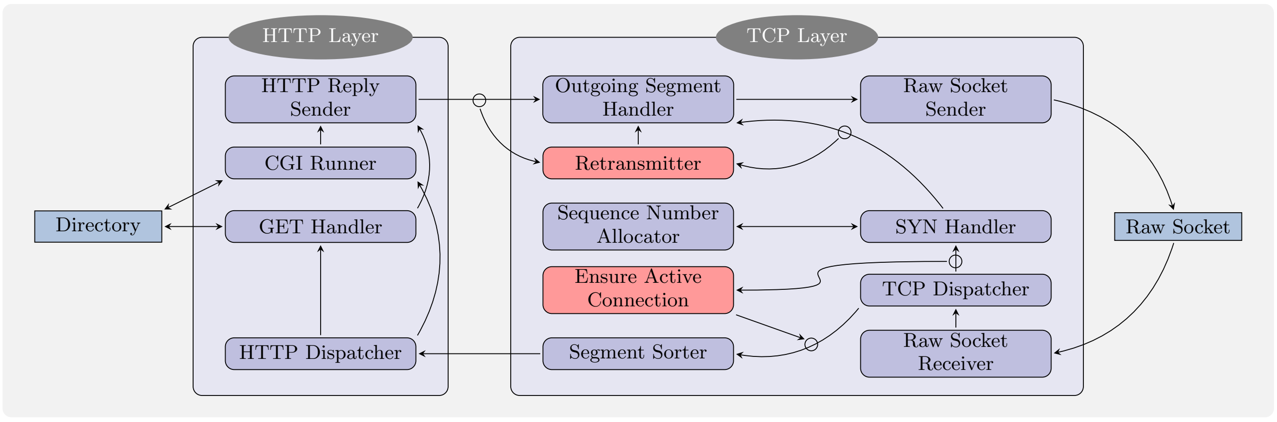

Figure 11: **A overview of the web-server's architecture. For clarity, many details have been omitted; the full code is available online [2]. Rounded rectangles represent threads, and edges represent the logical interactions described throughout Section 8. The threads are partitioned into a TCP layer and a HTTP layer; the layers interact with each other and with additional sources of input ("Raw Socket" and "Directory"). Threads marked in orange were *incrementally* added on top of an already working basis: (1) the *Retransmitter* mechanism that waits for outgoing segments and re-feeds them to *Outgoing Segment Handler* if they are not acknowledged, and (2) the *Ensure Active Connection* mechanism that waits for connection activation/termination events, and blocks HTTP segments on inactive connections from reaching *Segment Sorter*.**

"standalone" threads exist in the system: for instance, the requirement that TCP segments be sent only on active connections is enforced by the *TcpEnsureActiveConnection* thread. This thread uses blocking to ensure that a tcpSynRequest is triggered before other TCP events — such as those signalling PUSH or ACK segments — are triggered. Likewise, once a FIN segment is triggered, the thread blocks any additional TCP events for that connection.

8.1.1 Segment Reordering

TCP segments that contain data for the HTTP layer are not guaranteed to arrive in the order in which they were sent. Hence, the TCP layer needs to reorder them before passing them on.

During data transfer, TCP segments with data for the HTTP layer cause the triggering of dataToHttp events. Each of these events carries the received segment's sequence number as a parameter. The TCP stack knows the expected sequence number of the next data segment: the initial sequence number is stated by the client at the time of connection establishment, and is subsequently incremented for each segment. Whenever an incoming sequence number is greater than the one expected, the stack realizes that a segment is missing; and when this segment later arrives, reordering takes place. Pseudocode for the *SegmentSorter* thread, which is in charge of this reordering, appears in Fig. 12.

8.1.2 Segment Retransmission

As mentioned in Section 3, when sending out a segment, the TCP stack must wait for an acknowledgment message — and if one does not arrive, the segment needs to be resent. There exist several sophisticated retransmission policies, aimed at reducing traffic congestion, which have adjustable retransmission periods. For our case-study we opted for the simplest scheme — retransmission after a fixed waiting period. Implementing additional schemes is left for future work.

In our implementation, segments leaving the TCP stack on their way to be sent to the client via the raw socket

```
while( true ) {                    // SegmentSorter thread
   Vector<Event> requested, waited, blocked;
   waited.append(tcpSynRequest , dataToHttp);
   bSync(requested, waited, blocked, NO_TIMEOUT);

   if ( lastEvent().type() == tcpSynRequest )
      storeSeqNumber(lastEvent().seqNumber());
   else
   if ( lastEvent().seqNumber() != expectedNumber() )
      storeData( lastEvent().data() );
   else sendReorderedSegments( lastEvent().data() );
};
```

