On Composing and Proving the Correctness of Reactive Behavior

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

Amir Kantor
Weizmann Institute of Science
Rehovot, Israel
amir.kantor
@weizmann.ac.il

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

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

Lior Mizrahi
Ben-Gurion University
Beer-Sheva, Israel
liormizr
@cs.bgu.ac.il

Gera Weiss
Ben-Gurion University
Beer-Sheva, Israel
geraw
@cs.bgu.ac.il

ABSTRACT

We present a method and a tool for composing a reactive system and for accompanying the development and documentation process with a proof of its correctness. The approach is based on *behavioral programming* (BP) and the Z3 SMT solver. We show how program verification can be automated and streamlined by combining properties of individual modules, specified and verified separately, with application-independent specifications both of the BP semantics and of general theories. The method may yield an exponential acceleration of the verification process when compared with model-checking the composite application. We show that formalization of properties of independent modules in preparation for the correctness proofs can be useful as documentation for future development. We view this work as a further step towards making formal correctness proofs standard practice in the development of reactive systems, and carried out by programmers at large.

1. INTRODUCTION

The development and verification of large scale component-based software systems pose many challenges. During development, programmers working on separate modules may often be unaware of the fine details of inter-module interfaces, or these interfaces may not be well defined; and the *state explosion* problem prevents the model-checking of the entire system, which could discover the resulting errors at the system level.

In recent decades, a prominent approach for tackling these issues has been that of compositional development, based on well defined interfaces, assume-guarantee contracts, and verification. This approach calls for defining the interfaces between modules, programming each module separately, and then verifying that each module guarantees certain properties under certain assumptions on its environment. The modules' verified properties are then combined in order to deduce system-wide properties. For some of the notable compositional approaches see, e.g., [3–7, 9, 10, 12–14, 17–20, 22–25], which are reviewed in Section 5. A recent survey of behavioral interface specification languages [17] points to outstanding research challenges in this area, including how to deal with parallel programs, to tie module specifications to requirements specifications, and to further automate the verification process. This paper aims to contribute to the pursuit of these challenges.

We present a compositional approach and a tool chain for building and verifying reactive systems. In our proposed approach, properties of individual modules are formalized and then used for automated verification of the composite system. The formalized module properties serve also as part of system documentation and can be useful for other development tasks. Two key elements of the approach are (1) a specification and programming formalism that enables programming different aspects of system behavior independently of each other; (2) means to infer composite system properties from formally-specified properties of individual modules.

For system specification we use the formalism of *behavioral programming* (BP) [16] and for inferring system properties from module properties the Z3 SMT solver [8]. We believe, however, that the approach can be based on to other methods too as long as they cater to programming separate aspects in isolation. It can also be used with other inference tools and theorem provers.

Our goal is to improve the process of compositional development, documentation, and verification, by proposing ways that in some cases will give rise to efficient verification. Similarly important is providing tools for formal documentation of the module properties.

The methodology behind the approach consists of the following steps (which can be performed in almost any order):

Specification: Document in natural language each of the desired and undesired aspects of system behavior.

Module Properties: Design the system such that each aspect of the behavior will be implemented by its own separate program module (or a set thereof). Formalize the properties of each such module (or set of modules) as formulas in a solver or proof assistant.

Environment Properties: Similarly, formalize the description of external environment behavior and encode it in the solver or proof assistant.

Composition Properties: Specify the application-independent module-composition rules as formulas of the solver or proof assistant.

Domain Properties: Specify the application-independent domain knowledge in the solver or proof assistant.

Prove System Properties: Use the solver or proof assistant to prove that, given the module, composition and domain properties, the system will behave correctly.

System Implementation: For each independent aspect of the behavior, develop the code of the corresponding module(s). If needed for simulation purposes, also implement the external environment behavior as a separate program module (or a set thereof).

Module Verification: Verify that the individual modules satisfy their properties as stated in the Module Properties step. If the modules are small and simple, this step can be done, e.g., automatically by model-checking, or, by traditional testing techniques.

We present several examples to demonstrate that this technique might yield more efficient verification in some cases, and illustrate the benefits of formally documented properties in software comprehension, reuse and maintenance.

The paper is organized as follows. In Section 2 we recap the basic formal definitions of BP. In Section 3 we show how BP semantics can be formalized as a reusable application-agnostic model of the Z3 SMT solver. In Section 4 we apply the proposed approach to several examples, and discuss its benefits. In Section 5 we briefly review different approaches to compositional system specification and verification. In Section 6 we summarize the paper and discuss future research directions.

2. BEHAVIORAL PROGRAMMING

A behavioral program consists of independent threads of behavior that are interwoven at run time. Each *behavior thread* (abbr. *b-thread*) specifies events and event sequences which, from its own point of view must, may, or must not occur. The execution infrastructure synchronizes and interweaves all behaviors, selecting events that constitute integrated system behavior without requiring direct communication between b-threads. Specifically, at every execution cycle, all b-threads declare events that should be considered for triggering (called *requested events*), events whose triggering they forbid (*blocked events*), and events that they do not actively request but simply "listen-out for" (*waited-for events*). Following these declarations all b-threads synchronize. An event selection mechanism then triggers one event that is requested and not blocked, and resumes all b-threads

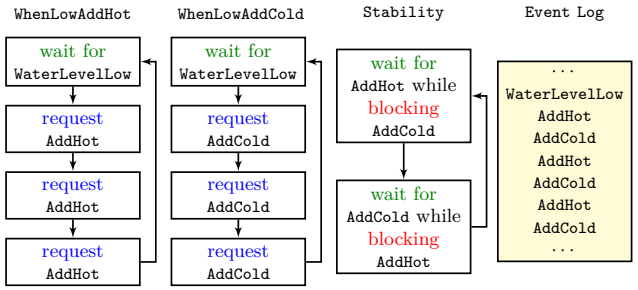

Figure 1: Incremental development of a system for controlling water level in a tank with hot and cold water sources. The b-thread `WhenLowAddHot` repeatedly waits for `WaterLevelLow` events and requests three times the event `AddHot`. `WhenLowAddCold` performs a similar action with the event `AddCold`, reflecting a separate requirement, which was introduced when adding three water quantities for every sensor reading proved to be insufficient. When `WhenLowAddHot` and `WhenLowAddCold` run simultaneously, with the first at a higher priority, the runs will include three consecutive `AddHot` events followed by three `AddCold` events. When a new requirement is introduced that water temperature be kept stable the b-thread `Stability` is added. It enforces the interleaving of `AddHot` and `AddCold` events by using event blocking.

that requested or waited for the event. Figure 1 demonstrates a behavioral application.

