Coordinating and Visualizing Independent Behaviors in Erlang

Guy Wiener Gera Weiss
Ben-Gurion University, Beer-Sheva, Israel
{gwiener,geraw}@cs.bgu.ac.il

Assaf Marron
Weizmann Institute of Science, Rehovot, Israel
assaf.marron@weizmann.ac.il

Abstract

Behavioral programming, introduced by the LSC language and extended by the BPJ Java library, enables development of behaviors as independent modules that are relatively oblivious of each other, yet are integrated at run-time yielding cohesive system behavior. In this paper we present a proof-of-concept for infrastructure and a design pattern that enable development of such behavioral programs in Erlang. Each behavior scenario, called a behavior thread, or b-thread, runs in its own Erlang process. Runs of programs are sequences of events that result from three kinds of b-thread actions: *requesting* that events be considered for triggering, *waiting* for triggered events, and *blocking* events that may be requested by other b-threads. A central mechanism handles these requests, and coordinates b-thread execution, yielding composite, integrated system behavior. We also introduce a visualization tool for Erlang programs written in the proposed design pattern. We believe that enabling the modular incremental development of behavioral programming in Erlang could further simplify the development and maintenance of applications consisting of concurrent independent behaviors.

Categories and Subject Descriptors D.2.11 [*Software*]: Software Architecture—Patterns; D.3.2 [*Programming Languages*]: Language Classifications—Erlang

General Terms Design, Languages

Keywords Design Patterns, Behavioral Programming, Live Sequence Charts

1. Introduction

Scenario-based programming, or behavioral programming, is a programming paradigm introduced by the language of Live Sequence Charts (LSC) [1] and its Play-Engine implementation [3]. This work was extended in [6] through the BPJ library that implements behavioral principles in a traditional Java programming context. In behavioral programming, behaviors are programmed relatively independently of each other, and are interlaced at run-time to create a cohesive, integrated, system behavior. This approach turns out to be very natural, and enables incremental development of highly modular system, where the decomposition of the system is according to behaviors - software components that may cross subsystem boundaries and are not necessarily tied to a specific class or object. LSC and BPJ represent two different approaches for behavioral programming. LSC is based on centralized execution of a collection of sequence charts [7, 9] enhanced with modalities that control what must, may or must not be done. The Play Engine examines all charts in the specification, and triggers events such that the execution satisfies the modal specification. The LSC language also includes rich programming constructs such as objects with properties, control flow, conditions, variables, symbolic objects and implemented functions that expand the capabilities of the developed applications. Behavioral programming in BPJ is based on running behaviors in Java threads (behavior threads, or b-threads, for short) that call API functions to announce events that they request, wait for, or block, and to invoke a coordination mechanism that weaves these requests yielding integrated system behavior.

In both approaches, candidate next events from each behavior are considered for triggering. One of these is selected, subject to the condition that it is not forbidden, or blocked, by other behaviors. Behaviors affected by the triggered event advance and perform arbitrary processing. All behaviors then are synchronized and coordinated, resulting in selection of next event in the system.

In Section 8 we outline in some more detail the different approaches of both LSC and BPJ.

In this paper we adopt the approach used by BPJ and propose (through a proof-of-concept) implementing b-threads as Erlang processes. We provide a central coordination mechanism, and an interface that b-threads can use to report the events that they request, wait for, or block. We propose a design pattern for coding behavioral programs, and provide a visualization tool that depicts b-threads coded in the proposed pattern as transition systems.

We believe that enabling the modular incremental development of behavioral programming in Erlang could further simplify the development and maintenance of applications consisting of concurrent independent behaviors.

The sections of this paper follow largely the section structure used by "The Gang of Four" [2] for documenting design patterns, including sections such as intent, motivation, applicability, structure, sample code, and related patterns. The visual tools is described in the Structure section.

2. Intent

We propose a design pattern called BP and an associated module (called bp), for iteratively creating a sequence of events, where the next event is chosen with the help of the bidding protocol described below. The bidders are Erlang processes registered as behavior threads (b-threads). In each iteration:

1. Each b-thread places a bid:
   - Watched events: events that the b-thread waits for and asks to be notified of.
   - Requested events: events that the b-thread proposes that they be considered for triggering.
   - Blocked events: event that the b-thread forbids.

Ninth ACM SIGPLAN Erlang Workshop Sept. 30, 2010, Baltimore.
Copyright © 2010 ACM [to be supplied]. . . $10.00

2. When all b-threads place their bids, an auction takes place:
   - An evaluation mechanism chooses an event requested by some b-thread and not blocked by any b-thread.
3. B-threads are notified of the auction outcome:
   - The b-threads that asked to be notified (the selected event is in their watched set) are resumed.
4. B-threads can execute arbitrary computations before placing their next bid in the next iteration.

3. Motivation

The motivation for proposing the design pattern is to provide a simple mechanism through which systems can be constructed from software components each of which controls and coordinates a particular behavior. As behaviors may cross object boundaries, such construction complements the traditional approach to software development where programs modularity revolves around objects and data-structures. Using events as markers of system behavior, and applying the proposed bidding mechanism for choosing events, the resulting integrated system behavior is an event sequence that reflects, at every step, each b-thread's view of how the system should proceed.

The BP design pattern helps programmers maintain b-thread independence, by unifying the occurrence of events requested by multiple b-threads and not notifying b-threads of events that they do not watch. Particularly, requesting scenarios need not care how events that they trigger affect other scenarios.

4. Applicability

The BP design pattern should be used when the system's behavior can be naturally decomposed into, or described as an interleaved execution of relatively independent scenarios as described in [1, 3]. BP allows for programming each of these scenarios in an explicit and natural way in its own module, as an alternative to allowing the scenario to emerge implicitly from code that is scattered in multiple participating objects.

