Relaxing Synchronization Constraints in Behavioral Programs

David Harel, Amir Kantor, and Guy Katz

Dept. of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel
{dharel,amir.kantor,guy.katz}@weizmann.ac.il

Abstract. In *behavioral programming*, a program consists of separate modules called *behavior threads*, each representing a part of the system's allowed, necessary or forbidden behavior. An execution of the program is a series of synchronizations between these threads, where at each synchronization point an event is selected to be carried out. As a result, the execution speed is dictated by the slowest thread. We propose an *eager execution* mechanism for such programs, which builds upon the realization that it is often possible to predict the outcome of a synchronization point even without waiting for slower threads to synchronize. This allows faster threads to continue running uninterrupted, whereas slower ones catch up at a later time. Consequently, eager execution brings about increased system performance, better support for the modular design of programs, and the ability to distribute programs across several machines. It also allows to apply behavioral programming to a variety of problems that were previously outside its scope. We illustrate the method by concrete examples, implemented in a behavioral programming framework in C^{++}.

Keywords: behavioral programming; synchronization; eager execution; modular design; distributed design.

1 Introduction

This work is carried out within the framework of *behavioral programming (BP)* [10] — a recently proposed approach for the development of reactive systems, which originated from the language of *live sequence charts* [5, 8]. The basis of the approach is the construction of systems from special threads, called *behavior threads (b-threads)*, each of which represents an aspect of the system's behavior which is specified as being allowed, necessary or forbidden. A simultaneous execution of these threads constitutes the combined system behavior.

An execution of the program is comprised of a series of synchronization points between the threads, each of which results in an event being triggered. The choice of the triggered event is performed by a global *coordinator*, which, at every synchronization point, receives input from all the threads before making the choice. This high amount of coordination grants behavioral programs many of their qualities: it eliminates race conditions between the threads, allows for multi-modal, modular and incremental development, and, in general, promotes the development of comprehensible and maintainable code. See [10].

However, extensive synchronization has implications on system performance (see [11]). Since all threads must synchronize before the system can continue to the next synchronization point, the step from one point to another is constrained by the slowest b-thread. In parallel architectures (e.g., multi-core processors), execution resources may stand idle while the system waits for a slow b-thread to finish performing nontrivial computations or time-consuming actions and reach the next synchronization point. Similar situations can also occur in programs that run on a single processor — for instance, if a b-thread is performing lengthy input/output actions that require no processing power, but delay its synchronization.

We introduce a new execution mechanism for behavioral programs, which we term *eager execution*. It allows relaxing the synchronization constraints between b-threads, resulting in a higher level of concurrency when executing the program. At the same time, eager execution maintains all information necessary for triggering events, and thus adheres to BP's semantics and supports its idioms.

Eager execution is made possible by automatically *analyzing* a thread prior to its execution, resulting in an approximation of the thread's behavior. With this information at hand, the eager execution mechanism can sometimes choose events for triggering without waiting for all of the threads to synchronize, thus improving the efficiency of the system's run and avoiding excessive synchronization. We present two analysis methods that lead to more eager execution: one is *static* and considers the thread as a whole, whereas the other is *dynamic* and takes into account the thread's state during the run. Both methods have been implemented and tested in *BPC*, a framework for behavioral programming in C^{++}. The framework itself, along with the examples described in this paper, is available online [1].

Relaxing synchronization is helpful in several contexts. First, it improves system performance and reduces processor idle time. Moreover, it gives rise to better modular design of the system, by grouping together related threads into components, which we call *behavioral modules*, and allowing these to operate independently on different time scales. Finally, the techniques presented in this paper can be leveraged to support a decentralized assimilation of the modules on different machines by distributing BP's execution mechanism. Distributed execution has been implemented and tested in BPC. It is not included in this paper due to space limitations; it is discussed in Appendix I of [2].

The paper is organized as follows. A short description of behavioral programming and the BPC tool appears in Section 2. We define the eager execution mechanism and present the two analysis methods in Section 3. In Section 4, we show how eager execution allows for a modular design of programs. Related work is discussed in Section 5, and we conclude in Section 6. Proofs are included in the appendices to this paper.

2 Behavioral Programming

A behavioral program consists of a set of *behavior threads (b-threads)*, each of which is an independent code module, which implements a certain part of the system's behavior. The threads are interwoven at run time through a series of *synchronization points*, and together produce a cohesive system.

The b-threads are driven by *events*, which are managed by a global *coordinator* that is implemented at the core of the behavioral programming framework. At every synchronization point, each thread BT passes to the coordinator three disjoint sets of events: those *requested* by BT, those for which BT *waits*, and those *blocked* by BT. BT then halts until the coordinator wakes it up.

Once *all* b-threads have reached a synchronization point, the coordinator calculates the set of *enabled events* — events that are requested by at least one b-thread and blocked by none. It then selects one of these events for triggering, say e, and passes it to some of the b-threads, and those then continue their execution until the next synchronization point. More specifically, e is passed to a thread BT if it is either requested or waited-upon by BT; other threads remain at the synchronization point, and their declared event sets are re-considered when the coordinator selects the next event. The model assumes that all inter-b-thread communication is performed through the synchronization mechanism.

Various implementations of reactive systems as behavioral programs have been carried out, using frameworks built on top of high-level programming languages such as Java, Erlang and Blockly; see [10] and references therein. These frameworks allow the user to use the full flexibility offered by the underlying programming language in writing threads. In this paper, we demonstrate our techniques using a BP framework in C^{++}, termed *BPC* [1].