In the BP package for Java (BPJ), for example, b-threads are coded by the developer as ordinary Java threads. Synchronization and declaration of requested, blocked and waited-for events is done by calling a method named `bSync`, passing it the three sets of events as parameters.

Behavioral programming facilitates incremental non-intrusive development. For example, in a game-playing program each game-rule and each strategy can be implemented in a dedicated independent b-thread, enabling or forbidding moves. A rule for alternation of player-turns, for example, can be implemented by an independent b-thread that repeatedly blocks one player's move while waiting for a move by the other player, and vice versa. Considering another example program, in the reactive program for stabilizing an aerial vehicle, separate independent b-threads can control UAV altitude and each of the three attitude angles (see, e.g., [16] and references therein).

While the motivation behind BP is natural and intuitive development using almost any programming language [1,16], its underlying infrastructure was formally described and analyzed with formal definitions based on composition of transition systems. Below we recap the definitions as appeared in [15].

A *deterministic labeled transition system* is a quadruple $\langle S, E, \rightarrow, init \rangle$, where S is a set of states, E is a set of events, $\rightarrow$ is a (possibly partial) function from $S \times E$ to S, and $init \in S$ is the initial state. The *runs* of 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 the function $\rightarrow$ maps the pair $\langle s_{i-1}, e_i \rangle$ to s_i, written $s_{i-1} \xrightarrow{e_i} s_i$. The system $\langle S, E, \rightarrow, init \rangle$ is said to be total if the function $\rightarrow$ is total. Behavior threads can be modeled by transition systems, with S and E finite, and with states being labeled

by event sets, as follows.

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

Definition 2. The *runs of a set of b-threads* $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i\rangle\}_{i=1}^{n}$ are 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

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

and

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

The basic BP semantics allows alternative implementations of event selection among all those requested and not blocked, including, e.g., random, prioritized, planned or adaptive.

In this paper we keep to an assumption, introduced in [15], that b-threads do not share data and depend solely on events for input and output. A coding convention that guarantees this assumption allows us to use the tool presented in [15] for explicit model-checking of behavioral code.

3. FORMULATING BP IDIOMS IN Z3

In this section we show how the application-agnostic composition semantics of BP can be defined in Z3 towards serving in compositional proofs.

We begin the definition with the concepts of time and events[1]:

```
Time     = IntSort();
Event    = Datatype('Event')
...
Event    = Event.create()
```

In our implementation, time is discrete and is represented by integers, and in fact refers to the sequence number of an event in a trace. Events are defined as a Z3 data type, followed by the definition and creation of the application-specific event objects.

We then define functions that model the requesting and blocking of events by b-threads, and the resulting trace of triggered events:

```
requested = Function('requested', Event,
                     Time, BoolSort())

blocked   = Function('blocked', Event,
                     Time, BoolSort())

trace     = Function('trace', Time, Event)
```

[1]Most of this work with Z3 was done using the Python API. For readability, the presentation here interchanges and mixes such Python code, formatted Z3 output, and plain mathematical formulations. The code can be found at [2].

The first line states that the function `requested` maps any event and time instant to a boolean flag, which specifies whether or not the event was requested at that time by any b-thread. Similarly, the function `blocked` maps an event and a time instant to a boolean flag that is true if and only if the event was blocked at that time. The `trace` function associates each time instant with the event that was triggered at that instant.

We then model behavioral programming semantics as a property of these functions.

```
∀e, t: trace(t) = e ⇒
       requested(e,t) ∧ ¬blocked(e,t)
```

This rule states that, in order to be triggered at a particular time instant, an event must be requested at that time and must not be blocked.

Since we want to establish the proof by analyzing each of the b-threads in isolation, we introduce a helper function called `requested_by` that takes the same parameters as the function `requested`, with one additional parameter that indicates which b-thread initiated the request. The function `blocked_by` indicates in a similar way which b-thread initiated the blocking of an event at a given time. We then define:

$$\texttt{requested(e,t)} \Leftrightarrow \bigvee_{\texttt{bt}\in\texttt{BThreads}} \texttt{requested\_by(e,t,bt)}$$

and

$$\texttt{blocked(e,t)} \Leftrightarrow \bigvee_{\texttt{bt}\in\texttt{BThreads}} \texttt{blocked\_by(e,t,bt)}.$$

I.e., an event is considered requested (respectively, blocked) if and only if it is requested (respectively, blocked) by some b-thread. The set `BThreads` of participating b-threads is encoded as a Z3 list.

There are various methods that can be applied in event selection, such as priority or planning-based schemes, with which it may sometimes be more convenient to program. The axioms corresponding to these models can also be formulated in Z3. Appendix 1 in the supplementary material [2] contains an example. Our axiomatization presupposes that all executions of the program are infinite, since `trace` is defined to be an infinite sequence of events. However, finite executions can also be dealt with, by adjusting the axioms to include a special `nop` event that is only triggered when no other events are enabled.

4. EXAMPLES

In this section we demonstrate the application of the method outlined in the introduction to several examples. In each example, we specify module and system properties, prove the latter given the former, and, when applicable, verify that the individual implemented modules indeed satisfy their properties. The source code for all examples (b-threads and Z3 code) is available online at [2]. We then discuss how the basic proof of correctness of the application also results in opening the way to possible acceleration of the proof when compared to explicit model checking, and in benefits in documentation.

4.1 Counting with small orthogonal modules

Before going into more practical examples, we describe a small example that highlights how domain knowledge that

leads to a particular design can be known to and used by the SMT solver, leading to efficient verification of a composite reactive program.