In this context, it is worth noting that interesting behaviors often emerge already in early stages of software design and specification, even with small sets of b-threads. Thus, BP allows for programming initial designs and software specifications to be presented to users for feedback (e.g., for finding errors in the way requirements are understood).

More generally, the BP design pattern is suitable for incremental development, as it allows adding and removing behaviors with little or no change to existing code.

Another context where BP can be particularly applicable is end-user customization. For example, imagine a (remote) control for a video player. If that player's behavior is programmed in BP, users can customize it without going into existing code. They can add or remove behaviors, say for simplifying certain activities, or for avoiding common mistakes, by adding to the system b-threads that handle specific sequences of user actions.

5. Structure

5.1 Participants

In this design, the participants are:

The BP controller: A central server process.

- Receives the synchronization requests from b-threads.
- Decides on the next event.
- Sends the next event to b-threads that are waiting for it.

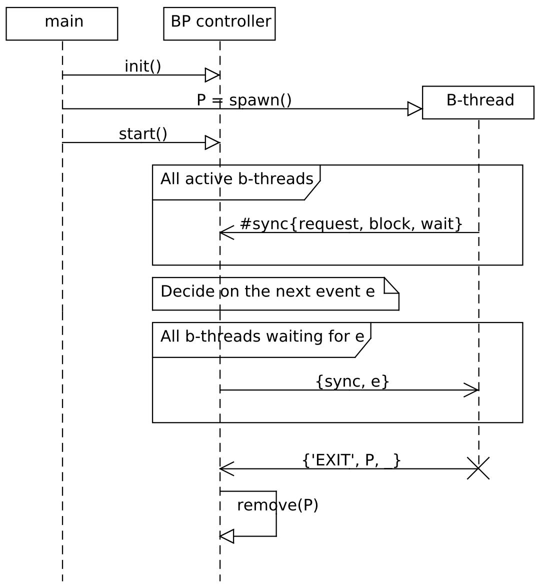


Figure 1. The collaboration between processes using BP. Filled arrows mark function calls, vee arrows mark messages.

b-threads: The work-doing processes.

- Bid for the next event by sending a synchronization request to the BP controller with the following parameters: (1) Requested events. (2) Blocked events. (3) Watched events.
- Wait until the BP controller sends them an event that they are waiting for.

5.2 Collaboration

The collaboration between the BP controller and the b-threads follow these steps (see Figure 1):

1. The BP controller is initialized.
2. B-thread processes are spawned and added to the controller.
3. The BP controller is started.
4. B-threads call `bp:sync` to send their requests to the BP controller, and are suspended until they receive a response from the controller.
5. The BP controller waits until all the active registered b-threads send their requests.
6. The BP controller decides on the next event (see the next section) and sends responses to the b-threads that wait for it.
7. The b-threads that receive the event continue their computation until they call `bp:sync` again.
8. When a b-thread exits, it is removed from the BP controller.

5.3 Model

The BP design pattern is based on the following mathematical model [6]:

First, the code of each b-thread is abstracted as a transition system whose states are valuations of program variables at the times where the b-thread places its bids (synchronizes with the other b-threads). To model the bidding, we attach to each state s a set $R(s)$ of requested events and a set $B(s)$ of blocked events, as formalized in Definition 1. Recall that a (labeled) transition system is defined (see e.g [8]) 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.

The runs of such a transition system are sequences of the form $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_i} s_i \cdots$, where $s_0 = init$, and for all $i = 1, 2, \cdots, s_i \in S, e_i \in E$, and $s_{i-1} \xrightarrow{e_i} s_i$.

Definition 1 (behavior thread)**.** A b-thread is a tuple $\langle S, E, \rightarrow, init, R, B\rangle$, where $\langle S, E, \rightarrow, init\rangle$ forms a labeled transition system, $R\colon S \rightarrow 2^E$ is a function that associates each state with the set of events requested by the b-thread when in that state, and $B\colon S \rightarrow 2^E$ is a function that associates each state with the set of events blocked by the b-thread when in that state.

Using this abstraction, we can formalize the auction mechanism as an operator for composing transition systems, as given in Definition 2.

Definition 2 (runs of a set of b-threads)**.** We define the runs of a set of b-threads $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i\rangle\}_{i=1}^n$ as the runs of the labeled transition system $\langle S, E, \rightarrow, init\rangle$, where $S = S_1 \times \cdots \times S_n$, $E = \bigcup_{i=1}^n E_i$, $init = \langle init_1, \ldots, init_n\rangle$, and $\rightarrow$ includes a transition $\langle s_1, \ldots, s_n\rangle \xrightarrow{e} \langle s_1', \ldots, s_n'\rangle$ if and only if

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

and

$$\bigwedge_{i=1}^{n} \Big(\underbrace{(e \in E_i \implies s_i \xrightarrow{e}_i s_i')}_{\text{affected b-threads move}} \wedge \underbrace{(e \notin E_i \implies s_i = s_i')}_{\text{unaffected b-threads don't move}} \Big) \quad (2)$$

This definition specifies a transition system whose runs are the interleaved executions of the composed system. Specifically, we say that a sequence of events is a run of the system if, at each step, the selected event is requested by some b-thread and not blocked by any. The second part of the definition says that the selected event may change the state of the b-threads that have it in their alphabet.

Note that this mathematical model allows for nondeterminism. In particular, if there are multiple requested events that are not blocked the model allows choosing any of them. In implementations, however, it is easier to program deterministic systems. In particular, we prioritize the b-threads and event requests such that event selection is deterministic.