Figure 12: **The *SegmentSorter* thread waits for tcpSynRequest and dataToHttp events. When a tcpSynRequest event occurs, the thread extracts the sequence number for later use. When a data segment is received, its sequence number is compared to the expected number. If it does not match, the segment is stored. If it does match, the segment is passed on to the HTTP layer, along with any consecutive segments previously stored, and the expected sequence number is updated.**

always pass as tcpOutgoingSegment events. Our retransmission mechanism waits for these events and stores the outgoing segments. Then, if they are not acknowledged withing a fixed period of time, it retransmits them — until an acknowledgment is received. Pseudocode appears in Fig. 13[1]

8.1.3 Customized Event Selection

As mentioned in Section 4, we occasionally found customizing the event selection strategy a straightforward method in order to enforce certain requirements. For example, in one case we wanted to ensure that all outgoing segments finish sending prior to sending the segment indicating the connection being closed (a FIN segment) — a property

[1]The depicted solution spawns a thread for each outgoing segment, which incurs overhead (as discussed in Section 5). In practice, we found that it was more efficient to spawn one *Retransmitter* thread per connection, and have it handle all of that connection's segments. Nevertheless, we feel Fig. 13 better illustrates the principles described in Section 5.

```
class Retransmitter : public BThread {
   void entryPoint() {
      Vector<Event> requested, waited, blocked;
      waited.append(tcpOutgoingSegment);
      while ( true ) {
         bSync(requested, waited, blocked, NO_TIMEOUT);
         new PeriodicSender(lastEvent());
}}};

class PeriodicSender : public BThread {
   PeriodicSender( Event tcpOutgoingSegment ) {
      storedSegment = tcpOutgoingSegment;
   }

   void entryPoint() {
      bool done = false;
      while ( !done ) {
         Vector<Event> requested, waited, blocked;
         waited.append(ackForStoredSegment());

         bSync(requested, waited, blocked, 2);
         if ( timeoutOnlastSync() ) {
            waited.clear();
            requested.append(storedSegment);
            bSync( requested, waited, blocked, NO_TIMEOUT );
         }
         else done = true;
}}};
```

Figure 13: Pseudocode for the segment retransmission mechanism. The *Retransmitter* thread waits-for tcpOutgoingSegment events — events that indicate a TCP segment about to be sent — and for each such event it spawns an instance of the *PeriodicSender* thread. The *PeriodicSender* instance receives through its constructor the segment that it is supposed to monitor. It then waits for an acknowledgment of that segment for 2 seconds. If an acknowledgment message fails to arrive, the thread retransmits the segment, and the process repeats. When an acknowledgment is received, the thread terminates. Note that the ackForStoredSegment method (code omitted) is a *predicate* — it evaluates to true only for tcpAckReceived events with the proper acknowledgment information. Also, a bookkeeping mechanisms (also omitted) is required to prevent the creation of additional *PeriodicSender* threads for a segment that is being retransmitted.

that was not trivially upheld by the TCP stack. Another example was giving priority to starving connections, namely connections whose events have not been triggered in a while, in order to avoid retransmission of segments and the congestion incurred by it.

Pseudocode for a customized event selection function that addresses these two issues appears in Fig. 14.

8.2 Features and Evaluation

Our implemented TCP stack supports connection establishment and termination, data sending and acknowledgments, keep-alive messages, segment reordering and segment retransmission. Simultaneous connections are also supported, whereas dealing with flow and congestion control is still work in progress. The HTTP stack supports GET requests, error and redirection messages, and the execution of CGI scripts. The project contains over 20k lines of code, and is available online [2].

We constructed our case-study as a proof of concept, and, in our experiments, it provided "smooth" surfing of websites. Nonetheless, it cannot presently compete with industrial web-servers performance-wise, e.g. in throughput rate. We thus focused our evaluation on proper adherence to the

```
Event choose( Vector<Event> enabledEvents ) {
   Vector<Event> candidates =
       eventsOfStarvedConnection(enabledEvents);

   if ( candidates.has(sendTcpFin) &&
        candidates.hasOtherThan(sendTcpFin) )
      return candidates.otherThan(sendTcpFin);
   else return candidates[0];
};
```

Figure 14: Pseudocode for the customized event selection strategy. At every synchronization point, this function is invoked with the set of enabled events, of which it must select one for triggering. Information from previous iterations may be stored. Our specific implementation gives precedence to previously "starved" connections: that is, it favors the connection that has waited the longest for an event to be triggered. This part is abstracted away in the method eventsOfStarvedConnection. Once a connection is selected, its associated events are the candidates for triggering; among these, we prefer events that are not sendTcpFin, so that pending data transmission requests are addressed before the connection is closed. Otherwise, an arbitrary event is selected.

TCP/HTTP protocols.

We tested the system with two widespread browsers, Firefox and Google Chrome. In both of these, the server properly displayed non-trivial sites, with both static and dynamic (e.g., PHP, CGI) pages. In particular, we ran a copy of the BP website [34] on the behaviorally-programmed server.