Let $p_1, p_2 \ldots, p_n$ be n large prime numbers, and let $N = \prod_{i=1}^{n} p_i$. Let E_0 and E_1 be two events, and consider the ω-regular language $L_N = \left((E_0 + E_1)E_1^{N-1}\right)^{\omega}$. Thus, in every run E_0 can only be triggered at times that are divisible by N, and E_1 may always be triggered.

Our goal is to create a behavioral program that generates L_N and prove its correctness. For $i = 1, 2, \ldots, n$ consider the b-threads $BT_1, \ldots, BT_n$ and BT_{gen} defined by the following pseudo-code:

```
BT_i  for i ∈ {1,2,...,n} {
      for(;;) {
           bSync(wait for {E_0,E_1});
           for(j=0; j<p_i-1; j++)
           bSync(wait for E_1 while blocking E_0);
      }
}

BT_gen {
      for(;;) {
           bSync(request {E_0,E_1});
      }
}
```

In words, BT_i blocks E_0 at all time instants that are not divisible by p_i, and BT_{gen} requests both E_0 and E_1 at all synchronization points. We now prove that together these b-threads generate L_N.

We first express the properties of each of the b-threads separately (shown here for $n = 2$, with $p_1 = 3$ and $p_2 = 7$):

```
∀t, e: ((t%3≠0) ⇔ blocked_by(E_0, t, BT_1)) ∧
       ¬blocked_by(E_1, t, BT_1) ∧
       ¬requested_by(e, t, BT_1)

∀t, e: ((t%7≠0) ⇔ blocked_by(E_0, t, BT_2)) ∧
       ¬blocked_by(E_1, t, BT_2) ∧
       ¬requested_by(e, t, BT_2)

∀t, e: requested_by(e, t, BT_gen) ∧
       ¬blocked_by(e, t, BT_gen)
```

The first formula says that BT_1 blocks E_0 if and only if t is not a multiple of 3, that it never blocks E_1, and that it never requests any event. The second formula states similar properties for BT_2 with the difference that it blocks E_0 at times not divisible by 7 instead of 3. The third formula captures the properties of BT_{gen}, namely, that it always requests both events and never blocks either of them. Note that the states and transitions of the b-threads are not explicitly modeled in this case, and that the Z3 formulas also cover what the modules *do not* do, i.e., do not block or request, which is needed for our application-agnostic composition.

As these b-thread properties concentrate only on what b-threads request or do not request and what they block or do not block, and not on which events are actually triggered, each b-thread's properties can be verified by model-checking the b-thread in isolation from the rest of the program. This method of model-checking relies on the abstraction of a b-thread code as consisting of atomic transitions between synchronization points, in which requests and blocked events are declared. As each of the first n b-threads has p_i states and BT_{gen} has a single state, model-checking the individual b-threads entails examining a total of $1 + \sum_{i=1}^{n} p_i$ states. In contrast, explicit model-checking of the entire system with all b-threads would have to traverse all the $\prod_{i=1}^{n} p_i$ reachable states in the product transition system.

When we add these properties to the Z3 model described in the preceding section, we see that Z3 can indeed quickly verify that the system satisfies its desired property. Namely, that E_0 is only enabled at times divisible by N (21 in this case), and that E_1 is enabled at all times.

```
∀t: requested(E_0,t) ∧ ¬blocked(E_0,t) ⇔ t%21==0
∀t: requested(E_1,t) ∧ ¬blocked(E_1,t)
```

The duration it takes Z3 to verify this property is affected only negligibly by an increase in the p_i values. This illustrates the fact that the verification is performed with the aid of additional arithmetical knowledge and not by traversing the entire state space. It yields (in this case) the ultimate desired property of compositional verification – establishing correctness based on proving individual modules separately, without explicitly model-checking the product transition system.

Observe that the described Z3 formulation can also be used to prove *liveness* properties. For instance, in order to prove the property "event E_0 is triggered infinitely often" we would follow the same steps described above, formulate in Z3 the property that E_0 is enabled infinitely often (at intervals of 21 steps), and let Z3 prove it. Then, using reasonable fairness assumptions, the liveness property can immediately be deduced. Liveness property verification is also supported by verifying liveness properties of individual b-threads or group thereof using the BPJ model checker.

This example also shows the power of the blocking idiom in BP. Specifically, if we remove the ability of a b-thread to block events, we can prove (to be published separately) that one must then use at least one b-thread whose size is exponentially larger than the size of the b-threads proposed here. This shows that, in our setting, blocking may allow for an exponential saving in the size of the state space needed for verification, thus accelerating verification when appropriate compositional techniques exist.

4.2 Simulating constrained movement

Consider an application simulating movement of a particle in a two-dimensional grid, as follows. The grid has $(2n+1)^2$ points, with coordinates $\langle x, y\rangle$ where $-n \leq x, y \leq n$. Initially, the particle is at the center of the grid, at point $\langle 0, 0\rangle$. In each simulation step, the particle moves randomly from its then-current position to one of its four neighbor positions. Apart from the particle process, the system includes processes that make areas of the grid inaccessible to the particle. For example, we analyze the case where each such inaccessible area can be described by a continuous function $f(x)$ such that point $\langle x, y\rangle$ is inaccessible if and only if $y \geq f(x)$ (alternatively, $y \leq f(x)$). In other words, the area above (or below) the curve $y = f(x)$ is inaccessible.

The goal is to discover compositionally whether the inaccessible areas jointly prevent the particle from reaching the grid's boundaries.

Each process b-thread may have a rich behavior and a complex transition system unrelated to the particle movement, where the constraining of the particle movement may be only a side-effect of the behavior. This complex behavior is abstracted here by a b-thread with ℓ states, visited sequentially in a cycle, such that the constraining effect is true in all of them. The approach that we use may be applied also to more complex processes with branches in the transition system and varying sets of blocked events.

The pseudo-Java code of the particle b-thread is:

```
x = 0; y = 0;

for (;;) {
    bSync(Request the moves:
          Move(x+1,y),Move(x-1,y),
          Move(x,y+1),Move(x,y-1));

    /* The triggered Move event is returned
    in bp.lastEvent */
    x = bp.lastEvent.x; y = bp.lastEvent.y;
}
```

Observe that the particle b-thread is "unaware" of movement constraints imposed by either the grid or the process threads.