For illustration, we provide an example of a vending machine programmed in BPC. The example is extended in later sections to demonstrate various aspects of our techniques. In this section, we only implement the basic functionality of the machine — collecting coins and dispensing products. The code consists of three b-threads, called *Dispenser*, *KeyPad* and *ProductSlot*; they are depicted in Fig. 1, 2, and 3, respectively. Observe that coin insertions and product selections are inputs from the environment. In the actual application they are implemented using a simple user interface, which is omitted from the code snippets. The same applies to the actual dispensing of the product in the *ProductSlot* thread.

We stress the key fact that the threads' transition from one synchronization point to the next may not be immediate. Since all the threads are required to synchronize in order for the coordinator to trigger an event, the thread that takes the longest to move from one synchronization point to the next dictates the speed of the entire system. This is the issue we address in the paper.

2.1 Behavioral Programming Formalized

While behavioral programming is geared toward natural and intuitive development using programming languages, its underlying infrastructure can be conveniently

```
class Dispenser : public BThread {
    void entryPoint() {
        while ( true ) {
            bSync( none, {CoinInserted}, none );
            bSync( none, {ProductChosen}, {CoinInserted} );
            bSync( {ProvideProduct}, none, {CoinInserted} );
}}};
```

Fig. 1: The ***Dispenser*** thread. This thread is responsible for dispensing wares, after the user inserts a coin and selects a desired product. The programmer writes behavioral code by overriding the method `entryPoint` of class *BThread*. The thread runs in an infinite loop, invoking the synchronization API `bSync` three times in each iteration; each invocation corresponds to a synchronization point, and includes three sets of events: requested (blue), waited-upon (green) and blocked (red). In the first synchronization point, the thread waits for a coin insertion, signified by a CoinInserted event. In the second, it waits for product selection, signified by a ProdcutChosen event. Finally, in the third, it dispenses the product, by requesting a ProvideProduct event. Since each call suspends the thread until an event that was requested or waited-for is triggered, one product is dispensed per coin; also, it is impossible to obtain the product without inserting a coin. Observe that the thread also blocks CoinInserted events during its last two synchronization points; otherwise, extra coins inserted before a product is provided could be swallowed by the machine.

```
while ( true ) {
    waitForCoinInsertion();
    bSync( {CoinInserted}, none, none );
    waitForProductSelection();
    bSync( {ProductChosen}, none, none );
}
```

Fig. 2: The main method of the ***KeyPad*** thread. This thread is an input "sensor" — a thread responsible for receiving inputs from the environment and translating them into BP events. It waits for the user to insert a coin and then requests a CoinInserted event. Then, it waits for the user to select a product, and requests a ProductChosen event. Coin insertions and product selections are inputs coming from the environment, and are abstracted away inside the functions `waitForCoinInsertion` and `waitForProductSelection`. The thread translates these inputs into events that are to be processed by other threads.

described and analyzed in terms of transition systems. We present an abstract formalization of behavioral programs and their semantics, similarly to [9, 11].

In the following definitions we implicitly assume a given set Σ of *events*. A *behavior thread (b-thread)* BT is abstractly defined to be a tuple $BT = \langle Q, q_0, \delta, R, B \rangle$, where Q is a set of *states*, $q_0 \in Q$ is an *initial state*, $\delta : Q \times \Sigma \to Q$ is a *transition function*, $R : Q \to \mathcal{P}(\Sigma)$ assigns for each state a set of *requested events*, and $B : Q \to \mathcal{P}(\Sigma)$ assigns for each state a set of *blocked events*. A *behavioral program* P is defined to be a finite set of b-threads.

Note that in the definitions above, a b-thread's transition rules are given as a *deterministic*, single valued, function δ, assigning the next state given a state and an event trigger in that state. A natural variant in which the transitions are *nondeterministic* is analogously defined; see Appendix II of the supplementary material [2]. The latter is useful for reactive systems, where the next state might also depend on external input. Also note that in the formal definition of a b-thread, there is no need to distinguish between events that are waited-upon by the thread, and those that are not. In any of the thread's states, an event that is not waited-upon can be captured by a transition that forms a self-loop; i.e., a transition that does not leave the state.

```
while ( true ) {
    bSync( none, {ProvideProduct}, none );
    provideActualProduct();
}
```

Fig. 3: The main method of the ***ProductSlot*** thread. This thread is an output "actuator"; it is responsible for translating ProvideProduct events into the dispensing of actual products. Thus, it waits for a ProvideProduct event, and then provides the product by invoking `provideActualProduct`.

Semantics Let $P = \{BT^1, \ldots, BT^n\}$ be a behavioral program, where $n \in \mathbb{N}$ and each $BT^i = \langle Q^i, q_0^i, \delta^i, R^i, B^i \rangle$ is a distinct b-thread. In order to define the semantics of P, we construct a deterministic *labeled transition system (LTS)* [12] denoted by LTS(P), which is defined as follows. LTS$(P) = \langle Q, q_0, \delta \rangle$, where $Q := Q^1 \times \cdots \times Q^n$ is the set of states, $q_0 := \langle q_0^1, \ldots, q_0^n \rangle \in Q$ is the initial state, $\delta : Q \times \Sigma \to 2^Q$ is a deterministic[1] transition function, defined for all $q = \langle q^1, \ldots, q^n \rangle \in Q$ and $a \in \Sigma$, by

$$\delta(\langle q^1, \ldots, q^n \rangle, a) := \begin{cases} \{\, \langle \delta^1(q^1, a), \ldots, \delta^n(q^n, a) \rangle \,\} & ; \text{if } a \in E(q) \\ \emptyset & ; \text{otherwise}\,. \end{cases}$$

where $E(q) = \bigcup_{i=1}^{n} R^i(q^i) \setminus \bigcup_{i=1}^{n} B^i(q^i)$ is the set of *enabled events* at state q.