To test the more advanced features of the server, such as segment retransmission and segment reordering, we conducted tests in which the client connected to the server through a *proxy* — a third piece of software, which we controlled. We then simulated unreliable networks by having the proxy delay or drop segments, or deliver them out of order. All webpages were nevertheless properly displayed.

Finally, for stress testing we ran ten clients that were simultaneously trying to upload a 10 megabyte file each to the server. The proxy was set to maximal interference — that is, not a single segment was delivered to the server in the correct order. All files were successfully received and reassembled by the server.

8.3 Discussion: Incremental Development

An important feature previously attributed to scenario-based programming (and hence BP too) is that it facilitates incremental development [23, 24]. One of our goals was to check whether this would still hold for large programs. While this property is difficult to quantify and may depend on coding habits, the experience we gained when developing our case-study indicates an affirmative answer.

We built our web-server iteratively, repeatedly translating parts of the specification into threads, and with only a vague "big picture" in mind. For instance, we first programmed the connection establishment and data exchange parts of the TCP stack, but without considering segment retransmission or ignoring segments on closed connections. Later, we found that adding these features incurred no changes to existing code (see also Fig. 11). Naturally, in some cases — e.g., adding the reordering of TCP segments — some code was changed, but the changes were usually local and contained.

One could argue that the incremental development of our case-study was made possible because of the *dispatcher-*

handler design pattern that we used. This was indeed partially the case, but we feel that two remarks are in order: (i) BP promoted the use of a dispatcher-handler pattern in the first place; and (ii) in some cases, as in the case of segment retransmission, incremental development was made possible because communication between threads was performed strictly through the triggering of events. Hence, we could easily "hook" onto these events when needed.

9. RELATED WORK

Our proposed extensions to BP are common programming idioms, and exist, in various forms, in numerous high-level languages. Thus, comparisons in this section focus mainly on popular programming formalisms that, similarly to BP, are geared towards discrete event systems.

The principal extension to BP that we proposed is the timeout idiom. This is a step in moving away from the *synchrony hypothesis*, according to which local computation takes negligible time. The synchrony hypothesis is used in popular languages for programming event-driven reactive systems, such as Esterel [10], Lustre [15] and Signal [30], and also in the non-object oriented version of Statecharts [26]. Formally allowing non-negligible computation time broadens the scope of problems to which BP can be applied.

Other parallel programming languages support various idioms for manipulating time. *UML Sequence Diagrams* [4] support constructs that impose a required delay between the send and receive events of a message [11]. *Message Sequence Charts* (*MSC*s) support timers [7,29] that can be set, reset and checked for timeouts, and delay intervals [7,33] to specify maximal or minimal delay between actions. We have demonstrated that similar constraints can be applied using our timeout mechanism. Of particular interest is the *live sequence charts* (*LSC*s) language [12,21], the precursor to and main motivation for behavioral programming. In this context, the ongoing evolution of BP is, perhaps unsurprisingly, similar to the one LSCs underwent with the addition of time-aware charts [22], allowing one to bound the flow of time between consecutive events. Similar concepts appear also in component based programming languages, such as *BIP* [9]. In BIP, components may contain *timed variables*, and may use them transition guards. These variables are globally incremented when no higher priority action is enabled.

The second extension that we proposed was customizable event selection strategies. Scenario-based programs typically have to choose between several enabled events, and several selection strategies have been previously proposed. Among these are arbitrary selection, look-ahead algorithms (*smart play-out*) [18,19] and planning-based approaches [27]. Selection can also be interactive, which is useful, for instance, in the context of debugging unrealizable behavioral specifications [31]. By allowing general user-specified event selection strategies, these approaches could be more readily integrated into BP.

Another extension that proposed was to consider threads as templates of which multiple copies may be instantiated. This technique bears resemblance to the situation in LSCs, where multiple instances of the same chart may run simultaneously. There, additional copies are spawned based on preconditions defined in each chart, instead of actively by other threads as in our case, although the two approaches seem equivalent. The direct spawning of modules by other modules is also supported in Esterel and Signal.

Finally, we proposed to extend BP with parameterized events. Parameterized message passing between modules is quite fundamental in concurrent programming. It exists, e.g., in Esterel, UML sequence diagrams, LSCs and, for bounded parameters, also in earlier work on BP [20].

10. CONCLUSION AND FUTURE WORK

In this paper we set out to study the applicability of the BP paradigm to real-world systems. Through our work on a large case-study, we were able to identify several common programming tasks for which BP's traditional idioms provide only partial solutions, and proposed additional idioms to overcome these difficulties while — trying to maintain BP's simple and intuitive interfaces. The new idioms include time-aware threads, a customizable event selection mechanism, the dynamic creation of threads, and parameterized events. By integrating these idioms into our development environment we were able to complete our implementation, thus providing what we feel is significant evidence that BP does indeed scale-up to real-world problems.

