An Initial Wise Development Environment for Behavioral Models

David Harel, Guy Katz, Rami Marelly and Assaf Marron
Dept. of Computer Science and Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel
{*dharel, guy.katz, rami.marelly, assaf.marron*}*@weizmann.ac.il*

Keywords: Behavioral models; interactive development; proactive analysis; reactive models; wise computing.

Abstract: We present a development environment that proactively and interactively assists the software engineer in modeling complex reactive systems. Our framework repeatedly analyzes models of the system under development at various levels of abstraction, and then reasons about these models in order to detect possible errors and to derive emergent properties of interest. Upon request, the environment can then augment the system model in order to repair or avoid detected behavior that is undesired, or instrument it in order to monitor the execution for certain behaviors. Specialized automated and human-assisted techniques are incorporated to direct and prioritize the analysis and related tasks, based on the relevance of the observed properties and the expected impact of actions to be taken. Our development environment is an initial step in the direction of the very recent *Wise Computing* vision, which calls for turning the computer (namely, the development environment) into an equal member of the development team: knowledgeable, independent, concerned and proactively involved in the development process. Our tool is implemented within the context of *behavioral programming* (*BP*), a scenario-based modeling approach, where components are aligned with how humans often describe desired system behavior. Thus, our work further enhances the naturalness and incrementality of developing in BP.

1 INTRODUCTION

The development of large reactive software systems is an expensive and error-prone undertaking. Deliverables will often fail, resulting in unintended software behavior, exceeded budgets and breached time schedules. One of the key reasons for this difficulty is the growing complexity of many kinds of reactive systems, which increasingly prevents the human mind from managing a comprehensive picture of all their relevant elements and behaviors. Moreover, of course, the state-explosion problem typically prevents us from exhaustively analyzing all possible software behaviors. While major advances in modeling tools and methodologies have greatly improved our ability to develop reactive systems by allowing us to reason on abstract models thereof, specific solutions are quickly reaching their limits, and resolving the great difficulties in developing reliable reactive systems remains a major, and critical, moving target.

Over the years it has been proposed, in various contexts, e.g., (Rich and Waters, 1988; Reubenstein and Waters, 1991; Cerf, 2014; Harel et al., 2015c), that a possible strategy for mitigating these difficulties could lay in changing the role of the computer in the development process. Instead of having the computer serve as a tool, used only to analyze or check specific aspects of the code as instructed by the developer, one could seek to actually transform it into a member of the development team — a proactive participant, analyzing the entire system and making informed observations and suggestions. This way, the idea goes, the computer's superior capabilities of handling large amounts of code could be manifested. Combined with human insight and understanding of the system's goals, this synergy could produce more reliable and error-free systems.

In this paper we follow this spirit, and present a methodology and an interactive framework for the modeling and development of complex reactive systems, in which the computer plays a proactive role. Following the terminology of (Harel et al., 2015c), and constituting a very modest initial effort along the lines of the Wise Computing vision outlined there, we term this framework a *wise* framework. Intuitively, a truly *wise* framework should provide the developer with an interactive companion for all phases of system development, "understand" the system, draw attention to potential errors and suggest improvements and generalizations; and this should be done via two-

way communication with the developer, which will be very high-level, using natural (perhaps natural-language-based) interfaces. The framework presented here is but a first step in that direction, and focuses solely on providing an interactive development assistant capable of discovering interesting properties and drawing attention to potential bugs; still, it can already handle non-trivial programs, as we later demonstrate through a case-study.

Various parts of this approach have been implemented by a variety of researchers in other forms, as described in Sec. 6. A main novel aspect of our approach, however, is in the coupling of the notion of a proactive and interactive framework with a modeling language called *behavioral programming* (Harel et al., 2012b) — a scenario-based language, in which systems are modeled as sets of independent scenarios that are interleaved at runtime. This formalism makes it possible for our interactive development framework to repeatedly and quickly construct abstract executable models of the program, and then analyze them in order to reach meaningful conclusions. It is now widely accepted that a key aspect in the viability of analysis tools and environments is that they are sufficiently lightweight to be integrated into the developer's workflow without significantly slowing it down (Sadowski et al., 2015; Cristiano et al., 2015). We attempt to achieve this by leveraging scenario-based modeling. As demonstrated in later sections, the proactiveness of our approach and its tight integration into the development cycle can lead to early detection of bugs during development, when they are still relatively easy and cheap to fix.

The rest of this paper is organized as follows. In Sec. 2 we introduce scenario-based programming — the modeling formalism on top of which our approach is implemented, and also discuss some analysis techniques for scenario-based programs that are used in subsequent sections. In Sec. 3 we introduce our development framework by means of a simple example. In Sec. 4 we discuss the various components of the framework in more detail, and in Sec. 5 we describe a case-study that we conducted. Related work appears in Sec. 6, and we conclude in Sec. 7.

2 SCENARIO-BASED MODELING

Behavioral programming (*BP*) (Harel et al., 2012b) is a modeling approach aimed at designing and incrementally developing reactive systems. BP emerged from the *live sequence charts* (*LSC*s) formalism (Damm and Harel, 2001; Harel and Marelly, 2003), and, like LSCs, its basic modeling objects are *scenarios*. A behavioral model consists of independent scenario objects, each encoding a single desired or undesired behavior of the system. These behavioral models are executable: when run, the behaviors encoded by their constituent objects are all interwoven together, in a way that yields cohesive system behavior.