An execution of P is an execution of the induced LTS(P). The latter is executed starting from the initial state q_0. In each state $q \in Q$, an enabled event $a \in \Sigma$ is selected for triggering if such exists (i.e., an event $a \in \Sigma$ for which $\delta(q, a) \neq \emptyset$). Then, the system moves to the next state $q' \in \delta(q, a)$, and the execution continues. Such an execution can be formally recorded as a possibly infinite sequence of triggered events, called a *run*. The set of all *complete* runs is denoted by $\mathfrak{L}(P) := \mathfrak{L}(\text{LTS}(P))$. It contains either infinite runs, or finite ones that terminate in a state in which no event is enabled, called a *terminal state*.

3 The Eager Execution Mechanism

We begin with a general description of our proposed execution mechanism for BP, termed *eager execution*. Let $P = \{BT^1, \ldots, BT^n\}$ be a behavioral program consisting of b-threads $BT^1, \ldots, BT^n$. Assume that at some point in the execution of P, a subset $P_{\text{sync}} \subseteq P$ of the threads has reached a synchronization point, while the rest are still executing. Further, assume that the coordinator has additional information about the events that the threads in $P \setminus P_{\text{sync}}$ will request and block at the next synchronization point. If, combining the information from threads in P_{sync} with the information about threads in $P \setminus P_{\text{sync}}$, the coordinator can find an event e that will be enabled at the next synchronization point, then e can immediately be chosen for triggering.

The coordinator may then pass e to the threads in P_{sync} to let them continue their execution immediately, without waiting for the remaining threads to synchronize. Once any of these other threads reaches its synchronization point, the

[1] I.e., its range includes only singletons and the empty set.

coordinator immediately passes it event e, as this event was selected for that particular synchronization point. This is accomplished by having a designated queue for each of the b-threads, of events that are waiting to be passed, and putting e in the queues corresponding to the not-yet synchronized threads. The execution mechanism described is *eager*, in the sense that it uses predetermined information to choose the next event as early as possible.

When a thread BT reaches a synchronization point, if the corresponding queue is nonempty, the coordinator dequeues the next pending event e'. If BT requests or waits for e', it is passed to the thread, which then continues to execute. Otherwise, e' is ignored, and the coordinator continues with the next event pending in the queue. In order to reflect the semantics of BP, from the coordinator's global perspective BT is not considered synchronized as long as it has events pending in the queue. Particularly, the events that are requested or blocked by BT at this point are not considered for the selection of the next event; the coordinator considers only threads that have synchronized and for which there are no pending events (so that they are halted).

Observe that the eager execution mechanism strictly adheres to the semantics of BP, as described in Section 2; at every synchronization point, the triggered event is indeed enabled. Consequently, we get the following result:

Proposition 1. *Given a behavioral program P, the sequence of events triggered by the eager execution mechanism is a valid run (under BP's semantics).*

The key point, however, is that the eager mechanism makes its decisions more quickly, and thus often produces more efficient runs. The eager execution mechanism is formalized in Section 3.3, and Proposition 1 is proved in Appendix A.

It remains to show how the execution mechanism knows which events could be requested and blocked by threads that are yet to synchronize. We propose two approaches: *static analysis* and *dynamic analysis*.

3.1 Static Analysis

In this approach, the coordinator is given in advance a static over-approximation of the events that a thread might block when synchronizing. Explicitly, if a thread has states $s_1, \ldots, s_n$, this over-approximation is $\bigcup_{1 \leq i \leq n} B(s_i)$, where $B(s_i)$ is the set of events blocked in state s_i. The over-approximation is static in the sense that it does not change throughout the run.

When a thread synchronizes, the coordinator checks if there are events that are enabled based on the data gathered so far — namely, events that are requested and not blocked by threads in P_{sync}, and that are never blocked by the other threads, based on their over-approximations. If such an event exists, it can be triggered immediately. Otherwise, the coordinator waits for more threads to synchronize. This generally results in more events becoming enabled, since the actual set of events that are blocked by a thread is always a subset of the over-approximation, and since additional requested events are revealed. As soon as enough information is gathered to deduce that an event is enabled, it is immediately triggered and passed to all synchronized threads. For threads that are yet to synchronize, the event is stored in a designated queue, to be passed to them upon reaching their synchronization point.

Observe that we only discuss over-approximating blocked events but not the approximation of requested events. The reason is that the analogous version would entail using an under-approximation of requested events; and, since threads do not generally request an event in each of their states, these under-approximations are typically empty.

Example: Using Static Analysis We further evolve the example from Section 2. Suppose that the vending machine's developer wishes to introduce a maintenance mechanism. Once every fixed period of time, the machine is to go into maintenance mode and measure its inner temperature and humidity.

This type of requirement poses a challenge, in the form of integrating different time scales into a behavioral program. If maintenance is to occur every t seconds, a natural approach is adding a thread with the following structure, wrapped in a loop: (a) sleep for t seconds; (b) request an InitiateMaintenance event. Unfortunately, under a traditional BP execution mechanism, this results in the entire system pausing for t seconds at a time; since the thread does not reach the next synchronization point while asleep, the coordinator is unable to trigger an event, and any coin insertions or product requests by the user go unanswered between maintenance phases.

One solution is to have the event which initiates the periodic maintenance be triggered by some external entity — similarly to coin insertions and product selections. This approach, though feasible, means that the system would depend on these external events in order to operate properly; the BP framework does not offer a way to enforce their proper generation.