5.4 Code Structure

Our implementation of the BP design pattern is based on a supporting module, `bp`. The `bp` module exports the following functions:

`init/0` Initialize the BP controller.

`add/2` Add a process with a given priority.

`start/0` Start the controller.

`sync/1` Send a synchronization request, that includes requested, blocked and watched events to the controller and wait until one of the watched events is selected. The `sync/1` function takes a record as an argument. The record definition is `-record(sync, {request, wait, block})`. The record fields match the arguments described in Section 5.3, and have a default value of an empty list. For example:

```
bp:sync(#sync{
        request=[E1],wait=[E1,E2],block=[E3]})
```

`remove/1` Remove a process from the controller.

For its basic operation, the bp module imposes a few constraints on the structure of the code. B-threads must be spawned as processes, and the program must include initialization code for the `bp` process. Section 5.2 describe the initialization sequence in more detail. There are no other constraints on the calculations that b-threads perform before or after the calls to `bp:sync`.

```
morning() ->
  [sync(#sync{wait=[morning], request=[morning]}) ||
    _ <- lists:seq(1,3)].

evening() ->
  [sync(#sync{wait=[evening], request=[evening]}) ||
    _ <- lists:seq(1,3)].

interleave() ->
  bp:sync(#sync{wait=[morning], block=[evening]}),
  bp:sync(#sync{wait=[evening], block=[morning]}),
  interleave().

display() ->
  Event = bp:sync(#sync{wait=[morning, evening]}),
  io:format("Good ~w~n", [Event]),
  display().

test() ->
  bp:init(),
  bp:add(spawn(fun morning/0), 1),
  bp:add(spawn(fun display/0), 2),
  bp:add(spawn(fun evening/0), 3),
  bp:add(spawn(fun interleave/0), 4),
  bp:start().
```

Figure 2. "Hello, World!" - An example of using the bp module

```
Good morning
Good evening
Good morning
Good evening
Good morning
Good evening
```

Figure 3. Output of code from Figure 2

Figure 2 shows a basic example of code that follows the classical "'Hello, World!" program and uses the bp module to issue "Good morning" and "Good evening" greetings. The example includes four b-threads. The events in the system are `morning` and `evening`. The `morning` and `evening` b-threads request one of these events three times and terminate. The `interleave` b-thread, in an alternating manner, blocks one of these events while waiting for the other, causing the interleaving of the two independent event sequences. The `display` b-thread prints the selected event. The expected result — alternating `Good morning` and `Good evening` greetings — is shown on Figure 3.

It should be noted that at the current step of the development, `bp` is not a generic module: It does not take another module as an argument and does not define a behavior for another module. Instead, it allows for starting b-threads by spawning a process. We find this approach more flexible and free-form. However, if the code follows specific conventions, listed in the next section, an auxiliary visualization module can produce a diagram of it.

5.4.1 Code Visualization

The `bp` module supports any unstructured code, as long as it uses `init`, `add`, `start` and `sync`. However, if the code has a specific structure, it is possible to produce a visual representation of its behavior automatically: A module named `bp_vis` produces a diagram of a transition system depicting the b-thread. The diagram

```
case bp:sync(#sync{...}) of
  x -> state1();
  y -> state2();
  ...
end.
```

Figure 4. An example of a state case

is generated as a Graphviz[1] file. There are two ways to generate the diagram: Use `bp_vis` as a `parse_transform` module, or run `bp_vis:visualize(BeamFile, GvFile)` directly. In the first case, the generated file has the same base name as the module.

The code visualizer makes the following assumptions about the structure of the code:

1. The code for each b-thread is contained in a separate module.
2. Each function that is called in a clause of a state case is a state function, that represents the next state where:
   - A *state function* is a function that its last term is a *state case*.
   - A *state case* is a case statement where the expression is a call to `bp:sync` and each clause maps from an event to a function call. Figure 4 shows an example of a state case.
3. If a function called `start` is found in the module, it is assumed to be the first state.

The format of the generated diagram follows.

- State functions appears as ellipses with multiline labels. The first line is the function signature. The following lines show the content of the request, wait and block arguments of the call to `bp:sync` in the state case.
- An edge from ellipse A to ellipse B appears if one of the clauses in the state case of A is a function call to B.
- Each edge has a label. The label format for an edge from A to B is "`Event (when Guard) / Call`". The `Event` is the head of the case clause whose body is the function call to B. The `Guard` is the guard part of the head of the clause, which is optional. The `call` is the exact function call to B, including arguments.
- If one of the state functions is called "`start`", it is emphasized by a small arrow, starting from a black dot.

For example, suppose that some device can print, scan, send a fax and stop. The b-thread module in Figure 5 has the following behavior: It tries to print 3 times. If the printing starts, it waits for it to end. If it fails more then 3 times, it gives up. When waiting for printing, it prevents the machine from stopping. Figure 6 shows a visualization of that code. Since this b-thread does not have a designated start function, it does not include the additional arrow. The transition diagram figures in Section 7.2 were generated using this code visualization tool, while designating a start function.

In the future we expect that the structural requirements that simplify the visualization process will be expanded to include other design patterns, or accommodate less constrained code. It is our view that the code for behavioral program should be as free as possible, and need not be aligned with the state transitions used for the formal definitions or visualization. In Section 8 we discuss the relation of our work to state transition coding with `gen_fsm`, using callback-functions as well as explicit state.