More specifically, an execution of a behavioral model is a sequence of points in which all scenario objects synchronize and declare events that they want to be considered for triggering (called *requested events*), events that they do not actively request but merely "listen out" for (*waited-for events*), and events whose triggering they forbid (*blocked events*). During execution, an event that is requested by some scenario and not blocked by any scenario is selected for triggering, and every scenario object that requested or waited for the event can update its internal state. Fig. 1 (borrowed from (Harel et al., 2014)) demonstrates a simple behavioral model. The formal definitions of behavioral modeling appear in Sec. 2.1.

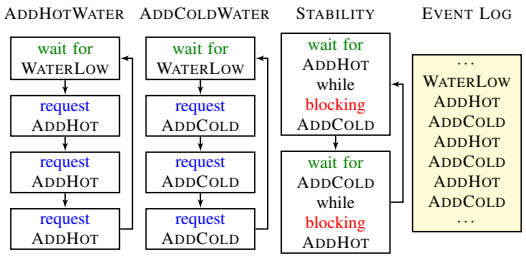

Figure 1: The incremental modeling of a system for controlling the water level in a tank with hot and cold water sources. Each scenario object is given as a transition system, where the nodes represent synchronization points. The scenario object AddHotWater repeatedly waits for WaterLow events and requests three times the event AddHot. Scenario object AddColdWater performs a similar action with the event AddCold, capturing a separate requirement, which was introduced when adding three water quantities for every sensor reading proved to be insufficient. When a model with objects AddHotWater and AddColdWater is executed, the three AddHot events and three AddCold events may be triggered in any order. When a new requirement is introduced, to the effect that water temperature be kept stable, the scenario object Stability is added, enforcing the interleaving of AddHot and AddCold events by using event blocking. The execution trace of the resulting model is depicted in the event log.

The motivation for using behavioral modeling is its strict and simple mechanism for inter-object communication. In particular, BP's request/wait-for/block interface facilitates incremental, non-intrusive development, and the resulting models often have scenario objects that are aligned with the requirements (Harel et al., 2012b). This is lent additional support by

studies that indicate that BP is *natural*, in the sense that it is easy to learn and fosters abstract programming (Gordon et al., 2012; Alexandron et al., 2014).

In practice, behavioral modeling is usually performed using various high level languages, such as Java, C++, Erlang, Javascript and, of course, LSCs, on which BP is based and from which it grew (see the BP website at `http://www.b-prog.org/`). Models written in these languages are fully executable, and are also referred to as *behavioral programs*. There, each scenario object is typically implemented as a separate thread, and inter-thread communication is restricted to event requesting, waiting-for and blocking — thus preserving the semantics of behavioral modeling. Technically, this is performed by having the scenario threads invoke a special synchronization method called *BSYNC*, and pass to it their requested/waited-for/blocked events. Once every scenario has synchronized, an *event selection mechanism* triggers one event that is requested and not blocked, and notifies the relevant scenarios.

For actual programming purposes it is often helpful to allow threads to also perform local actions — e.g., read from a file or turn on a light bulb. These actions are not included in the underlying behavioral model (i.e., they are abstracted away). The wise framework that we present here is designed to accompany the development of such behavioral programs, and is built on top the *BPC* package (Harel and Katz, 2014) for behavioral modeling in C++. This package also supports the distributed execution of behavioral programs (Harel et al., 2015a).

2.1 Formal Definitions

For completeness, we recap here briefly the formal definitions of behavioral modeling. Following the definitions in (Katz, 2013), a scenario object O over event set E is a tuple $O = \langle Q, \delta, q_0, R, B \rangle$, where Q is a set of states, q_0 is the initial state, $R : Q \to 2^E$ and $B : Q \to 2^E$ map states to the sets of events requested and blocked at these states (respectively), and $\delta : Q \times E \to 2^Q$ is a transition function.

Scenario objects can be composed, in the following manner. For objects $O^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1 \rangle$ and $O^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2 \rangle$ over a common event set E, the composite scenario object $O^1 \parallel O^2$ is defined by $O^1 \parallel O^2 = \langle Q^1 \times Q^2, \delta, \langle q_0^1, q_0^2 \rangle, R^1 \cup R^2, B^1 \cup B^2 \rangle$, where $\langle \tilde{q}^1, \tilde{q}^2 \rangle \in \delta(\langle q^1, q^2 \rangle, e)$ if and only if $\tilde{q}^1 \in \delta^1(q^1, e)$ and $\tilde{q}^2 \in \delta^2(q^2, e)$. The union of the labeling functions is defined in the natural way; e.g. $e \in (R^1 \cup R^2)(\langle q^1, q^2 \rangle)$ if and only if $e \in R^1(q^1) \cup R^2(q^2)$.

A *behavioral model* M is simply a collection of scenario objects $O^1, O^2, \ldots, O^n$, and the executions of M are the executions of the composite object $O = O^1 \parallel O^2 \parallel \ldots \parallel O^n$. Each such execution starts from the initial state of O, and in each state q along the run an enabled event is chosen for triggering, if one exists (i.e., an event $e \in R(q) - B(q)$). Then, the execution moves to state $\tilde{q} \in \delta(q, e)$, and so on.