Instead, we adopt a solution that combines in-line waiting and eager execution. We use the method described above, and declare (or, as we later discuss, find automatically) that the new thread does not block any events; in effect, this tells the coordinator that it should not wait for it at any synchronization point. The system can then progress, and go along serving clients, while the thread is asleep. When the thread awakes and synchronizes, it is informed, one at a time, of the events that have occurred so far, and it can then synchronize and request that maintenance be triggered. The new thread is depicted in Fig. 4.

```
while ( true ) {
    sleep( TimeBetweenMaintenancePeriods );
    bSync( {InitiateMaintenance}, none, none );
}
```

Fig. 4: The main method of the ***MaintenanceTimer*** thread. `TimeBetweenMaintenancePeriods` is a constant, indicating the desired time between consecutive maintenance cycles. Whenever the thread wakes up it requests an InitiateMaintenance event, and then goes back to sleep. Observe that since the thread neither requests nor waits for any other events, any events that were triggered while it was asleep — such as coin insertions or user selections — are not passed on to it when it awakes. Therefore, it immediately catches up with the execution upon waking up.
In order to tell the coordinator that the *MaintenanceTimer* thread blocks no events, the following line of code is provided as well: `bProgram.addThreadBlockingData( "MaintenanceTimer", none );`
This allows the coordinator to trigger an event even if this thread has not synchronized yet.

3.2 Dynamic Analysis

In this approach, the coordinator is given complete *state graphs* of the threads, which are automatically calculated before the program is executed. The labeled vertices of a state graph correspond to the thread's synchronization points and requested/blocked events, while the labeled edges correspond to the program's events (that are not blocked at that state). The graph thus provides a complete description of the thread from the coordinator's point of view — that is, a complete description of the events requested and blocked by the thread, but without any calculations or input/output actions performed by the thread when not synchronized. For more details on these state graphs, see [6].

During runtime, the coordinator keeps track of the threads' positions in the graphs, allowing it to approximate the events they will request and block at the next synchronization point — even before they actually synchronize. This method is dynamic, in the sense that the approximations for a given thread can change during the run, as different states are visited. The fundamental difference between running a thread and simulating its run using its state graph is that in the latter, no additional computations are performed, and consequently transitions can be considered immediate.

Recall that our definition of threads dictates that a thread's transitions be deterministic. Therefore, simulating a thread through its state graph yields precise predictions of its requested and blocked events at each synchronization point. In the nondeterministic model, where threads may depend on coin tosses or inputs from the environment, it may be impossible for the coordinator to determine a thread's exact state until it synchronizes; however, the coordinator can approximate the thread's requested and blocked events by considering all the states to which the nondeterministic transitions might send the thread. If, due to a previous transition, the thread is known to be in one of states $s_1, \ldots, s_n$, then the blocked events may be over-approximated by $\bigcup_{1 \leq i \leq n} B(s_i)$ — similarly to what is done in static analysis. Analogously, the requested events may be under-approximated by $\bigcap_{1 \leq i \leq n} R(s_i)$. For more details see Appendix II of the supplementary material [2]. As before, if these approximations leave no enabled events, the coordinator waits for more threads to synchronize.

The other details are as they were in the static analysis scheme. Once an event is triggered, it is immediately sent to all synchronized threads, and is placed in queues for threads that are yet to synchronize.

Example: Using Dynamic Analysis In Section 3.1, we added a thread that periodically initiates a maintenance process in the vending machine. We now describe this process in greater detail. Suppose that the goal of the maintenance process is to keep the machine's temperature and humidity at a certain level. Maintenance thus includes two phases: measurement and correction, applied once for temperature and once for humidity. For simplicity, assume that both values are always out of the safe range; i.e., that they always require adjusting.

To handle these requirements, we add two new threads to the program — one to do the measurements, and one to do the corrections. The first, the *Measurer*, reads information from the environment through sensors, while the second, the *Corrector*, affects the environment, through air conditioning and humidity control systems. These threads are triggered by the periodic InitiateMaintenance event, as described earlier. Code snippets appear in Fig. 5 and Fig. 6.

```
while ( true ) {
    bSync( none, {InitiateMaintenance}, none );

    if ( temperatureTooHigh() )
        bSync( {DecreaseTemperature}, none, {ProvideProduct} );
    else bSync( {IncreaseTemperature}, none, {ProvideProduct} );
    bSync( none, {TemperatureCorrected}, {ProvideProduct} );

    if ( humidityTooHigh() ) bSync( {DecreaseHumidity}, none, none );
    else bSync( {IncreaseHumidity}, none, none );
    bSync( none, {HumidityCorrected}, none );
}
```

Fig. 5: The main method of the ***Measurer*** thread. Upon triggering of the InitiateMaintenance event, this thread wakes up, asks for the appropriate temperature correction, and waits for confirmation. Afterwards, an analogous process is performed for the humidity level. Observe that the ProvideProduct event is blocked during the temperature phase, but not during the humidity phase.

```
while ( true ) {
    bSync( none, allEvents(), none );
    if ( lastEvent() == IncreaseTemperature ) {
        increaseTemperature(); bSync( {TemperatureCorrected}, none, none );
    }
    else if ( lastEvent() == DecreaseTemperature ) {
        decreaseTemperature(); bSync( {TemperatureCorrected}, none, none );
    }
    else if ( lastEvent() == IncreaseHumidity ) {
        increaseHumidity(); bSync( {HumidityCorrected}, none, none );
    }
    else if ( lastEvent() == DecreaseHumidity ) {
        decreaseHumidity(); bSync( {HumidityCorrected}, none, none );
    }
}
```

Fig. 6: The main method of the ***Corrector*** thread. The thread waits for events IncreaseTemperature, DecreaseTemperature, IncreaseHumidity or DecreaseHumidity; if they are triggered, it responds by adjusting the temperature or humidity (this part is abstracted away in the subroutines). Then, the thread requests an event notifying that the request has been handled, and goes back to waiting for new requests. Accessing the last event triggered is performed via the `lastEvent` method.

Another requirement is that, due to constraints in the machine, it is forbidden to dispense products between temperature measurement and correction, otherwise the correction might be interrupted. Therefore, the *Measurer* thread blocks events of type ProvideProduct during temperature measurement and correction. During humidity measurement, however, this limitation does not apply. As measurement and correction operations take a non-zero amount of time, there is a time window during maintenance in which the dispensing of products is forbidden.