In choosing our proposed extension to BP, we took care to only add idioms that allowed us to accomplish programming tasks that were previously beyond the scope, or at least very difficult to accomplish, in BP. Thus, we hope we were able to avoid clutter, and retain most of the simplicity that characterizes the traditional BP framework.

Our proposed extensions were driven by the needs that arose during the development of our specific case-study; and hence it is possible that, through the development of additional behavioral projects, BP may need to be extended with additional idioms. However, due to the large variety of programming tasks entailed by the web-server project (e.g., handling timeouts, string manipulation, file access, checksum calculations, etc), we believe that our proposed extensions are robust, and could prove sufficient for a variety of programming tasks. We regard our extensions, and also future extensions to BP, a part of the typical evolution of programming languages.

In the future, we plan to enhance our case-study by adding features like flow control, congestion control, selective acknowledgments and smart retransmission schemes to our protocol stacks. These extra features may reveal additional idioms worth adding to BP. Further, we plan to work on improving the efficiency of our case-study, in order to gain a better understanding of the overhead the BP infrastructure might incur in large systems.

Another direction we hope to pursue regards the analysis of behavioral programs. Previous work has shown that BP's structure makes it amenable to program verification and analysis techniques, such as model-checking [20], program repair [17] and eager execution [16]. We plan to work on extending these techniques to support the newly proposed idioms, and in particular the concept of timeouts.

11. ACKNOWLEDGMENTS

We thank Assaf Marron and Gera Weiss for their helpful comments on this work. This work was supported by an Advanced Research Grant to DH from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013), and by an Israel Science Foundation grant.

12. REFERENCES

[1] BPC: Behavioral Programming in $C++$. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/`.

[2] Case-study: a BPC web-server. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/webserver.zip`.

[3] Supplementary material. `https://sites.google.com/site/guykatzhomepage/AGERE2014_Supplementary.pdf`.

[4] The UML Standard. `http://www.uml.org/`.

[5] G. A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[6] G. Alexandron, M. Armoni, M. Gordon, and D. Hagen. Scenario-Based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking. In *Proc. 36th Int. Conf. on Software Engineering (ICSE)*, pages 311–320, 2014.

[7] R. Alur, G. Holzmann, and D. Peled. An Analyzer for Message Sequence Charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[8] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG.* Prentice Hall, 1993.

[9] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In *Proc. 4th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.

[10] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[11] G. Booch, J. Rumbaugh, and I. Jacobson. Unified Modeling Language for Object-Oriented Development (Version 0.91 Addendum). RATIONAL Software Corporation, 1996.

[12] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[13] N. Eitan and D. Harel. Adaptive behavioral programming. In *23rd IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 685–692, 2011.

[14] M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming. In *Proc. 17th Conf. on Innovation and Technology in Computer Science Education (ITICSE)*, pages 198–203, 2012.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data-Flow Programming Language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, 1991.

[16] D. Harel, A. Kantor, and G. Katz. Relaxing Synchronization Constraints in Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 355–372, 2013.

[17] D. Harel, G. Katz, A. Marron, and G. Weiss. Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 3–12, 2012.

[18] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 378–398, 2002.

[19] D. Harel, H. Kugler, and A. Pnueli. Smart Play-Out Extended: Time and Forbidden Elements. In *Proc. 4th Int. Conf. on Quality Software (QSIC)*, pages 2–10, 2004.

[20] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. In *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, pages 279–288, 2011.

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

[22] D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc. 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006.

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

[24] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90–100, 2012.

[25] D. Harel, A. Marron, G. Weiss, and G. Wiener. Behavioral Programming, Decentralized Control, and Multiple Time Scales. In *Proc. 1st SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, pages 171–182, 2011.

[26] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* McGraw-Hill, 1998.

[27] D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 485–499, 2007.

[28] T. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Proc. 14th Int. Symp. on Formal Methods (FM)*, pages 1–15, 2006.

[29] ITU. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC), 1996.

[30] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[31] S. Maoz and Y. Sa'ar. Counter Play-Out: Executing Unrealizable Scenario-Based Specifications. In *Proc. 35th Int. Conf. on Software Engineering (ICSE)*, pages 242–251, 2013.

[32] A. Marron, G. Weiss, and G. Wiener. A Decentralized Approach for Programming Interactive Applications with JavaScript and Blockly. In *Proc. 2nd SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, pages 59–70, 2012.

[33] N. Meng-Siew. Reasoning with Timing Constraints in Message Sequence Charts. Master's thesis, University of Stirling, 1993.

[34] The Behavioral Programming Webpage. `http://www.b-prog.org/`.