[1] `http://graphviz.org`

```
-module(printjob).
-compile([{parse_transform, bp_vis}]).
-include("bp.hrl").
-define(LIMIT, 3).

pending(N) ->
  case bp:sync(#sync{request=[print],
             wait=[print, scan, fax],
             block=[stop]}) of
    print -> working();
    _ when N < ?LIMIT -> pending(N+1);
    _ when N >= ?LIMIT -> idle()
  end.

working() ->
  case bp:sync(#sync{wait=[finish],
                     block=[stop]}) of
    finish -> idle()
  end.

idle() ->
  case bp:sync(#sync{wait=[stop]}) of
    stop -> ok
  end.
```

Figure 5. A b-thread module with a structured code

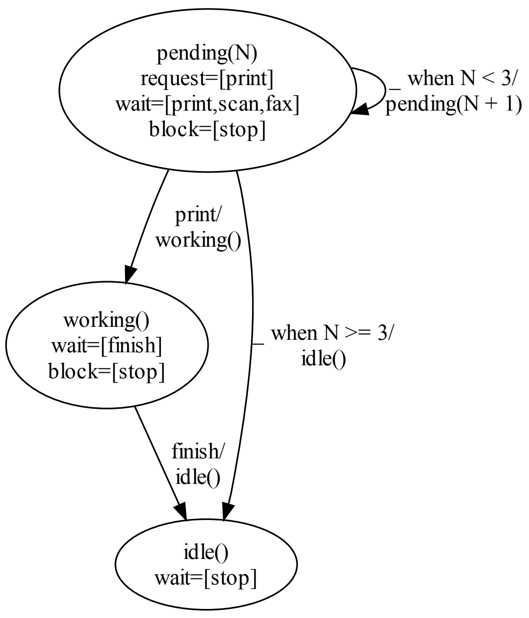


Figure 6. A visualization of the module in Figure 5

6. Consequences

Behavioral programming is based on consensus: It requires all participating processes to agree on the next step. Therefore, it requires a synchronization point for all b-threads that are not waiting. To exit the blocking call to `bp:sync`, each active b-thread must wait until all other active b-thread call `bp:sync`. Although this is a reasonable demand for an agreement protocol, a specialized protocol for a specific problem can be more efficient.

Reaching an agreement also requires sharing information between processes. In our implementation, we use a central process (the BP controller) to collect and handle the requests from all participating b-threads. Therefore, this process is critical to the operation of the system. In a production system, special attention should be given to this potential single-point-of-failure, and common actions such as monitoring, automatic recovery and redundancy should be considered.

In the proof-of-concept implementation, the execution rate can reach thousands of events per second, with thousands of participating b-threads. Detailed performance analysis remains as future work. Such analysis should take into consideration, among others, the amount of event-based synchronization as compared with other processing and computation performed by the b-threads. For applications where synchronization may introduce exececssive delays we are considering for example clustering of b-threads in synchronization groups, or using design patterns in which long-running processes do not run as b-threads, but instead use dynamically-created b-threads to communicate with the rest of the behavioral program. Lastly it should be noted, that certain benefits of programming multiple independendent b-threads may be manifested also when the number of processors is small, or even one, as complex behaviors are decomposed into simpler ones in a natural way.

7. Examples and Sample Code

7.1 Coordinated Sequential Processing

To illustrate how the BP design pattern can be used, we discuss, as an example, the structure of applications that require bulk processing of a large volumes of records to perform business operations. Examples of business operations include time based events (e.g. generation of periodic correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g. interest accrual or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Such batch processing systems are used to process billions of transactions every day for enterprises around the world. Easy interweaving of such processes can be of great value. For example - consider the printing of different notices as well as paper advertisements and coupons for insertion in each customer's envelope. Sequential processes may independently customize individual messages to customers in a large database, but they need to be coordinated, such that all messages to a given customer are printed consecutively. The BP design pattern enables such coordination with minimal dependency across the different sequential processes.

In BP terms, sequential batch processing can be formulated as an iterative bidding/consensus process where, for each record of data, a set of independent b-threads collaborate by expressing their views of how the record should be processed (using the request/wait/block idioms). More specifically, the system can be programmed using a sequencer b-thread that controls the sequencing of the records and a set of b-threads that model different considerations of how the records should be processed.

To demonstrate the technique, we present an implementation of the Sieve of Eratosthenes algorithm in Figure 7. The `sequencer` is a b-thread that leads the sequential processing of the natural numbers, and attempts to declare each one as a prime. The `pFactors` b-thread blocks the multiples of a prime number from being declared as prime. The `factory` b-thread is responsible for spawning and registering a `pFactors` b-thread whenever a prime number is discovered. Note that such dynamic addition of b-threads is an extension of the basic collaboration described above, and requires further attention in definition and development. The `start` method starts an instance of the `sequencer` and the `factory` b-threads. This code does not conform with the assumptions outlined in Section 5.4.1 above and therefore cannot be automatically visualized by `bs_wis`.

```
sequencer(I) when I < 100 ->
  sync(#sync{wait=[I], request=[I]}),
  T = sync(#sync{
    wait=[prime,not_prime],
      request=[prime,not_prime]}),
  io:format("~w is ~w ~n", [I,T]),
  sequencer(I+1);

sequencer(I) -> io:format("---~n").

pFactors(I) -> pFactors(2*I,I).

pFactors(N,I) ->
  sync(#sync{wait = [N]}),
  sync(#sync{block=[prime], wait = [N+1]}),
  pFactors(N+I,I).


factory(I) ->
  I = sync(#sync{wait = [I]}),
  T = sync(#sync{wait = [prime,not_prime]}),
  if
    T == prime ->
       add(spawn(fun() -> pFactors(I) end), 1);
    true -> ok
  end,
  factory(I+1).

run() ->
  init(),
  add(spawn(fun() -> sequencer(2) end), 3),
  add(spawn(fun() -> factory(2) end), 4),
  start().
```

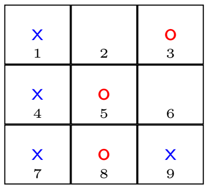
Figure 7. Illustration of coordinated sequential batch processing: BP version of the Sieve of Eratosthenes.