We seek a solution that would prevent dispensing products during the temperature phase, but would permit it during the humidity phase. Static analysis does not suffice: as the *Measurer* thread blocks the ProvideProduct event at some of its states, the over-approximation includes this event — and so ProvideProduct events would not be triggered during humidity measurement and correction. Dynamic analysis, on the other hand, resolves this issue, as it is able to distinguish between the two phases; see Table 1 for performance comparison.

Table 1: Performance of the vending machine program using the different execution mechanisms. The measurements were performed using a customer simulator, purchasing 250 products in random intervals. The table depicts the time the experiment took, the number of maintenance rounds performed during the experiment, and the average delay — the time between making an order and receiving the product. The improvement column measures the reduction in delay compared to the traditional execution mechanism.

Execution	**#Servings**	**Time (min)**	**#Maintenance**	**Delay (sec)**	**Improvement**
Traditional	250	15:40	59	1.68	—
Static	250	12:30	50	0.85	50%
Dynamic	250	9:20	37	0.18	90%

We point out that the *Measurer* thread's transitions are not deterministic — as they depend on input from the `temperatureTooHigh` and `humidityTooHigh` subroutines. As previously explained, this does not pose a problem, as the coordinator calculates an over-approximation based on all the successor states of the thread's last known state.

Remark: Recall that dynamic analysis includes spanning the state graphs of threads and integrating these graphs into the program. Manual spanning of state graphs is prone to error, and is rather tedious in large systems with many events. Consequently, BPC includes an automated tool for performing this spanning without any overhead on the programmer's side.

The spanning is performed by separating the thread under inspection from its siblings, and then iteratively exploring its state graph until all its states and transitions have been found. Starting at the initial state, we check the thread's behavior in response to the triggering of each event that is not blocked by the thread in that state. After the triggering of each event, the thread arrives at a new state (synchronization point) — and, with proper book keeping, it is simple to check if the state was previously visited or not. New states are then added to a queue to be explored themselves, in an iterative BFS-like manner.

Isolating threads is performed using the CxxTest [15] tool, which is able capture and redirect function calls within programs. The thread's calls to the synchronization method `bSync` are captured, and used to determine the thread's current state; similarly, calls to the `lastEvent` method are captured and used to fool the thread into believing that a certain event was just triggered. The strength of this method is that the entire process takes place using the original, unmodified program code. Other methods, such as the one used in BPJ [7], include adding dedicated threads for this purpose — a process that might in itself introduce additional errors. Once the state graph has been spanned, it is automatically transformed into a C^{++} code module and integrated into the program.

3.3 Eager Execution Formalized

We now formally define the the eager execution mechanism. All definitions in this section exclusively consider deterministic b-threads; handling nondeterministic ones is similar (see Appendix II of the supplementary material [2]).

Let $P = \{BT^1, \dots, BT^n\}$ be a behavioral program, where $n \in \mathbb{N}$ and each BT^i is a distinct b-thread. In order to define the eager execution mechanism, we construct a labeled transition system (LTS) denoted by $\widehat{\mathrm{LTS}}(P) = \langle \widehat{Q}, \widehat{q_0}, \widehat{\delta} \rangle$, which is defined next. We use some of the notation introduced in Section 2.1.

The set of states is given by $\widehat{Q} := (Q^1 \times \Sigma^*) \times \cdots \times (Q^n \times \Sigma^*)$. Each state is thus a tuple consisting for each thread of its state and the contents of its event queue. Let $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$ be a state. We use the standard notation $\delta^i(q^i, u^i)$ to denote the state in Q^i after applying the transition function δ^i of thread BT^i starting from state q^i for each event in the queue u^i. Given q, we denote the tuple comprised of these states by $\overline{q} := \langle \delta^i(q^i, u^i) \rangle_{i=1}^n$; we refer to it as the *indication* of q. Note that $\overline{q}$ naturally corresponds to a state in Q, which is the set of states of $\mathrm{LTS}(P) = \langle Q, q_0, \delta \rangle$ defined in Section 2.1. We slightly abuse notation and write that $\overline{q} \in Q$. Naturally, the initial state is $\widehat{q_0} := \langle (q_0^1, \varepsilon), \dots, (q_0^n, \varepsilon) \rangle \in \widehat{Q}$.

In each state $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$, eager execution *approximates* the requested and blocked events of each thread. This is indicated by the following sets of events: $\mathcal{R}^i(q) \subseteq \Sigma$, for the requested events of thread BT^i, and $\mathcal{B}^i(q) \subseteq \Sigma$, for the its blocked events. As previously mentioned, eager execution has various forms (depending on the analysis technique that is used); each form is characterized by its specific choice for these approximations. The requirements imposed on them are the following. We require that $\mathcal{R}^i(q)$ is a subset of the events that are requested by thread BT^i at state $\delta^i(q^i, u^i)$, and that $\mathcal{B}^i(q)$ is a superset of the blocked events at that state. That is,

$$\mathcal{R}^i(q) \subseteq R^i(\delta^i(q^i, u^i)) \qquad B^i(\delta^i(q^i, u^i)) \subseteq \mathcal{B}^i(q)\,. \tag{1}$$

Moreover, we require that in case a thread is synchronized, the two approximations are precise. More formally, if $u^i = \varepsilon$ for some $i \in [n]$ (where $[n]$ denotes the set of indices $\{1, \dots, n\}$), so that in particular $\delta^i(q^i, u^i) = q^i$, then we require

$$\mathcal{R}^i(q) = R^i(q^i) \qquad \mathcal{B}^i(q) = B^i(q^i)\,. \tag{2}$$

These two requirements are sufficient for our purposes. One may easily verify that the eager execution with either static or dynamic analysis technique complies with the requirements. From these, we obtain that the *approximated enabled events*, defined in the following, are contained in the enabled events at the indication state $\overline{q} \in Q$; i.e.,

$$\mathcal{E}(q) := \bigcup_{i=1}^{n} \mathcal{R}^i(q) \setminus \bigcup_{i=1}^{n} \mathcal{B}^i(q) \subseteq E(\overline{q})\,. \tag{3}$$