A b-thread corresponding to a process that forbids particle movement into the region $y \geq f(x)$, and has a single cycle of ℓ states, is:

```
state = 0
for(;;) {
    bSync(Wait for all events,
          while blocking moves to all
          ⟨x,y⟩ s.t. y ≥ f(x));

    state = (state+1) % ℓ;
}
```

A direct approach to verifying this application is to span its entire state graph, and to check that there is no reachable state where x or y equals $\pm n$. For example, in explicit model-checking of the composite application the number of states that will be visited is on the order of $n^2 \cdot \prod_{bt \in P} \ell_{bt}$, a quantity that grows exponentially with the size of the set P of all process b-threads.

By contrast, the compositional approach that we suggest is to model-check each process b-thread separately (with all its internal dynamics), and to employ Z3 for the compositional part. For example, the relevant properties of the particle b-thread `ParticleBT` are coded in Z3 as:

```
∀e, t: ¬blocked_by(e, t, ParticleBT)

∀x, y:
    requested_by(Move(x,y), t, ParticleBT) ⇔
    (trace(t-1).x = x-1 ∧ trace(t-1).y = y) ∨
    (trace(t-1).x = x ∧ trace(t-1).y = y-1) ∨
    (trace(t-1).x = x+1 ∧ trace(t-1).y = y) ∨
    (trace(t-1).x = x ∧ trace(t-1).y = y+1)
```

and the properties of each process b-thread $\texttt{Process}_i$ (associated with function $\texttt{f}_i$) are coded as:

```
∀e, t   : ¬requested_by(e, t, Process_i)

∀x, y, t: blocked_by(Move(x,y), t, Process_i) ⇔
          y >= f_i(x)
```

Observe that this formulation holds for all ℓ_i states of the process b-thread, and there is no need to articulate properties of individual states.

Another Z3 formula (not shown) specifies the initial position of the particle. Finally, to define what we want Z3 to prove, we state the (undesired) property that the particle does reach the grid boundaries:

```
∃t: trace(t).x = n ∨ trace(t).x = -n ∨
    trace(t).y = n ∨ trace(t).y = -n
```

We then run Z3 to check that this model is unsatisfiable, taking advantage of Z3's knowledge of arithmetic to deduce that the inaccessible zone, as defined by the properties of the $\texttt{Process}_i$ b-threads, renders the edges of the grid unreachable.

It now remains to be verified that the original b-threads uphold the properties that we have encoded in Z3. Fortunately, this can be performed for each b-thread separately, without composing them — by using either static analysis or the BPJ model-checker [15]. In the latter case, each process b-thread is checked, along with a b-thread that repeatedly requests all possible movements into all $(2n+1)^2$ points of the grid. The property to be verified is that the single b-thread blocks movements into coordinates where $y \geq f(x)$. The particle b-thread is also verified separately, to ensure that successive points in the movement are always connected by a single grid edge.

Table 1 shows the savings when model-checking each process behavior and the particle behavior separately, as compared to checking the movement of the particle with all behaviors together. In this example, we set $n = 20$ (resulting in a 41×41 grid), and chose four processes with forbidden zones that prevent the particle from venturing outside the quadrilateral with vertices $\langle 15, 15\rangle$, $\langle -18, 16\rangle$, $\langle -19, -19\rangle$, $\langle 17, -18\rangle$. The movement-constraining processes have $2, 3, 5$ and 7 internal states. The run time improvement is evident.

For a similar setting with $n = 10^{17}$, it took Z3 approximately 7 seconds to reach a conclusion. However, a related test run, in which one of the constraints was omitted and the resulting model is satisfiable did not terminate (in a reasonable amount of time). We believe that this is a technical issue in our implementation, and not a fundamental problem in the underlying approach. Future work will include optimizing our implemented model to better fit Z3's constraints, as well as leveraging future enhancements of Z3. Note that our present model does permit one to use Z3 to verify, within a fraction of a second, that a given trace that reaches the grid boundaries is valid.

We now demonstrate how a formalization of properties of the modules in Z3 supplements the code with documentation that is useful beyond the verification process, for tasks such as module reuse or enhancement.

Consider a requirement, which arrives from the user after the system is up and running, to expand the application to a three-dimensional setting. That is, the grid is extended to 3D and the original requirement that the particle is constrained within a box around the origin and cannot reach the grid boundaries, remains, but now is interpreted in 3D. After adding an attribute to the `Move` event that gives the z axis position and adding to the particle b-thread movements in the z direction, the question we ask next is how to enhance the b-threads for the processes that constrain the particle movement.

In a standard development process, without the Z3 formulation, a programmer wanting to reuse or enhance existing modules for the new requirement would need to check their code directly. While the code in our example is simple, the code of the movement-constraining b-threads may be complex, and the relevant properties may not readily emerge.

With the Z3 formulation, the contemplation of how to extend the system can be done in the context of the high-level theory. In this case, the Z3 code explicitly talks about the lines that form a closed polygon contained in the boundary of the two-dimensional grid. When we formulate in Z3 a 3D extension of the properties, we can start by checking if the current modules and formulated properties already satisfy

Table 1: Comparing the monolithic approach to the compositional one. Rows 1 and 2 describe checking each of the b-threads separately using model-checking (MC); row 3 describes the compositional step (using Z3); and row 4 summarizes the total cost of the compositional approach. The last row of the table describes model-checking the entire system, as a single unit.

Checked entity	Number of states	Method of checking	Run time (sec.)
Four process b-threads	2,3,5,7	MC	160 (total)
Particle	1681	MC	4
Compositional step	—	Z3	0.03
Total compositional proof	1698	MC+Z3	164.03
MC of Entire system	119385	MC	426

the new requirement. If not, Z3 gives us a counterexample that we can use to guide the development. From the counterexample we may realize that, in 3D, the b-threads form infinite walls in the z dimension rising from the edges of the 2D polygon and that it is sufficient to add a "floor" and a "ceiling". More generally, we see that the boundaries can be avoided by forming a set of planes in the 3D space that form a polytope that contains the origin and is contained within the bounding box.