2.2 Analyzing Behavioral Models

Earlier we explained the motivation behind behavioral modeling, from a developer's point of view. However, it turns out that due to its simple synchronization mechanism, behavioral modeling lends itself naturally also to formal analysis. We briefly recap a few such analysis methods, which are used by our proposed wise development framework.

2.2.1 Model Checking Behavioral Models

In (Harel et al., 2011; Harel et al., 2013a) a technique is presented, by which the underlying transition systems of individual scenario objects are extracted from high-level behavioral code and are then used in order to model check the behavioral model. The extraction of these transition systems is performed by running individual scenario objects in sandboxes and passing to them events, just as if they were triggered by the event selection mechanism, in a way that allows one to methodically explore their state spaces (Harel et al., 2013a). Model checking is then performed by adding special behavioral objects to the model that mark undesired behavior, and then traversing the states of the composite model to see if a violation can occur.

In order to mitigate the state-explosion problem and allow the model checking of larger behavioral models, one can replace behavioral objects or sets thereof with abstract behavioral objects (Katz, 2013). Intuitively, within a behavioral object, a set of states $q_1, q_2, \ldots, q_\ell$ can be abstracted away using a single state q, such that

$$R(q) = \bigcup_{i=1}^{\ell} R(q_i) \quad \text{and} \quad B(q) = \bigcap_{i=1}^{\ell} B(q_i).$$

The transition relation is then adjusted so that any transition between states s and t in the original model becomes a transition between s' and t' in the abstract object, where s' and t' are the abstract states representing s and t, respectively.

In (Katz, 2013) it is shown that, because abstract states block fewer events and request more events than their concrete counterparts, this sort of abstraction yields a behavioral model that is more *permissive* than the original one (i.e., it is an over-approximation). Typically, due to the reduction in the

number of states, this abstract model is also significantly smaller than its original counterpart. Model checking and program repair operations can then be performed on the abstract model (sometimes combined with local refinements steps), and the results are guaranteed to hold for the original system, thus enabling better coping with state-explosion. In later sections we make extensive use of this abstraction technique.

2.2.2 Compositional Verification of Behavioral Models

A useful property of behavioral modeling is that despite the small number of simple-looking concurrency idioms that it provides (i.e., the requesting, waiting-for and blocking idioms) it provides significant *succinctness* advantages. Specifically, it allows specifying behavioral objects that are *exponentially smaller* than what is possible using non-concurrent modeling formalisms, and even when compared to formalisms in which any of the requesting, waiting-for and blocking idioms are omitted (Harel et al., 2015b). An example appears in Fig. 2.

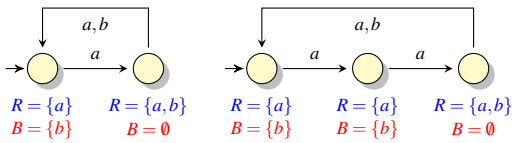

Figure 2: This behavioral model has two scenario objects, each depicted as a transition system. Every state corresponds to a synchronization point, and is labeled with its requested and blocked events, whereas the waited-for events are encoded on the transitions. The scenario on the left counts modulo two: at odd steps it requests event a and blocks event b, and at even steps it requests both events. The scenario on the right is similar, but counts modulo three, and only requests both events every third step. Together, these two objets count modulo 6, producing the language $(a^5(a+b))^\omega$. In (Harel et al., 2015b) it is shown that modeling this system in a non-concurrent formalism, or even in one that is devoid of the blocking idiom, requires $6 (= 3 \cdot 2)$ states instead of $5 (= 3+2)$. When generalized to the language $(a^n(a+b))^\omega$ for an arbitrarily large n, this gap between the sum and the product of the number of states in the constituent scenarios is exponential in n. For a more thorough discussion of the succinctness afforded by behavioral modeling, see (Harel et al., 2015b).

The succinctness afforded by behavioral modeling can sometimes be leveraged for efficient compositional verification (Harel et al., 2013b; Katz et al., 2015). For example, suppose that we wish to verify that in the model depicted in Fig. 2 event b can only be triggered every 6 steps. Direct model checking would entail exploring the 6 composite states of the system, but compositional verification would entail exploring the states of each object separately (a total of 5 states), characterizing the properties of each individual object, and then using an SMT solver to derive global correctness from these individual properties. More specifically, the individual object properties in this example can be formulated as $triggered(b,i) \implies i \equiv 0 (\text{mod } 2)$ for the object on the left and $triggered(b,i) \implies i \equiv 0 (\text{mod } 3)$ for the object on the right, where $triggered(b,i)$ means that the i'th event triggered was b. These properties can be verified on the individual objects. Using these object properties, an SMT solver can quickly deduce the desired property, $triggered(b,i) \implies i \equiv 0 (\text{mod } 6)$, circumventing the need to explore the composite states of the model. When the above example is generalized to $(a^n(a+b))^\omega$ for a large n, the gap in the number of explored states between the direct approach (roughly the product of the number of individual object states) and the compositional approach (roughly the sum thereof) is exponential in n (Harel et al., 2013b).

The key observation, which we leverage repeatedly in the following sections, is that in scenario-based modeling it is often simple, and computationally cheap, to analyze many small scenario objects — and then use this information to reason about the model as a whole.