In case all threads are synchronized, i.e., $u^i = \varepsilon$ for all $i \in [n]$, we obtain

$$\mathcal{E}(q) = E(\overline{q})\,. \tag{4}$$

The nondeterministic transition function $\widehat{\delta} : \widehat{Q} \times (\Sigma \dot{\cup} \{\varepsilon\}) \rightarrow 2^{\widehat{Q}}$ includes also silent ε-labeled transitions; these ε transitions are not considered part of the runs of the system. $\widehat{\delta}$ is defined for each state $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$, and $\sigma \in \Sigma \cup \{\varepsilon\}$, as:

- If $\sigma = \varepsilon$, then $\widehat{\delta}(q, \varepsilon)$ is defined to be those states $\langle r^i, v^i \rangle_{i=1}^n \in \widehat{Q}$ for which there is $i_0 \in [n]$ and $a \in \Sigma$ such that $u^{i_0} = a\, v^{i_0}$ and $r^{i_0} = \delta^{i_0}(q^{i_0}, a)$, and for all other $i \in [n] \setminus \{i_0\}$ it holds that $r^i = q^i$ and $v^i = u^i$. These transitions correspond to threads with queued events processing these events — they change states, while the other threads do not move.
- If $\sigma \in \Sigma$, and moreover $\sigma \in \mathcal{E}(q)$, then $\widehat{\delta}(q, \sigma)$ is defined to be the singleton $\widehat{\delta}(q, \sigma) = \left\{ \langle q^i, u^i\, \sigma \rangle_{i=1}^n \right\}$. These transitions correspond to new events being triggered.
- If $\sigma \in \Sigma$ and $\sigma \notin \mathcal{E}(q)$, we define $\widehat{\delta}(q, \sigma) = \emptyset$. This reflects the fact that events that are not enabled cannot be triggered.

For a rigorous proof of Proposition 1 using these definitions, see Appendix A.

4 Modularity by Eager Execution

Complex systems can generally benefit from being partitioned into several components, each assigned its own execution resources (e.g., a dedicated computer) [13]. An intelligent partitioning of the system into components makes it possible to execute different facets of system behavior independently, and thus improve response time to different tasks. This is particularly crucial when system behavior involves multiple time scales.

The fact that a behavioral program consists of a collection of threads, each addressing part of the system's behavior, suggests a natural way to design program components. We call a collection of b-threads that collectively addresses a certain facet of the system a *behavioral module*. Each such module can be assigned distinct computational resources (e.g., a computer) so as to form an independent component. However, BP's complete stepwise synchronization between the b-threads undermines the benefits expected from such a design. In particular, it would not result in alleviating run-time dependencies between the components.

In order to understand how eager execution affects behavioral modules, we make the following definitions. Consider a behavioral program P consisting of a set behavioral modules $M_1, \ldots, M_k$; thus, the threads in the program are $\bigcup_{i=1}^k M_i$. Denote by E_i the set of events that are *controlled* — i.e., requested or blocked — at some synchronization point of a thread of module M_i. Typically, these events are part of the 'vocabulary' corresponding to that facet of the system addressed in module M_i. The modular design of the program is termed *strict* if $E_1, \ldots, E_k$ are pairwise disjoint; i.e., $E_i \cap E_j = \emptyset$ for $i \neq j$. However, any thread can wait for any event. A strict modular design essentially means that while modules may signal one another (by waiting for each other's events), they do *not* control each other's events; i.e., they are assigned sufficiently independent duties.

For a strict modular design, the eager execution mechanism results in an implementation in which the threads in each module never need to wait for a thread in another module to synchronize. Here, static analysis, as described in Section 3.1, is enough. The modules are thus effectively independent and may involve different time scales. This is formalized by the following proposition:

Proposition 2. *Let P be a behavioral program having a strict modular design and executed with the eager execution mechanism. If all b-threads of module M_i are synchronized, then an event $e \in E_i$ is enabled if and only if it will also be enabled upon the arrival of any other thread at its synchronization point.*

The proposition implies that, in a strict design, as soon as a module's threads have synchronized any enabled event that they control may immediately be triggered, without waiting for threads from other modules. See Appendix III of the supplementary material [2] for a rigorous definition of a modular program design and a proof of the proposition.

4.1 Example: A Modular Design

We implement the traveling vehicles example from [11, Section 7]. The example includes several vehicles, each operating as an autonomous component traveling on pre-given cyclic route along an (x, y) grid; in each given time unit during the run, each vehicle can travel north, east, south or west. We assume that all vehicles travel at identical speeds, i.e., cover one unit of distance per time unit.

Using eager execution, this multi-component system can be programmed entirely within the behavioral programming framework, without relying on external means of communication. The threads of each vehicle, v_i, form an independent behavioral module M_i that involves a designated set of events. This results in a strict modular design allowing each vehicle to operate independently of others. A code snippet for the main thread of vehicle v_i is depicted in Fig. 7. If each module has a dedicated processor and event selection is fair, all vehicles are constantly moving — as the coordinator does not wait for vehicle v_i to finish moving and synchronize again before triggering the movement requested by another vehicle.

```
while ( true ) {
    Vector<Event> requestedEvents;
    if ( destinationIsNorth() ) requestedEvents.append( #iMoveNorth );
    if ( destinationIsSouth() ) requestedEvents.append( #iMoveSouth );
    if ( destinationIsEast() ) requestedEvents.append( #iMoveEast );
    if ( destinationIsWest() ) requestedEvents.append( #iMoveWest );
    bSync( requestedEvents, none, none );
    adjustPositionByLastEvent();
}
```

Fig. 7: The main method of each vehicle thread. The placeholder '#i' is replaced by the number of the vehicle; for instance, for vehicle v_5, the events are 5MoveNorth, 5MoveWest, etc. The thread requests moves in all directions that bring it closer to the destination. When the call to bSync returns, one of these moves was selected by the behavioral execution mechanism. The thread then updates its position (by invoking adjustPositionByLastEvent), and proceeds.