The role of Z3 in this process is to help the designers identify and document all the properties of the b-threads that are relevant to the requirements. When a property is missing (e.g., if we forget to mention that the polytope contains the origin) Z3 presents a counterexample, from which the missing properties may emerge. When Z3 proves that all the requirements are satisfied, we know that we have documented all the required properties of the b-threads. The completeness of the documentation of the properties of the b-threads is important, for example, when we want to replace a b-thread.

Once a set of sufficient properties is established in Z3, the implementation can proceed in different directions: some properties may already exist in the current modules (but are not documented because they were not relevant to the 2D case), others may be implemented as changes to existing modules, and yet others may be added as new independent b-threads. The implemented modules can then be model-checked to verify that they satisfy the properties and, if so, we can conclude that the application is correct.

4.3 A job scheduler

The following example demonstrates the development and compositional verification of a scheduling algorithm using behavioral programming. The program is actually incremental in nature: when new b-threads are added, the program can be verified without rechecking existing processes.

The problem is defined as follows. A scheduler needs to assign time slots for each of k processes $P_1, \ldots, P_k$. Each process P_i is associated with two parameters, m_i and n_i, meaning that it requires the assignment of m_i slots in each cycle of n_i slots. Put differently, process P_i needs to be assigned m_i slots in cycle $\{kn_i + 1, kn_i + 2, \ldots, (k+1)n_i\}$, for all $k \in \mathbb{N} \cup \{0\}$.

A schedule that satisfies all these constraints exists if and only if $\sum_{i=1}^{k} (m_i/n_i) \leq 1$, in which case an *earliest-deadline first* (abbr. EDF) policy will guarantee that none of the conditions are violated (see, e.g., [21]).

We suggest the following BP implementation. Given an instance of the problem, $\langle m_i, n_i \rangle_{1 \leq i \leq k}$, we program a b-thread for each process, presenting m_i requests in each cycle of n_i slots in the form of events $R(bt, j)$, where bt is the b-thread's identity and $1 \leq j \leq n_i$ is the number of slots (scheduling opportunities for this process) before the present cycle ends. The scheduler is implemented in BP by having all b-threads that have not yet been assigned sufficient slots in the present cycle block all event requests with a higher value of j. An additional b-thread, `Idle`, continuously requests the special event $R(idle, \infty)$ that is triggered only when no process requests any event in the slot.

At each behavioral synchronization point, one of the requested events is triggered, indicating that the requesting process is assigned the present slot. All b-threads are notified when an event is triggered and can then request to be scheduled in the next slot as needed. The b-threads for each of the processes can be modeled in BPJ as follows:

```
BT⟨m_i,n_i⟩ for i ∈ {1,...,k} {
    for(;;) {
        count = 0;
        for(j=n_i; j>0; j--) {
            if(count < m_i) {
                bSync(request R(i,j),
                      block all events R(s,t)
                      such that t > j,
                      wait for all events);
                if(lastEvent == R(i,j))
                    count++;
            }
            else {
                bSync(wait for all events);
            }
        }
    }
}
```

Without loss of generality, we assume that $\forall i, m_i = 1$ (for $m_i > 1$, as we can substitute the original process by m_i processes with parameters $\langle 1, n_i \rangle$ each). The b-thread for process P_i then has $O(n_i)$ states. Consequently, exhaustive verification of the application entails inspecting $O(\prod_{i=1}^{k} n_i)$ states (in the worst case, assuming the n_i's are pairwise mutually prime).

A compositional alternative is to verify each b-thread separately, to ensure that it constantly blocks all requests by processes with further-away deadlines than its own — until its scheduling quota has been filled. This can be accomplished by inspecting only $O(\sum_{i=1}^{k} n_i)$ states. If these properties hold, then EDF scheduling is guaranteed, and it only remains to check that $\sum_{i=1}^{k} (m_i/n_i) \leq 1$, which can be done manually, or using a calculator.

All in all, this approach yields much shorter verification times. Further, when adding a new process $\langle m_{k+1}, n_{k+1} \rangle$ at a later time, one need not repeat the verification of the original b-threads (assuming an upper bound on the number of threads and their cycle lengths). It suffices to check that the new b-thread adheres to its responsibility in the EDF

policy (by requesting and blocking events correctly), and then verify that $\sum_{i=1}^{k+1}(m_i/n_i) \leq 1$.

Another alternative solution we explored entails modeling the properties of the b-threads in Z3, and having the tool check whether a legal schedule exists in a model that includes all of them. In this approach, the properties of b-thread i with parameters $\langle 1, n_i \rangle$ are as follows:

```
∀e, t: requested_by(e, t, BT_i) ⇔
       (e=(BT_i, n_i-(t-1)%n_i) ∧
       ¬already_scheduled(BT_i, t))

∀e, t: ¬already_scheduled(BT_i, t) ⇒
       blocked_by(e, t, BT_i) ⇔
       deadline(e) > n_i-(t-1)%n_i

∀e, t: already_scheduled(BT_i, t) ⇒
       ¬blocked_by(e, t, BT_i)
```

where, as in the BPJ implementation, events consist of the requester's identity and the time remaining until its deadline, and the functions `deadline` and `requester` are used to retrieve the respective parameters. The helper function `already_scheduled` evaluates to true if and only if b-thread BT_i has already been scheduled in the cycle to which time t belongs.

The verification is then performed by giving Z3 the undesired property that one of the processes is not scheduled in some cycle, and having it prove that the model then becomes unsatisfiable. The property is given as:

```
∃BT_i, t_1: ∀t_2: t_1·n_i <t_2 ≤(t_1+1)n_i ⇒
                 requester(trace(t_2)) != BT_i
```

As the property to be proved is algebraic in nature, we expected the Z3 verification process to readily display superior performance as compared with explicit model checking using the BPJ model checker. Unfortunately, that was not the case, and the running time grew exponentially with the number of processes. We believe that our implementation can be improved and the running time greatly decreased, but we leave this for future work. Despite being applicable only to programs with few processes, our current model is still useful: it demonstrates that the set of thread properties we have identified is complete, and that it suffices for proving the correctness of the system. This indicates that we have documented any hidden assumptions about the various modules, and facilitates their redesign or reuse.

4.4 Dining philosophers

In this example we demonstrate a direct approach to encoding b-threads in Z3 by capturing their transition systems and the requested and blocked events in each state.