3 DEVELOPMENT IN A WISE FRAMEWORK: AN EXAMPLE

In this section we attempt to convey to the reader, intuitively, the sense of working in a wise development framework from a developer's point of view. Thus, we focus almost exclusively on the user experience, and defer more details about the inner workings of the framework itself to Sec. 4.

We demonstrate the framework's operation through the incremental modeling of a small, illustrative system. Suppose we are developing behavioral code for a safe that has three levers and an "open door" button. The specification given to us indicates that in order to open the door, a user needs to correctly configure the three levers and then click the button. Clicking the button when the levers are not correctly configured should not open the door. We refer to the three levers as levers A, B and C; and each lever has three possible positions, denoted as *one*, *two* and *three*. We denote the configuration of the levers as a tuple: for instance, configuration $\langle 1,3,2 \rangle$ indicates that lever A is in position one, lever B is in position three, and lever C is in position two. The initial configuration is $\langle 1,1,1 \rangle$, and the correct

configuration for opening the door is $\langle 2,3,2 \rangle$. The user can request the triggering of events of the form SETXTOY, indicating that lever X is set to position Y, and also of CLICKBUTTON events. The system may request an OPENDOOR event, as well as any internal event needed for the implementation.

We now describe the incremental modeling of this system in BPC, accompanied by the wise framework. We start by modeling the three levers. This is done by creating, for each lever, a scenario object that waits for events signaling that the position of that lever has changed, and storing the current position. The code appears and is explained in Fig. 3.

```
1  Event position = SetXToOne;
2  while ( true ) {
3    set<Event> requested = {};
4    set<Event> waitedFor =
5      { SetXToOne, SetXToTwo, SetXToThree };
6    set<Event> blocked;
7
8    switch( position ) {
9    case SetXToOne:
10     blocked = { LeverXInTwo, LeverXInThree };
11   case SetXToTwo:
12     blocked = { LeverXInOne, LeverXInThree };
13   case SetXToThree:
14     blocked = { LeverXInOne, LeverXInTwo };
15   }
16
17   BSYNC( requested, waitedFor, blocked );
18   position = lastEvent();
19 }
```

Figure 3: BPC code for a scenario object called LeverX, representing the behavior of a single lever X (X represents A, B or C). Line 17 contains the *BSYNC* synchronization call, where the object synchronizes with all other objects and declares its requested, waited-for and blocked events. The lever object never requests any events, and continuously waits for events signifying that the lever has changed its physical position — events SETXTOONE, SETXTOTWO, and SETXTOTHREE. When one of these is triggered, line 17 returns, and the object updates its internal state in line 18. Note also events LEVERXINONE, LEVERXINTWO and LEVERXINTHREE, which represent other scenarios querying the physical position of lever X. The lever object constantly blocks those events that correspond to all "wrong" physical positions. Thus, if another object requests all three events, then only one event — the one corresponding to the actual lever's position — will be triggered. An example appears in Fig .4.

After modeling the three lever objects, we get the first input from the wise development framework:

> **Warning:** *Objects* LeverA, LeverB *and* LeverC *constitute a ternary shared array. However, they are not used. Consider removing them.*

We should emphasize that the wise development framework is oblivious to the specifics of our program, i.e., it has no concept of levers. It did, however, recognize a pattern in our system model: that the three lever objects actually operate like a "shared array". Here, the term shared array means that other objects can "write" to it (i.e., by requesting SETXTOY events), or "read" from it (by requesting LEVERXINY events). This is an interesting insight about the implementation, which we did not even have in mind, but which the development framework will utilize later on. As for the comment that the levers are currently unused, this makes sense — as we have not yet written any additional code.

Next, we add a scenario that allows the user, through a simple interface, to request the triggering of SETXTOY events, and also the CLICKBUTTON event (code omitted). When we recompile the code, the development framework prompts us that now the shared array is written to but is never read from, and can still be removed. Then, we add the *ButtonPressed* scenario (Fig. 4) that handles the pressing of the button — it queries the lever configuration, and if it is $\langle 2,3,2 \rangle$ it requests an OPENDOOR event.

However, as the caption explains, the code in Fig. 4 is actually erroneous: we copied and pasted the code checking lever B but did not correctly modify it to check lever C. The wise development framework now produces the following message:

> **Warning:** *Scenario* ButtonPressed *has an unreachable synchronization point in line 23. Suggesting an optimization. Also, the state of* LeverC *is never read.*

This message immediately points us to the error in the model, giving us enough information to quickly realize what has happened. The optimization proposed by the framework (not shown), in which the unreachable state is removed, is actually a graphical representation using the Goal visualization tool (Tsay et al., 2007).

We stress that the realization that line 23 is unreachable is not trivial, as it is not a property that is local to the *ButtonPressed* object. In particular, it cannot be deduced by inspecting the *ButtonPressed* object in isolation, and thus it is very different from deducing, say, that in $if(false)(foo())$ the function $foo()$ can never be called. Rather, this property stems from the joint behavior of *ButtonPressed* and *LeverB*, where *ButtonPressed* expects *LeverB* to be in two different states simultaneously, which cannot occur.