Eager execution allows a light-weight solution if communication between the vehicles is required — e.g., for collision prevention. Each vehicle can be accompanied by an adviser thread that keeps track of other vehicles. Whenever its vehicle is dangerously close to another, the adviser blocks movement in the dangerous direction (for simplicity, deadlocks are ignored). As the modular design remains strict, adding the adviser threads does not impede the vehicles' ability to move independently.

5 Related Work

Within the scope of BP, an alternative approach for supporting modular designs and multiple time scales in behavioral programs is suggested in [11], where a program consists of sub-programs, called *behavior nodes* (*b-nodes*), each with its own pool of (internal) events. Coordination between the b-nodes is carried out by sending *external events* from one to another. Thus, internal events have to be translated into external events and vice versa. The feasibility of this approach is exhibited in [11] by using several examples.

Observing that the b-node approach naturally induces a strict modular design, our approach offers similar benefits but without the need to go beyond the behavioral programming idioms; indeed, no additional layer of external events is needed. Relaxed synchronization also supports more general, non-strict designs, in which behavioral programming idioms are used more liberally. In the case of non-strict designs, eager execution does not ensure that the modules are executed independently. Nevertheless, it avoids unnecessary synchronization between the modules (especially when using dynamic analysis of the threads), which may be sufficient in many situations.

Outside the scope of BP, performance optimization and communication minimization in parallel and distributed settings have been studied extensively. The trade-off between these two goals is discussed in [4, 16]. In [14], the author suggests imposing certain limitations on the communication between the components, which allows for execution-time optimization to be performed during compilation.

A method similar to our static analysis appears in [3], where invariants about system components are used for conflict resolution within the BIP framework.

6 Conclusion and Future Work

The contribution of this paper is in the proposed eager execution mechanism, which allows relaxing synchronization in behavioral programs. This scheme generally improves system performance, and allows behavioral programs to be written using a modular design that supports multiple time scales. Our approach is made possible by the realization that, by analyzing a b-thread prior to its execution, it is sometimes possible to accurately predict a valid outcome of a synchronization point without actually waiting for the thread to synchronize.

In this paper we made no assumptions on how the coordinator chooses the next event to be triggered from among the enabled events. In practice, however, such assumptions can sometimes simplify system development. One example is the *prioritized* event selection used in [9]. We believe that our methods can be naturally adapted to such mechanisms too.

The technique discussed in this paper requires that each b-thread communicate with a *global* coordinator at every synchronization point. While this constraint is significantly weaker than stepwise synchronization with all other b-threads, it may limit the applicability of the approach for designing multi-component applications in distributed architectures, in which communication is costly and time-consuming. In Appendix I of [2], we show how a variant of eager execution, called *distributed execution*, can be utilized to reduce these costs. This is done at the expense of not completely refraining from synchronization between threads of different modules, even in a strict modular design, so that Proposition 2 does not hold in that context. Finding ways to reduce communication costs while still upholding Proposition 2 is left for future work.

Acknowledgements We thank Assaf Marron, Gera Weiss and Guy Wiener 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.

References

1. BPC: Behavioral Programming in C^{++}. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/`.
2. Supplementary material. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/relaxedSync/`.
3. S. Bensalem, M. Bozga, J. Quilbeuf, and J. Sifakis. Knowledge-Based Distributed Conflict Resolution for Multiparty Interactions and Priorities. *Formal Techniques for Distributed Systems*, Giese, H. & Rosu, G. (eds.), Lecture Notes in Computer Science, 7273:118–134, 2012.
4. Y. Cheng and T. Robertazii. Distributed Computation with Communication Delay [Distributed Intelligent Sensor Networks]. *IEEE Transactions on Aerospace and Electronic Systems*, 24(6):700–712, 1988.
5. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
6. 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.
7. 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.
8. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer-Verlag, 2003.
9. 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.
10. D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Comm. Assoc. Comput. Mach.*, 55(7):90–100, 2012.

11. 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.
12. R. Keller. Formal verification of parallel programs. *Comm. Assoc. Comput. Mach.*, 19(7):371–384, 1976.
13. D. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. Assoc. Comput. Mach.*, 15(12):1053–1058, 1972.
14. A. J. van Gemund. The Importance of Synchronization Structure in Parallel Program Optimization. In *Proc. 11th ACM Int. Conf. on Supercomputing (ICS)*, pages 164–171, 1997.
15. E. Volk. CxxTest: A Unit Testing Framework for C^{++}. http://cxxtest.com/.
16. J. K. Yook, D. M. Tilbury, and N. R. Soparkar. Trading Computation for Bandwidth: Reducing Communication in Distributed Control Systems Using State Estimators. *IEEE Transactions on Control Systems Technology*, 10(4):503–518, 2002.

A Proof of Proposition 1

In this section we formally prove Proposition 1, stating that the runs produced by the eager execution mechanism are valid runs according to BP's original semantics. We consider $\widehat{\mathrm{LTS}}(P)$ from Section 3.3, which captures the execution of program P using the eager execution mechanism, and $\mathrm{LTS}(P)$ from Section 2.1, which captures the original semantics. Technically, we claim that each complete run of $\widehat{\mathrm{LTS}}(P)$ is a complete run of $\mathrm{LTS}(P)$; i.e., $\mathfrak{L}(\widehat{\mathrm{LTS}}(P)) \subseteq \mathfrak{L}(\mathrm{LTS}(P))$. This is a consequence of the following lemmata.