Consider for example a BP model for the famous dining philosophers problem[2]. Assume that this abstract problem is a specification for a larger BP application, e.g., a circle of industrial robots where each two adjacent ones share a tool, and each robot requires both its adjacent tools to perform its task. This behavior of the robots can be specified in BP as follows: there is a b-thread per tool (fork), with two states — "up" (`fork_state` is true) and "down" (`fork_state` is false), and a b-thread per robot (philosopher), with its four states known as the fixed cycle of "thinking", "picked up one fork", "eating", and "put down one fork". The events are of the form `E(i,j,up)` or `E(i,j,down)` and represent "philosopher i picked up (or put down) fork j" for $0 \leq i \leq n$, and $j = i$ or $j = (i+1) \bmod n$. All philosophers but one are right-handed (they first pick up the fork on their right) and one is left-handed. Each fork thread blocks events that pick it up when in the "up" state and events that put it down when in the "down" state, without ever requesting events.

We proceed to explain the transition system by formulating its properties in Z3 as part of a proof that the industrial robotic application satisfies its specification and is deadlock-free. Below we describe parts of a model for a system with eight philosophers (hence, e.g., `(fo+1)%8` is the index of the fork next to fork `fo` (and the philosopher of same number) in cyclic order):
Fork b-threads never request events:

```
∀e, t, fo : ¬requested_by(e, t, Fork(fo))
```

A b-thread for a fork that is down blocks the events of putting the fork down again (and only these):

```
∀t, fo:
  ¬fork_state(fo, t) ⇒
  (blocked_by(E(fo, fo, down), t, Fork(fo)) ∧
   blocked_by(E((fo+1)%8, fo, down),
              t, Fork(fo)) ∧
   (∀e1:
     blocked_by(e1, t, Fork(fo)) ⇒
     e1 = E(fo, fo, down) ∨ e1 = E((fo+1)%8,
                                   fo, down)))
```

A b-thread for a fork that is up blocks the events of picking the fork up again (and only these):

```
∀t, fo:
  fork_state(fo, t) ⇒
  (blocked_by(E(fo, fo, up), t, Fork(fo)) ∧
   blocked_by(E((fo+1)%8, fo, up),
              t, Fork(fo)) ∧
   (∀e1:
     blocked_by(e1, t, Fork(fo)) ⇒
     e1 = E(fo, fo, up) ∨ e1 = E((fo+1)%8,
                                 fo, up))
```

The state of the fork changes according to the pick-up/put-down actions of the philosophers on the right or left of the fork (and only these) :

```
(∀t, fo:
  trace(t) = E(fo, fo, up) ∨
  trace(t) = E((fo+1)%8, fo, up) ⇒
  fork_state(fo, t+1)) ∧
(∀t, fo:
  trace(t) = E(fo, fo, down) ∨
  trace(t) = E((fo+1)%8, fo, down) ⇒
  ¬fork_state(fo, t+1)) ∧
(∀t, fo:
  ¬(trace(t) = E(fo, fo, up) ∨
    trace(t) = E((fo+1)%8, fo, up) ∨
    trace(t) = E(fo, fo, down) ∨
    trace(t) = E((fo+1)%8, fo, down)) ⇒
  fork_state(fo, t) = fork_state(fo, t+1))
```

Once these properties are formulated, system properties can be proven. In our case, Z3 can verify that the system does not deadlock, i.e., that there is always an event that is requested and not blocked in all executions of the program. Z3 does this in under 10 seconds. This verification is performed using a slightly modified version of the axioms presented in Section 3, which considers both finite and infinite executions of the program.

Note that the robotic implementation may be very different from the specification. Still, to verify that each property

[2]There are n philosophers sitting around a table. There is a fork between each two adjacent philosophers. To eat, a philosopher needs to hold both of her adjacent forks.

in the specification holds it should suffice to model-check exhaustively only a few robots and tools.

4.5 Tic-Tac-Toe

In this example we demonstrate the use of Z3 to verify a slightly larger example, highlighting that the properties of the individual modules are quite independent of each other, and refer to the basic specification of the system. We illustrate our technique on the b-threads of the Tic-Tac-Toe game application presented in [15]; we briefly summarize the application's features in a description taken from [15], and refer the reader to that paper for a more detailed explanation of the application itself.

In the (classical) game of Tic-Tac-Toe, two players, X and O, alternately mark squares on a 3×3 grid whose squares are identified by $\langle row, column \rangle$ pairs: $\langle 1,1 \rangle, \langle 1,2 \rangle, \ldots, \langle 3,3 \rangle$. The winner is the player who manages to form a full horizontal, vertical or diagonal line with three of his/her marks. If the entire grid becomes marked but no player has formed a line, the result is a draw.

In our example, player X is played by a human, and player O is played by the application. Each move (marking of a square by either player) is represented by a matching event, $\texttt{X}_{\langle row,col \rangle}$ or $\texttt{O}_{\langle row,col \rangle}$. The events `XWin`, `OWin` and `Tie` represent the respective victories and a draw. A play of the game may be described as a sequence of events. E.g., the sequence $\texttt{X}_{\langle 1,1 \rangle}$, $\texttt{O}_{\langle 2,2 \rangle}$, $\texttt{X}_{\langle 3,2 \rangle}$, $\texttt{O}_{\langle 1,3 \rangle}$, $\texttt{X}_{\langle 3,0 \rangle}$, $\texttt{O}_{\langle 2,1 \rangle}$, $\texttt{X}_{\langle 3,3 \rangle}$, `XWin` describes a play in which X wins, and whose final configuration is:

The BP implementation of the game as described in [15] contains two types of b-threads: game rules and strategies. Examples for game rule b-threads are the `SquareTaken` thread that blocks further marking of squares already marked by X or O, and the `EnforceTurns` thread that alternately blocks O moves while waiting for X moves, and vice versa (we assume that X always plays first).

Strategy b-threads are responsible for helping the program to play "wisely" — that is, to contribute towards ensuring that the program does not lose the game. An example for one such b-thread is `PreventThirdX`: when it notices two Xs in a line, it requests the marking of an O in the third square of this line (to prevent an immediate loss).