And so, we correct the error in line 19 of *ButtonPressed*. Now the warnings from the development framework disappear, and instead we receive the following information:

```
1  while ( true ) {
2    BSYNC( {}, { CLICKBUTTON }, {} );
3
4    Set<Event> queryA = { LEVERAINONE,
5      LEVERAINTWO, LEVERAINTHREE };
6    Set<Event> queryB = { LEVERBINONE,
7      LEVERBINTWO, LEVERBINTHREE };
8    Set<Event> queryC = { LEVERCINONE,
9      LEVERCINTWO, LEVERCINTHREE };
10
11   BSYNC( queryA, {}, {} );
12   if ( lastEvent() != LEVERAINTWO )
13     continue;
14
15   BSYNC( queryB, {}, {} );
16   if ( lastEvent() != LEVERBINTHREE )
17     continue;
18
19   BSYNC( queryB, {}, {} );
20   if ( lastEvent() != LEVERBINTWO )
21     continue;
22
23   BSYNC( { OPENDOOR }, {}, {} );
24 }
```

Figure 4: The *ButtonPressed* scenario, which waits for a CLICKBUTTON event, queries the configuration of the three levers (lines 11, 15 and 19), and if they are correctly set requests an OPENDOOR event (line 23). Querying the position of lever X is performed by simultaneously requesting events LEVERXINONE, LEVERXINTWO and LEVERXINTHREE. Only the "correct" event, i.e. the event that corresponds to lever X's current position, will be triggered, because the other two events will be blocked by LeverX's scenario object. Observe that this scenario has a bug: in line 19, instead of checking whether lever C is in position two, we mistakenly check if lever B is in position two. When this line in the code (line 19) is reached we already know that lever B is in position three (line 15), and so line 23 can never be reached until this bug is fixed.

> **Information:** *Event* OPENDOOR *appears to only be triggered after event* LEVERCINTWO.

And then, a few seconds later:

> **Information:** *Event* OPENDOOR *appears to only be triggered when the shared array is in configuration* LEVERAINTWO, LEVERBINTHREE, LEVERCINTWO.

Here, the development framework was able to deduce — without any information regarding the specific system being modeled — that configuration $\langle 2,3,2 \rangle$ is of special importance in the triggering of OPENDOOR events! This does not indicate a potential error that the development framework found, as in the previous cases shown, but rather an *emergent property* that the framework was able to deduce — completely on its own — and which may be of interest to the developer. Such emergent properties can serve to either draw attention to bugs or reassure the developer that the model functions as intended, which was the case here. Details about how this conclusion was reached are presented in the next section. A video demonstrating the examples described in this section is available online at (Harel et al., 2016).

4 EXPLAINING THE FRAMEWORK: THE THREE "SISTERS"

We now describe in some detail the inner workings of our wise development framework and the various components from which it is comprised. Although this framework is but a first step towards the ultimate goal described in (Rich and Waters, 1988; Reubenstein and Waters, 1991; Cerf, 2014; Harel et al., 2015c), it utilizes some powerful techniques, and building it was far from trivial. An up-to-date version of the tool, as well as video clips demonstrating its main principles, can be found online at (Harel et al., 2016).

As mentioned earlier, our wise development framework is designed to accompany the development of behavioral models, as defined in Sec. 2, and in particular behavioral programs written in C++ using the BPC package (Harel and Katz, 2014). The framework involves three new logical components, over and above the BPC package itself, and apart from the additional external tools we invoke, such as a model checker and an SMT solver (see Fig. 5). We call these components *the three sisters*: *Athena*, *Regina* and *Livia*.

Intuitively, each sister handles a different set of services provided by the wise development environment. *Athena*, the wise one, works proactively during development, in an off-line fashion. Her purview is the usage of formal tools to analyze scenario objects and produce logically accurate conclusions about them, which are valid for all runs. For instance, in the example discussed in Sec. 3, the conclusion that a certain scenario state could never be reached was derived by Athena, using model checking.

Regina, more regal than her sisters, also works off-line, but her purview includes semi-formal methods: using abstract models of the system, she runs multiple simulations, collecting statistical information as she goes. In what is a form of specification mining she then attempts to reach interesting conclusions, to be presented to the modeler. Her conclusions may not be valid for all runs, but they have the advantage of

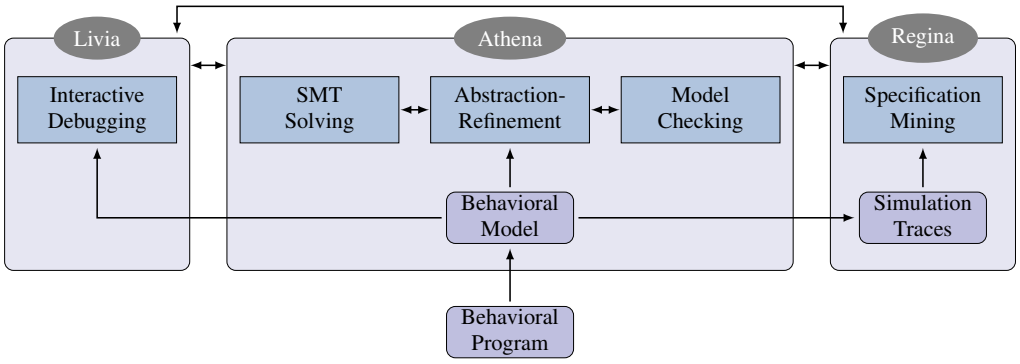

Figure 5: A high-level overview of the three sisters. The developer provides a behavioral program, from which Athena extracts a behavioral model. She then analyzes this model using abstraction-refinement, model checking and SMT solving. Athena also shares the behavioral model with her sisters: with Regina for the purpose of specification mining, and with Livia for interactive debugging. The three sisters also exchange information with each other — for instance, Regina may ask Athena to attempt to formally prove an emergent property that she mined.

reflecting numerous executions, and can thus provide valuable insights about what will happen in typical runs. Again recalling the example in Sec. 3, the discovery that OPENDOOR events were related to lever configuration $\langle 2,3,2\rangle$ was made by Regina, as a result of running multiple simulations of the system.