7.2 Tic-Tac-Toe

As another (larger) example of a code that uses the proposed BP design pattern, we describe an implementation of a computer program that plays the well known game of Tic-Tac-Toe. Game playing behavior by humans is naturally decomposed into independent behaviors of complying with the rules, random or default moves, and an attempt to apply various tactics accumulated with experience. Thus, the purpose of this example is to show how the BP pattern can be used to construct a composite behavior from a set of simpler, intuitive b-threads.

The game involves two players: one marked x is played by the human, and a second marked o is played by the computer. The events in the program are pairs of the form $\langle \mathsf{x}, Sq\rangle$ or $\langle \mathsf{o}, Sq\rangle$, where Sq references a square in the 3×3 board, and is an integer between 1 and 9 (see figure below).

A game is played as a sequence of events; e.g the sequence $\langle \mathsf{x}, 1\rangle$, $\langle \mathsf{o}, 5\rangle$, $\langle \mathsf{x}, 9\rangle$, $\langle \mathsf{o}, 3\rangle$,$\langle \mathsf{x}, 7\rangle$, $\langle \mathsf{o}, 8\rangle$, $\langle \mathsf{x}, 4\rangle$, describes a game round in which x wins, and its final configuration is:



Below we list the b-threads of the program and use the visualization (automatically generated from the Erlang code) to explain their behavior.

detect_win: This b-thread detects the occurrence of winning scenarios. Independently of how the game is played, the basic rule of the game that says "the first player to get three in a line wins" can be directly translated to a simple b-thread as shown in Figure 8.

enforce_turns: Another rule for this game is that "players alternate placing xs and os on the board". This, again, translates directly to a simple b-thread as depicted in Figure 9.

disallow_square_reuse: This b-thread prevents a given square from being marked twice. See Figure 10.

default_moves: This b-thread simply requests the marking of all squares. The order of the requested events determines their priorities in our strategy: try to mark the center square first, then the corners, and only then the remaining squares. Requested moves will be triggered only when not blocked and when there are no higher priority unblocked requests. With the addition of this b-thread, the program can now play legally and can complete any game - though its strategy is quite simplistic.

prevent_line_with_two: One of the first rules of thumb, taught to someone learning how to play the game, is that when your opponent (in our case the x player) is about to complete a line of three, you should put your mark (in our case o) on that line to preempt the attack. This rule can be directly translated to a b-thread, as shown in Figure 12.

complete_line_with_two: This b-thread implements another basic rule of thumb of tic-tac-toe: whenever you can complete a (vertical, horizontal, or diagonal) line and, by that, win the game – do it. Note that, since the winning is immediate, the attack can have a higher priority than defense. In particular, when `prevent_lines_with_two` requests to preempt an attack and `complete_line_with_two` requests the winning move – the latter request should be chosen. This is implemented in the BP design pattern using the priority mechanism – events requested by a b-thread with higher priority are chosen over requests of b-threads with lower priorities. The `complete_line_with_two` b-thread is depicted in Figure 13.

intercept_ single_ fork: This multi-instance b-thread defends against situations where a future marking by player x will present him/her with the choice of winning in one of two different lines. For example, following a mark of square 6 and square 8 by x, player o will try to mark square 9. See Figure 14.

intercept_ double_ fork: This b-thread defends against situations where a future marking by player x will present him/her with the choice of creating two forks as described above by marking two opposite corners when o marks the center. The defense used by this b-thread is to attack, by marking square 2 and forcing the opponent to defend and abandon his own attack. See figure 15.

This set of b-threads constitutes a complete computer program that plays tic-tac-toe against a user. The strategy for the o player, played by the computer, emerging from the composition of the b-threads is optimal in the sense that the o player will never lose the game (the x player, played by the user, can force a tie but can never win). In addition to the b-threads, the code includes one additional module, not listed here, containing initialization code that spawns and registers copies of the b-threads.

8. Known Uses and Related Work

The event-based and state-like nature of BP makes b-threads somewhat similar to the gereric finite state machine (`gen_fsm`) module[2] from the Erlang standard library. Both the `gen_fsm` and `bp` modules deal with a state-based reaction to events. However, there are several differences between the two modules:

- `gen_fsm` does not provide the option to block events.
- `gen_fsm` does not deal with coordinating between several instances.
- `gen_fsm` does not distiguish between events that are waited for and other events. It will handle any call to `gen_fsm:send_event`.

One can view the `bp` module as an extension of `gen_fsm`, designed for coordinating between several processes.

In addition to the general capabilities and broad usage of Erlang in concurrent processing, particular attention to programming independent behaviors in Erlang can be seen in systems such the ERES rule-production system [11] or the eXat agent programming system [10]. What distinguishes b-thread synchronization in the proposed pattern from the classical programming of concurrent behaviors in Erlang is the ability of one process to prevent the occurrence of an event requested by another process, without each party's explicit awareness of the existence of the other party. It will be interesting to explore the addition of the proposed synchronization approach with its compact blocking idiom into the above systems.

The proposed pattern and module for the Erlang language follows in the footsteps of scenario based programming and behavioral programming of LSCs [1, 3]and BPJ[6].