If neither player makes any mistakes, a Tic-Tac-Toe game ends in a draw. Therefore, we consider our game playing application to be achieving its goals if it never loses the game — namely, if the event `XWin` is never triggered in any run. In [15], this property was verified via explicit model-checking of the Java application with concurrent execution of all b-threads. By contrast, in the present work the proof of correctness begins with the properties of the b-threads as may be verified individually, or as may be planned or designed in early development stages. Z3 is used to verify that these properties, when composed, yield the desired results.

In our proposed Z3 formulation, each event has three fields: x, y and *type*. The *type* field can have values `X,O,X_WIN,O_WIN` and `TIE`. If the event is of one of the first two types, the x and y fields hold the row-column coordinates of the move; otherwise, these fields are meaningless.

As in the previous examples, we formulate the properties of the various b-threads as Z3 code. For instance, the `EnforceTurns` thread, BT_{et}, is formulated as:

```
∀t, e: (t==1) ⇒
       blocked_by(e,t,BT_et) ⇔ e.type() == O

∀t, e: (t>1) ⇒ blocked_by(e,t,BT_et) ⇔
       (e.type()==O ∧ trace(t-1).type()==X) ∨
       (e.type()==X ∧ trace(t-1).type()==O)
```

The code states that in the first move ($t == 1$) the b-thread blocks all of O's moves, and that in subsequent moves the b-thread blocks the player who played last.

Next, we see how the `PreventThirdX` b-thread, BT_{ptx}, translates into Z3 code:

```
∀t, e: requested_by(E(row,col,O), t, BT_ptx) ⇔
  ∃t1,t2: t1 < t ∧ t2 < t ∧
      trace(t1).type()==X ∧
      trace(t2).type()==X ∧
      ((trace(t1).x() == trace(t2).x() == row) ∨
       (trace(t1).y() == trace(t2).y() == col) ∨
       ((trace(t1).x() == trace(t1).y()) ∧
        (trace(t2).x() == trace(t2).y()) ∧
        (row == col)) ∨
       ((trace(t1).x() + trace(t1).y() == 4) ∧
        (trace(t2).x() + trace(t2).y() == 4) ∧
        (row + col == 4)))
```

The above code indicates that the b-thread only requests an O move in square row, col if: (1) X has already marked two squares in that row, or (2) X has already marked two squares in that column, or (3) The square is part of the main diagonal ($row == col$), and X already has two squares of that diagonal, or, (4), if the square is part of the secondary diagonal ($row + col == 4$), and X has already marked the two other squares of that diagonal.

Due to the larger extent of this example, we omit the remaining b-threads. The code is available online at [2].

Once all the other rule and strategy b-threads have been translated into Z3 in a similar fashion, we had the tool prove the desired property, namely that O can never lose:

```
∀t: trace(t).type() != X_win
```

Z3 replied in the affirmative. Further, running the same test with one of O's strategy b-threads omitted resulted in a failure. Printing the Z3 model, the listing of the function *trace* reveals a counter-example scenario in which X wins:

```
[1 → TraceEntry(E(3, 3, X)),
 2 → TraceEntry(E(2, 2, O)),
 3 → TraceEntry(E(1, 1, X)),
 4 → TraceEntry(E(1, 3, O)),
 5 → TraceEntry(E(3, 1, X)),
 6 → TraceEntry(E(2, 1, O)),
 7 → TraceEntry(E(3, 2, X)),
 else → TraceEntry(E(1, 1, X_WIN))]
```

Apart from enabling us to prove the desired property, we observe that formulating the b-thread's properties as Z3 axioms also provides a more precise documentation than natural-language requirements, as well as a useful abstraction of program code. For instance, the Z3 code for the the `EnforceTurns` b-thread (displayed above) states explicitly, and thus documents, the fact that player X plays first, and that neither player can make two consecutive moves. For comparison, the (pseudo) Java code of this b-thread is:

```
for(;;) {
    bSync(block O moves, wait for X moves);
    bSync(block X moves, wait for O moves);
}
```

In the Z3 code, the reader can interpret each formula separately. Even if a formula is long — its scope is well defined and it is always complete. When reading program code like the above the reader has to mentally follow the flow of the `for` loop, and the instructions within it and translate them into conditions and possible scenarios. We find that the combination of the natural scenario-oriented program code with the precise yet abstract Z3 properties complement each other in development, verification and maintenance processes. If modules from this application are to be used in another application, or in an enhanced version of the same application, the developers can readily see whether the existing code supports, e.g., more than two players, changing the order of player moves, or allowing a player two consecutive moves under some conditions.

5. RELATIONSHIP TO OTHER WORK

Much research on compositional and modular verification has been conducted in recent years. While most proposed approaches are similar in their underlying assume-guarantee framework, they differ in several aspects: the modeling formalisms (for both program and specification), the way assumptions are inferred (manual or various automatic variants), and the type of reasoning used to deduce the desired system-wide property from the module properties. In this section we review some of these approaches.

In [24], [20] and [25], the authors study assume-guarantee proof rules for parallel programs, where communications or interference between programs are via messages (in [24]) or shared variables (in [20, 25]). Our focus is also on parallel programs, but in our work the components (b-threads) do not communicate directly with each other, but rather use the simple protocol offered by BP semantics. In addition to providing a concise interface for interweaving independent modules that represent separate facets of behavior, the protocol also allows for a reduction in the size of the state-space, as we are only interested in the state of a b-thread when it is at a synchronization point.

As shown in Section 2, while BP is oriented towards programming in standard languages, composition in BP is event-based and may be formalized in terms of finite-state transition systems. System composition and modular verification in such finite-state settings were described in [22] with the introduction of I/O automata, in [14] in the context of a subset of CTL, in [7] using interface automata, and in the research on the behavior-interaction-priority formalism (BIP), see, e.g., [4]. Our work can be viewed as contributing towards applying these methods in programming contexts and towards making the application of formal methods more accessible to programmers. In line with this goal, it would be interesting to explore compositional verification of behavioral programs based on properties formalized as assertions within the code, using behavioral interface languages such as JML, SPEC#, SPARK, separation logic, and Dafny. See [17] for a survey.