The last sister, *Livia*, who was not demonstrated in Sec. 3, complements the other components by providing on-line support for the developer, for debugging purposes. She can monitor the system as it runs, and help the developer recognize and comprehend unexpected behavior — also by sometimes running local simulations and tests, and by using an abstract model of the system.

The three sisters also cooperate: for instance, emergent properties recognized by Regina can be passed to Athena for formal verification, and Livia may use Athena's formal analysis tools for local analysis at runtime. Together, the three sisters are meant to accompany the programmer during development time and provide the various features which together constitute the initial wise development framework.

We now delve deeper into the technical aspects of the framework. The offline components Athena and Regina continuously run as background processes at development time. After each successful compilation of the code, these two sisters receive a fresh snapshot of the program and begin to analyze it. Next, we discuss the main steps in their analysis process, repeated after each compilation.

Step 1: Extracting a behavioral model. The first step is a key one, and is performed by Athena: she constructs an abstract, executable behavioral model of the program, to be used by all three sisters, in all their further analysis operations. Intuitively, Athena extracts from the program — given as C++ code — the underlying scenario objects, as described in Sec. 2.2.1. This technique, discussed in (Harel et al., 2013a), leverages the fact that concurrent scenarios communicate only through the strict BP synchronization mechanism. Athena thus runs each scenario individually in a "sandbox", while mimicking the program's event selection mechanism, exploring the scenario's states and constructing its underlying scenario object. The resulting abstract model of the program thus completely and correctly describes all inter-scenario communication, while the rest of the information (internal scenario actions) is abstracted away, allowing the development framework to handle larger programs. Athena then shares this abstract behavioral model with Regina for the purpose of running simulations, and with Livia for the purpose of online analysis.

Step 2: Identifying logical modules. The next phase is also performed by Athena, and it involves partitioning the program's scenarios into logical modules according to their functionality. This clustering phase is needed in order to increase the tool's scalability: when trying later to check a property ϕ that does not involve program module A, the sisters will attempt to abstract away module A — reducing the total number of states that have to be explored. We have set things up so that information regarding the scenario grouping into modules is not provided by the programmer; rather, Athena uses a clustering algorithm (Katz, 2013) to determine scenarios' correlations to events, and then groups them accordingly.

The clustering algorithm operates as follows. The basic idea is that objects that are logically related are likely to "care" about the same events. Thus, we define the *correlation* between a scenario object O (with state set Q) and an event e as

$$cor(O,e) = \frac{|\{q \in Q \mid e \in R(q) \cup B(q)\}|}{|Q|},$$

i.e. the portion of O's states in which event e is requested or blocked. Given a threshold M, this correlation relation defines an equivalence relation, where if $cor(O_1,e) > M$ and $cor(O_2,e) > M$ then objects O_1 and O_2 are in the same equivalence class. M is determined dynamically — Athena starts by setting it to 1, and then gradually reduces it until the computed equivalence classes are *sufficiently* large. For the definition of "sufficiently", we have empirically found that requiring at least 75% of the computed object classes to have at least 4 objects in them worked well on our examples — i.e., it leads to non-trivial equivalence classes that indeed contain logically related scenario objects.

Apart from applying this clustering algorithm, Athena also compares the extracted behavioral model to a predefined meta-model with known/common programming constructs (Katz et al., 2015) which we have built into our tool. Currently supported constructs include semaphores, shared arrays, sensors and actuators, and our on-going work includes adding support for additional ones. If it is discovered that certain scenario objects are instantiations of meta-objects that are logically connected (e.g., one scenario implements a semaphore and another scenario waits on that semaphore), they may also be grouped together into the same logical module. Recalling the example of Sec. 3, it was Athena who realized, by comparing the input model to her stored meta-model, that the lever scenarios constituted a shared ternary array.

Step 3: Deriving candidate emergent properties. The next step employs specification mining techniques, and is performed by Regina. She attempts to determine, by running multiple simulations on the behavioral model of the program (which was provided by Athena), a list of possible properties of the system. These are discovered by analyzing simulation traces and looking for patterns: events that always (or never) appear together, events that cause other events to occur, producer-consumer patterns, etc. Such abilities can be viewed as a form of mining traces for scenario-based specifications (see, e.g., (Lo et al., 2007)). The generated properties are not guaranteed to be valid, and need to be checked — either formally, by Athena (e.g., by model checking), or statistically, by Regina (e.g., by running even more simulations of the system). If and when proven correct, and assuming they are relevant, these emergent properties can serve as part of the official certification that the system performs as intended (an example appeared at the end of Sec. 3). However, even when the sisters guess "incorrectly", i.e., come up with properties that are later shown not to hold, this can still be quite useful, often drawing the developer's attention to bugs.