The formal visual language of live sequence charts (LSCs) was defined in [1]. The LSC language extended message sequence charts (MSC)[7], and the then-current UML sequence diagrams mainly by adding modalities to events (UML sequence diagrams were later enhanced to express some of these notions [9]). The LSC language adds to the sequence diagram a notation that distinguishes between events that must happen ("hot"), events that may happen ("cold"), and events that must not happen (marked explicitly or implicitly as forbidden). These modalities enable the direct execution of LSC specification where the Play Engine tool processes the LSC specification and generates a sequence of events that satisfies the specification. The Play Engine does this by keeping track of the next candidate events on each of the charts and selecting a next event to be triggered based on the specified modalities. If no event can be selected without violating the specification - the system stops. Together with other constructs (objects and properties, flow control, variables, access to functions in other languages, symbolic messages, symbolic objects, and a notion of time), the LSC language and the Play Engine show that independent units of behavior description can be used not only in requirements and specifications, but in building the final executable.

The same concepts were implemented in Java through BPJ library [5, 6] (and can be similarly implemented in other textual, procedural languages). Each behavior is coded in its own Java thread - called a b-thread. The b-thread calls the behavioral synchronization function of the BPJ library (called bSync), passing to it three parameters - a set of requested events, a set of watched events and

[2] `http://www.erlang.org/doc/man/gen_fsm.html`

a set of blocked events. The calls to BPJ also synchronize all b-threads, by suspending each caller until all registered b-threads post their wishes. Then, a central coordination mechanism selects an event that is requested by some b-thread and is not blocked by any b-thread, and resumes all b-threads that either requested the same event, or announced it as a watched event. In BPJ the b-thread relies on the underlying language for flow control, variables, objects and other programming necessities. The library and source code examples are available at [5].

9. Conclusion and Future Directions

A proof-of-concept is described for a design pattern and a supporting module for composing an application from a set of behavior threads that independently request, block, and wait for events. We believe that this design pattern can create valuable synergy between behavioral (or scenario-based) programming and functional programming in Erlang.

Among the useful features of behavioral programming are:

- Behavioral Modularity: B-threads can be coded relatively independent of each other – interacting mainly through events that are part of overall system behavior. The behavior of multiple b-threads is successfully interwoven, even though each b-thread has little or no awareness of the identity of the other b-threads or of their internal structure.
- Incremental Development: New modules that add or restrict behavior can be added to an existing system with little or no change to existing modules. The new module relates to the behavior and the events of the existing system, and not to its structure and code. For example, applications written in this behavioral approach can be more readily patched to correct errors or handle small changes in requirements. The patch, or the new module, can watch out for the event sequences whose handling should be changed, and override the existing behavior with new behavior.

  The incrementality of behavioral programming allows for observing meaningful behavior from early stages of development. As each b-thread generates observable system behavior, incomplete versions of the systems can be used to start validating or refining requirements and specifications.

  An additional aspect of incremental development can be post-deployment system customization, where an end-user can modify the system behavior by adding b-threads for simplifying certain user tasks.
- Naturalness: In natural language conversations and in requirements documents, people often describe behaviors of systems in terms of scenarios. Therefore, behavioral programming seems like a natural approach to development of software systems.

  Additionally, due in part to the behavioral modularity feature, behavioral programs can more readily “explain” their decisions (and behavior). Events that caused transitions in a recently executed chain of events can provide important insight into the rationale for the program’s progress, something that may be harder to infer from a usual trace. This may be useful in developing and debugging behavioral applications, in using b-threads for monitoring, and in developing intelligent agents, expert systems or systems capable of learning.
- Suitability to multi-core and distributed systems: In behavioral programming each behavior thread is associated with an executable system process or thread. This “automatically” structures the developed system as a set of concurrent processes. In the context of Erlang, this enables leveraging the ease and efficiency of handling concurrent processes in this language together with natural decomposition of system behavior, towards a system that utilizes resources effectively while maintaining a natural and robust structure.

The proposed pattern and module are in early development stages and can be considered a proof-of-concept for demonstrating the principles of behavioral programming in Erlang.

Future directions for research include devising higher level idioms to control behavior, and developing domain specific languages based on behavioral programming principles.

For the tool, future work includes robustness improvements and adding functionality that exists already in the BPJ library. For example, BPJ supports event filters - calling of a function to determine membership of an event in the sets of blocked or watched events. This allows more flexible definition of the event sets, and handling of very large, or possibly infinite, sets.

The visualization tool we propose here is focused on comprehension of individual b-threads. It will be interesting to explore visualization techniques that assist in comprehension of sets of b-threads, in particular the interaction between different b-threads, possibly along the line of visualizing LSC dependencies as done in [4].

As a general programming paradigm, behavioral programming is still in very early stages. It is presently manifested in the visual, multi-modal language of LSC, and in the Java library BPJ. We hope that the design pattern proposed here for functional programming in Erlang will help expand the reach of this promising concept, and drive additional research and development required for its success.

Acknowledgments

The research of the first two authors was supported by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University. The research of the third author was supported by an Advanced Research Grant to David Harel from the European Research Council (ERC) under the European Community’s FP7 Programme.

References

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

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995. ISBN 0201633612.

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

[4] D. Harel and I. Segall. Visualizing inter-dependencies between scenarios. In R. Koschke, C. D. Hundhausen, and A. Telea, editors, *SOFTVIS*, pages 145–153. ACM, 2008. ISBN 978-1-60558-112-5.

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

[6] D. Harel, A. Marron, and G. Weiss. Programming coordinated scenarios in java. In *Proc. 24th European Conference on Object-Oriented Programming (ECOOP 2010)*, 2010. to appear.

[7] ITU. *International Telecommunication Union Recommendation Z.120: Message Sequence Charts*. 1996.

[8] R. Keller. Formal verification of parallel programs. *CACM*, 19(7): 371–384, 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360248.360251.

[9] OMG. *Unified Modeling Language Superstructure Specification, v2.0*. Aug. 2005. URL `http://www.omg.org`.