The difficulty in formalizing environment behavior from the point of view of a single module is tackled in [5]. The authors verify individual parallel modules, together with interface processes that represent a module's dependencies on its environment, but which can be simpler than the full composite behavior of the environment. The interface modules are derived from the specification of the other modules. In [19], the authors describe assume-guarantee reasoning using iterative abstraction and refinements of the assumptions. In [12], the environment assumptions of a thread are automatically inferred and are abstracted from behaviors of the other threads. In [13], the authors present techniques for automatically decomposing the verification problem and generating component assumptions based on design-level artifacts. In another approach [3], a learning algorithm is used to infer the assumptions.

In our proposed setting, the strict interface through which modules communicate (that is, the events they request, wait-for and block) facilitates integrating assumptions about the environment into the verification process. Particularly, we have shown here that it is often straightforward to represent the environment by dedicated b-threads. This process is demonstrated in Section 4.5, where all strategies available to the environment — the X player — are represented by a simple b-thread. From the verifier's point of view, there is no difference between that b-thread and the actual program's b-threads.

Applying model-checking to the goal of establishing low-level properties and then using semi-automated high-level analysis also helps tackle the state-explosion problem. For example, in [23] the authors verify hardware systems using reasoning that is performed by a proof assistant, while the generated subgoals are verified by model-checking. In [6], the authors show that finding a decomposition that yields benefits in compositional verification is not easy and may not always be possible. In this context, one of the key goals of our present work is verifying modules and composing systems based on artifacts and properties that are aligned with the specifications. Similar approaches appear in [18] for causal behavioral obligations of classes and instances, in [9] for interference and cooperation of aspects, and in [10] for components as part of research in the field of *component-based software engineering*.

6. DISCUSSION AND CONCLUSION

We have shown how BP and the Z3 SMT solver can be used together for composing a reactive system from relatively independent modules, while accompanying the development with a proof of system's correctness.

As mentioned in Section 5, a major issue in compositional verification is automatically generating component properties. We address this by using the requirements that individual modules satisfy as these properties, thus leveraging the intuition the programmers used in building the modules. Using BP to code the modules ensures that module interfaces are always well defined (per the BP semantics), and it tends to produce modules with properties that are relatively self-standing. Consequently, apart from streamlining verification, the resulting module properties are of value for maintenance and debugging tasks. We believe that this approach has potential, as it bypasses the intricate task of looking for compositional properties in composite code — allowing the programmer to focus on module properties more than on inter-module relationships.

When the effects of module-to-module interaction are dependent on a domain theory that is known to the SMT

solver, an opportunity emerges for improving the efficiency of the automated verification. This is because system properties may be inferred directly from module properties, without explicitly examining all states of the composite application.

A key issue that we have encountered is the difficulty of formulating module properties. Per our methodology, component properties have to be formulated twice: once as SMT solver axioms for the compositional part, and once as model-checker properties to be proven on individual threads. In our examples, the first part was often time-consuming: it took some effort to formulate properties that appeared to us natural and aligned with system's requirements in ways that allowed Z3 to handle the proof in reasonable time. This is in line with [11], which suggests that the practical impact of compositional methods is constrained by the amount of non-trivial human input required for defining appropriate assumptions. Interestingly, we found that the second part — translating the Z3 properties into model-checker properties — was almost trivial, as the properties were typically simple postulations on the states of the threads. In the future we plan to automate this transformation.

Despite its difficulties, we found that the process of refining formal properties was instrumental to our understanding and to the corresponding documentation of module behavior. Our conclusion is that, given the right tools, programmers and designers may find the property formalization and automated verification processes beneficial.

Future research directions include developing IDE support for automated proofs of behavioral applications, guidelines for formulating module properties, and possible enhancements to Z3 (or an alternative solver). Another important direction is proving that our methodology fits industry practice, by applying it to a real large-scale system. A good example would be the industrial modeling system SCADE [26], which allows programmers to model systems as reactive state machines and data flows.

Acknowledgments

We thank M. Vardi for his vision and suggestions, and N. Bjørner and R. Lampert for their contributions. The research of Harel, Kantor, Katz and Marron was supported by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013) and by an Israel Science Foundation grant. The research of Mizrahi and Weiss was supported by the Lynn and William Frankel Center for CS at Ben-Gurion University and by a reintegration (IRG) grant under the European Community's FP7 Programme and by an Israel Science Foundation grant.

7. REFERENCES

[1] Behavioral programming. `http://www.b-prog.org`.

[2] Supplementary material. `http://www.wisdom.weizmann.ac.il/~bprogram/emsoft13/`.

[3] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV*, 2005.

[4] S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, 2008.

[5] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *LICS*, 1989.

[6] J.M. Cobleigh, G.S. Avrunin, and L.A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *ISSTA*, 2006.

[7] L. De Alfaro and T. A. Henzinger. Interface automata. In *ESEC*, 2001.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[9] C. Disenfeld and S. Katz. A closer look at aspect interference and cooperation. In *AOSD*, 2012.

[10] R. Dong, J. Faber, Z. Liu, J. Srba, N. Zhan, and J. Zhu. Unblockable compositions of software components. In *CBSE*, 2012.

[11] M.B. Dwyer, J. Hatcliff, R. Robby, C.S. Pasareanu, and W Visser. Formal software analysis emerging trends in software model checking. In *FOSE*, 2007.

[12] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, 2003.

[13] D. Giannakopoulou, C.S. Pasareanu, and J.M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *ICSE*, 2004.

[14] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16:843–871, 1994.

[15] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-checking behavioral programs. In *EMSOFT*, 2011.

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

[17] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Computing Surveys (CSUR)*, 44(3):16, 2012.

[18] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. *ACM SIGPLAN Notices*, 25(10):169–180, 1990.

[19] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV*, 1998.

[20] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Tran. on Prog. Lang. and Sys. (TOPLAS)*, 5(4):596–619, 1983.

[21] J.Y.T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3):115–118, 1980.

[22] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, 1987.

[23] K. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[24] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Eng.*, SE-7(4):417–426, 1981.

[25] A. Pnueli. In transition for global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, pages 123–144. 1985.

[26] Esterel Technologies. SCADE suite. `http://www.esterel-technologies.com/products/scade-suite/`.