Step 4: Prioritizing properties. Once Regina has obtained a list of candidate properties, the next step is to attempt to prove or disprove each of them. In our experience with the tool, for a large system this list tends to contain dozens of properties, and so it is typically infeasible to model-check each and every one of them and present the conclusions quickly. This difficulty is mitigated in our system in several ways: (i) We attempt to reduce redundancy. Thus, if we have identified a class of similar emergent properties, we may start by checking just one of them and assign the remaining properties a lower priority. (ii) We employ a prioritization heuristic, aimed at checking first those properties that are likely to be more interesting to the user. For instance, if a semaphore-like construct was identified, we will prioritize the checking of a property that states that in some cases mutual exclusion may be incorrectly implemented, as this is considered a safety critical property, and thus may be more interesting to the user. (iii) We present any conclusion to the user as soon as it is reached, while the sisters continue to check additional properties. (iv) We leave room for manual configuration of the framework; i.e. the developers can prioritize the testing of certain properties, if they so desire.

Having obtained a prioritized list of properties to check, the remainder of the framework's operation is dedicated to discharging each of them (step 5) and presenting the results to the user (step 6). The framework will thus alternate between steps 5 and 6 until all the candidate emergent properties have been discharged, or until it runs out of time — possibly due to a renewed compilation of the code and the start of another analysis cycle.

Step 5: Proving/disproving properties. The wise development framework now attempts to check, in sequence, each of the candidate properties. As there are typically many properties to check, it is desirable to dispatch each property as soon as possible — so that the results will be presented to the user in time to be relevant. To this end, we build upon a large body of existing techniques for formally analyzing scenario-based models, as discussed in Sec. 2.2. These include, e.g., abstraction-refinement techniques (Katz, 2013), program instrumentation techniques (Harel et al., 2014) and SMT-based compositional techniques (Harel et al., 2013b; Katz et al., 2015). Indeed, this is the main reason why we chose to implement a wise framework in the context of the scenario-based paradigm: it is sufficiently expressive for real-world systems (Harel and Katz, 2014), but on the other hand is amenable to, and even facilitates, program analysis (Harel et al., 2015d). Since the ability to quickly and repeatedly

analyze behavioral models is critical to our approach, this seemed like a natural fit.

By default, Athena will attempt to discharge properties using abstraction-refinement based model checking for scenario-based programs (Katz, 2013). Alternatively, the user may configure the framework to use other tools: explicit model checking or an SMT-based approach (also performed by Athena), or have Regina perform statistical checking. Here, statistical checking entails Regina running many simulations under various environment assumptions (fair/unfair environment, starvation, round-robin triggering of events, etc.), and repeatedly checking the property at hand. This technique is not guaranteed to be sound, of course, but it can yield interesting conclusions nonetheless. Moreover, it affords a level of assurance of the property holding, which may suffice for ones that are not safety-critical. We are currently in the process of implementing an adaptive mechanism that would attempt to run the various techniques in Athena's arsenal with a timeout value, abandoning a technique if it does not prove useful for a specific input.

Step 6: Presenting the results. The final phase of the sisters' analysis cycle involves displaying to the user the properties that were proved or disproved. In some cases, the mined properties are irrelevant, and the user may discard them. In other cases, desirable properties are shown to hold, and the user is then reassured that the program is working as intended. The remaining cases can either be undesired properties that do hold, or "classical" bugs, where a property that the user assumed to hold is proven by Athena to be violated. In the latter case, the user can interact with the development framework, and ask for (i) a trace log showing how the property was violated; (ii) a suggestion for a fix, in the form of a scenario that is to be added the model (Harel et al., 2012a; Harel et al., 2014); or (iii) the addition of a monitor scenario, to alert the user when the property is violated at run-time (usually used for debugging purposes).

Apart from the analysis flow just described, Athena also supports some forms of automatic optimization — e.g., identifying parts of the code that may never be reached and suggesting how to remove them, as we saw in Sec. 3.

So far we have dealt with the framework's *offline* capabilities, performed by Athena and Regina — that is, analysis performed during development, usually after compilation, but without running the actual system. In contrast, the *online* sister Livia participates in debugging the system as it runs. She connects to the system and monitors it by "pretending" to be a scenario object in the behavioral program, which constantly waits for every one of the program's events. Livia also has at her disposal the abstract model of the program produced by Athena in the first step of the analysis, and she uses it — along with the sequence of events triggered so far — to keep track of the internal states of every object in the system.

Livia's main capability is to launch bounded model checking from a given state, checking for properties at run time. For instance, the user debugging the program might believe that a corner case has been arrived at, from which the initial state can never be reached, and can ask Livia to investigate this. She will attempt to verify the property using bounded model checking. This sort of operation will typically be initiated manually by the user, but Livia also attempts to recognize problematic cases on her own — for instance, when certain objects in the system have become deadlocked or simply have not changed states in a while — and asks the user whether she should investigate. As previously mentioned, whenever a more thorough analysis is requested Livia can also pass queries along to Athena and Regina.