[10] A. D. Stefano and C. Santoro. Using the erlang language for multi-agent systems implementation. In *IAT*, pages 679–685. IEEE Computer Society, 2005. ISBN 0-7695-2416-8.

[11] A. D. Stefano, F. Gangemi, and C. Santoro. Eresye: artificial intelligence in erlang programs. In Erlang Workshop pages 62–71. ACM, 2005. ISBN 1-59593-066-3.

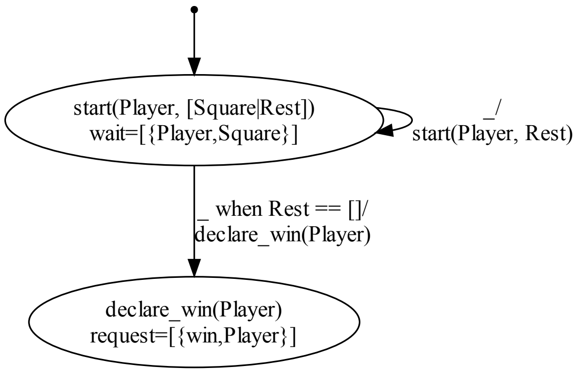


Figure 8. The `detect_win` b-thread. An instance is spawned for each of the 6 permutation of the 3 events that comprise one of the 8 winning lines (3 vertical, 3 horizontal, and 2 diagonal) for each of the two players (total of 96 instances). For example, an instance that waits for $\langle \text{x}, 9\rangle$, $\langle \text{x}, 5\rangle$, $\langle \text{x}, 1\rangle$, and announces a win by x is started by the line `spawn(detect_win, start, [x, [9,5,1]])`. For this example, in the first state, `Square` is matched with 9 and `Rest` with `[5,1]`. The b-thread waits for $\langle \text{x}, 5\rangle$ and, if that event occurs, `start(x, [5,1])` is called where `Square` is matched with 5 and `Rest` with `[1]` and so on. Eventually, if $\langle \text{x}, 5\rangle$ and afterwards $\langle \text{x}, 1\rangle$ occur, the b-thread requests the event $\langle win, \text{x}\rangle$ to announce that x won the game.

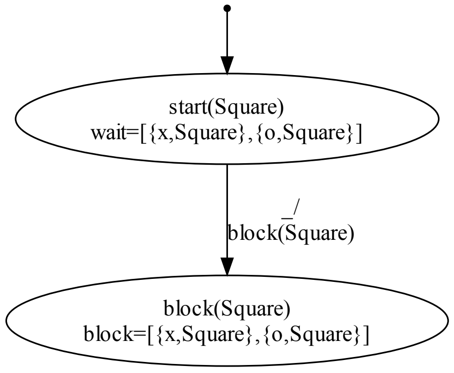


Figure 10. The `disallow_square_reuse` b-thread. Each of the nine instances of this b-threads waits for either $\langle \text{x}, Sq\rangle$ or $\langle \text{o}, Sq\rangle$, for a particular value of $Sq \in \{1, \ldots, 9\}$ and, when one of these two events is observed, the b-thread blocks them both forever.

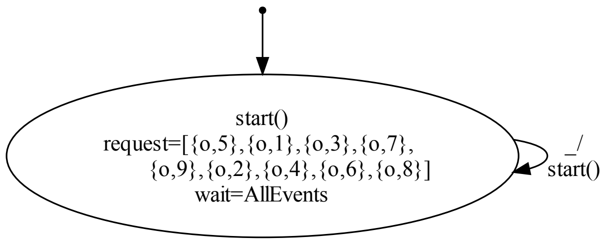

start()
request=[{o,5},{o,1},{o,3},{o,7},
{o,9},{o,2},{o,4},{o,6},{o,8}]
wait=AllEvents
_/
start()

Figure 11. The `default_moves` b-thread. Requested moves will be triggered only when not blocked and when there are no higher priority (unblocked) requests. The order of the requested events determines their priorities in our strategy: try to mark the center square first, then the corners, and only then the remaining squares.

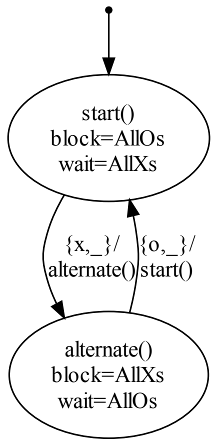


Figure 9. The `enforce_turns` b-thread. The variables `AllO` and `AllX` are lists of all the o events and all the x events, respectively. Turn enforcement is achieved by alternately blocking all o or all x events.

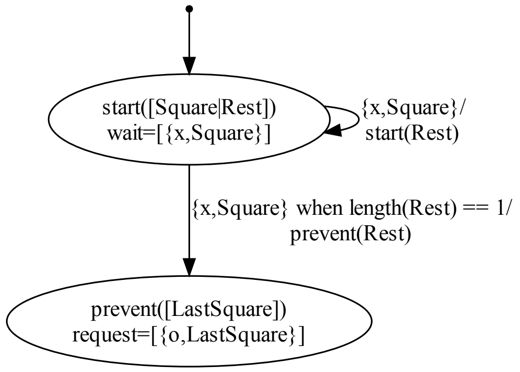


Figure 12. The `prevent_lines_with_two` b-thread. The parameter to the first state of the b-thread (called start) is a list of length three containing the three squares of a (vertical, horizontal, or diagonal) line that the b-thread is protecting (a copy of this b-thread is instantiated for every permutation of the squares of each line). Initially, the b-thread is waiting for the opponent to play the first square on the line. When this happens, the b-thread takes a self transition to the state `start` and waits for the opponent to mark the second square . Then, if the opponent marks the second square, the b-thread moves to the state `prevent` where a request to put an o on the last square is issued.