When considering $\widehat{\mathrm{LTS}}(P)$, $q \xrightarrow{\sigma} q'$ stands for $q' \in \widehat{\delta}(q, \sigma)$, as customary when discussing transition systems (for any states $q, q' \in \widehat{Q}$ and a possibly silent event $\sigma \in \Sigma \cup \{\varepsilon\}$). Also, recall that $q \in \widehat{Q}$ is a *terminal state* if for all $\sigma \in \Sigma \cup \{\varepsilon\}$ it holds that $\widehat{\delta}(q, \sigma) = \emptyset$. Similar notations and terminology apply to $\mathrm{LTS}(P)$.

Lemma 1. *Let $q, q' \in \widehat{Q}$ and $\sigma \in \Sigma \cup \{\varepsilon\}$ such that $q \xrightarrow{\sigma} q'$ in $\widehat{\mathrm{LTS}}(P)$.*

1. *If $\sigma = \varepsilon$, then $\overline{q'} = \overline{q}$.*
2. *If $\sigma \in \Sigma$, then $\overline{q} \xrightarrow{\sigma} \overline{q'}$ in $\mathrm{LTS}(P)$.*

Proof. 1: Denote $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$, and suppose that $\sigma = \varepsilon$. By the definition of $\widehat{\delta}$, we obtain that $q' = \langle r^i, v^i \rangle_{i=1}^n$, where all the coordinates are the same as in q, except for the one corresponding to $i_0 \in [n]$. In the latter coordinate we get $\delta^{i_0}(r^{i_0}, v^{i_0}) = \delta^{i_0}(\delta^{i_0}(q^{i_0}, a), v^{i_0}) = \delta^{i_0}(q^{i_0}, a\, v^{i_0}) = \delta^{i_0}(q^{i_0}, u^{i_0})$, as needed.

2: Now, suppose $\sigma \in \Sigma$. According to the definition of $\widehat{\delta}$, $\sigma \in \mathcal{E}(q)$ and $q' = \langle q^i, u^i\, \sigma \rangle_{i=1}^n$. By (3) (see Section 3.3) and by the definition of δ, we get that in $\mathrm{LTS}(P)$ it holds that $\overline{q} \xrightarrow{\sigma} \langle\, \delta^i(\delta^i(q^i, u^i), \sigma)\, \rangle_{i=1}^n = \langle\, \delta^i(q^i, u^i\, \sigma)\, \rangle_{i=1}^n = \overline{q'}$. □

Corollary 1.

1. *Let $r_0 \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_k} r_k$ be a finite execution of $\widehat{\mathrm{LTS}}(P)$ ($k \geq 0$). There exists a finite execution $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_t} s_t$ of $\mathrm{LTS}(P)$ ($t \geq 0$) such that $\overline{r_k} = s_t$ and $\sigma_1\, \sigma_2 \cdots \sigma_k = a_1\, a_2 \cdots a_t$.*

2. *Let* $r_0 \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots$ *be an infinite execution of* $\widehat{\mathrm{LTS}}(P)$. *There exists an execution* $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$ *of* $\mathrm{LTS}(P)$ *such that* $\sigma_1\,\sigma_2 \cdots = a_1\,a_2 \cdots$.

Proof (sketch). 1: By induction on k. For $k = 0$ the claim follows from the fact that $\overline{\widehat{q_0}} = q_0 \in Q$; the induction step follows from Lemma 1.

2: By an inductive construction of the execution, which similarly follows from Lemma 1. □

Lemma 2.

1. *If* $q \in \widehat{Q}$ *is a terminal state in* $\widehat{\mathrm{LTS}}(P)$, *then* $\overline{q}$ *is a terminal state in* $\mathrm{LTS}(P)$.
2. *There is no infinite sequence* $q \xrightarrow{\varepsilon} q' \xrightarrow{\varepsilon} q'' \xrightarrow{\varepsilon} \cdots$ *in* $\widehat{\mathrm{LTS}}(P)$.

Proof. 1: As q is terminal, by the definition of $\widehat{\delta}$ it holds that all the queues in q are empty (otherwise, $\widehat{\delta}(q, \varepsilon) \neq \emptyset$); i.e., $q = \langle q^i, \varepsilon \rangle_{i=1}^n$. Let $a \in \Sigma$. Because q is terminal, $a \notin \mathcal{E}(q)$. Thus, by (4) (see Section 3.3), $a \notin E(\overline{q})$, and therefore by the definition of δ, $\delta(\overline{q}, a) = \emptyset$.

2: For each state $q = \langle q^i, u^i \rangle_{i=1}^n \in \widehat{Q}$, consider the total size of the queues, denoted by $\varphi(q) := \Sigma_{i=1}^n |u^i| \in \mathbb{N}$. Given such an infinite sequence of states, φ is strictly decreasing (by the definition of $\widehat{\delta}$), which contradicts the well-foundness of the natural numbers. □

Corollary 2. *Let* $r_0 \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots$ *be a* complete *(finite or infinite) execution of* $\widehat{\mathrm{LTS}}(P)$. *There exists a* complete *(finite or infinite, respectively) execution* $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$ *of* $\mathrm{LTS}(P)$ *such that* $\sigma_1\,\sigma_2 \cdots = a_1\,a_2 \cdots$.

The corollary follows from Corollary 1 and Lemma 2. It is equivalent to $\mathfrak{L}(\widehat{\mathrm{LTS}}(P)) \subseteq \mathfrak{L}(\mathrm{LTS}(P))$, which is the technical formulation of Proposition 1.