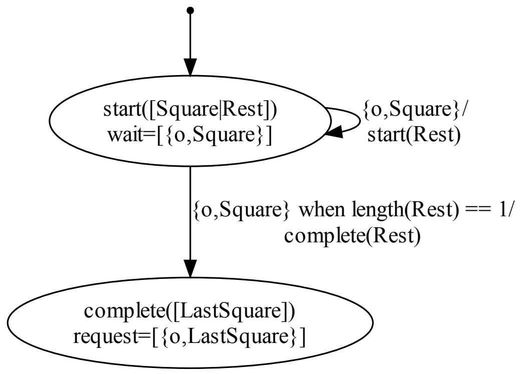


Figure 13. The `complete_lines_with_two` b-thread. Similar in structure to the `prevent_lines_with_two` b-thread, shown in Figure 12. The difference is that this b-thread is waiting for `o` moves. Note that the b-thread is independent of the scenarios that lead to requests of the first two `o` moves (issued by other b-threads).

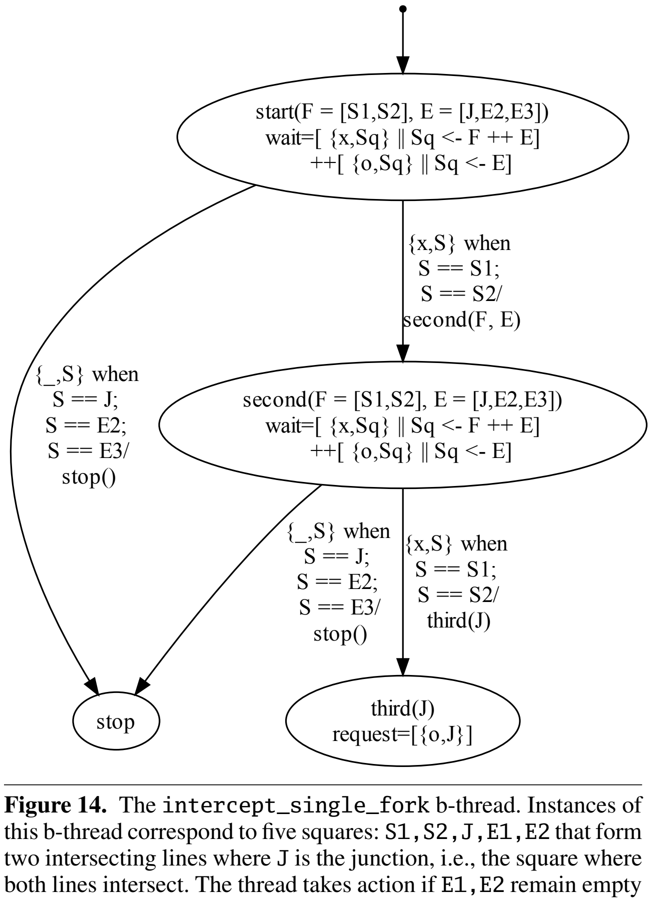


Figure 14. The `intercept_single_fork` b-thread. Instances of this b-thread correspond to five squares: `S1,S2,J,E1,E2` that form two intersecting lines where `J` is the junction, i.e., the square where both lines intersect. The thread takes action if `E1,E2` remain empty and `S1,S2` are marked by `x`. When this happens, the thread requests to mark `J` with `o`.

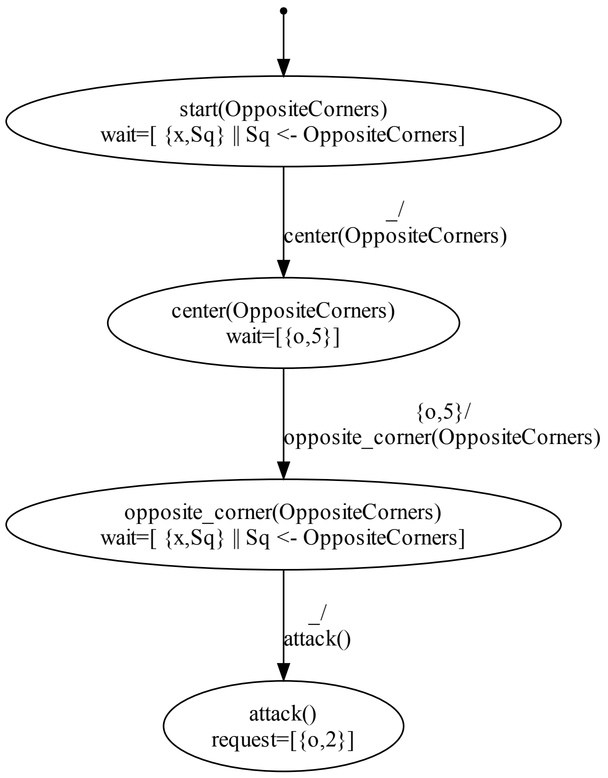

start(OppositeCorners)
wait=[{x,Sq} || Sq <- OppositeCorners]
_/
center(OppositeCorners)
center(OppositeCorners)
wait=[{o,5}]
{o,5}/
opposite_corner(OppositeCorners)
opposite_corner(OppositeCorners)
wait=[{x,Sq} || Sq <- OppositeCorners]
_/
attack()
attack()
request=[{o,2}]

Figure 15. The `intercept_double_fork` b-thread. Instances correspond to two opposite corners. The b-thread waits for `x` to mark one of the corners, then for `o` to mark the center and, lastly, if `x` marks the opposite corner, the b-thread requests to mark square number 2 (the middle of the upper row) with an `o`.