5 A CASE-STUDY: A CACHE COHERENCE PROTOCOL

In order to evaluate the applicability of our wise development framework to larger systems, we used it to develop a *cache coherence* protocol. Such protocols are designed to ensure consistent shared memory access in a set of distributed processors. In order to minimize the number of read operations on the actual memory, processors cache the results of previous reads. Consistency then means that cached values stored throughout the system need to be invalidated when a processor writes a new value to the actual memory. The motivation for choosing this particular example was that cache coherence protocols are notoriously susceptible to subtle, concurrency-related bugs, making them a prime candidate to benefit from a wise development environment. The specific protocol that we implemented is a variant of the well-studied Futurebus protocol (Clarke et al., 1995).

An important question that we attempted to address through the case-study was whether the notion at the core of our approach — namely, developing a non-trivial system *together* with the aid of a proactive framework — is convenient and/or useful. While this issue is highly subjective, we can report that in the systems we modeled the sisters' aid proved valuable. In particular, they typically displayed their insights about the program in a timely manner, with results starting to flow in seconds after each compilation;

and although sometimes the insights proved irrelevant, in several cases they pointed out concurrency-related bugs that we had overlooked, and which we then repaired. In other cases, the framework's conclusions served to confirm that the model was working as intended, which was particularly reassuring, for example, after adding a new feature.

Another goal that we had was to identify a basic methodology for how modeling or programming should be conducted in such an environment. A setup that we found convenient is depicted in Fig. 6. As for the flow of the process, we found it useful to have a quick glance at the framework's logs after each compilation to check for any critical mistakes, and to look more thoroughly at the logs after making significant changes to the code base. Occasionally, when certain properties draw our particular attention, we used the interactive interface (depicted in Fig. 7) to guide the framework.

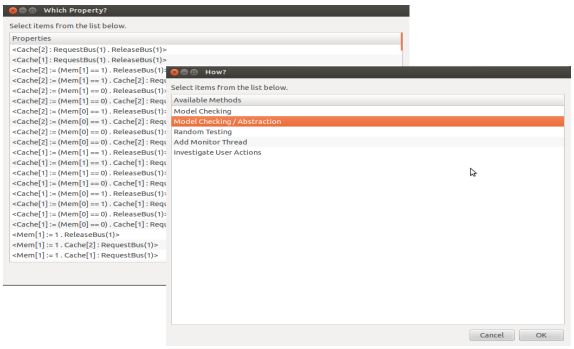

Figure 7: A simple GUI that we occasionally used in order to interactively instruct the development environment to focus on certain emergent properties. The interface allows us to choose which of the candidate emergent properties should be handled next, and how: explicit or abstraction-based model checking, statistical testing, creating a monitor thread, etc.

We now show two examples of the usage of the wise development framework during our case-study. A more complete set of examples, as well as the entire code base, is available online at (Harel et al., 2016). In order to properly illustrate the tool's usage during development, we took snapshots of our code at significant milestones, along with the conclusions that the wise framework was able to draw from it — these are also available online. Finally, we also provide there a video clip that features the development framework in action.

Fig. 8 depicts a list of emergent properties that the development framework produced at one point during development. Recall that unless given specific instructions by the developer, the tool begins to check these properties, one by one; the figure shows a list of

```
Checking emergent properties:

ReleaseBus(1) <--> Cache[2] : RequestBus(1)
   [fails]
Cache[2] : RequestBus(1) --> ReleaseBus(1)
   [holds]
ReleaseBus(1) <--> Cache[1] : RequestBus(1)
   [fails]
Cache[1] : RequestBus(1) --> ReleaseBus(1)
   [holds]
Cache[2] := (Mem[1] == 1) --> ReleaseBus(1)
   [holds]
ReleaseBus(1) <--> Pc[1] : Success
   [fails]
Cache[2] : RequestBus(1) <--> Pc[2] : Success
   [fails]
...
```

Figure 8: A list of emergent properties produced and checked by the wise development framework. The tool typically does not finish checking everything on the list, and so information is displayed as soon as it is available. A counter-example is available for properties that fail to hold.

properties that have already been checked, indicating which of them hold and which do not. The tool mines for various types of properties, two of which are depicted in the figure: implications, denoted $a \rightarrow b$, i.e., whenever event a occurs b also occurs a short time earlier or later, and equivalences, denoted $a \leftrightarrow b$, i.e., the implication holds in both directions.

Fig. 9 depicts an example for which Athena's abstraction-based model checking proved especially handy, allowing her to quickly cover more properties. There, the emergent property being verified was that "cache 3 cannot acquire bus 2 repeatedly without first releasing it" — a property that describes mutual exclusion in the bus ownership. This property is an instantiation of the general pattern "consecutive a events must have b events between them". At the time this property was mined and tested, directly model checking it entailed exploring 972233 reachable states and took over 27 minutes. By using the abstraction-refinement techniques discussed in Sec. 2.2.1, Athena was able to abstract away irrelevant parts of the code (namely code modules that only pertained to other buses). In this way, verifying the property entailed exploring just 21000 reachable states, and took less than 31 seconds. The key observation here is that this is by no means merely a standard direct usage of abstraction-refinement. The entire process — finding the emergent property, figuring out which modules are not likely to affect it so that they can be abstracted away, and then model checking the property on the abstract model — were all handled proactively and automatically by the framework.

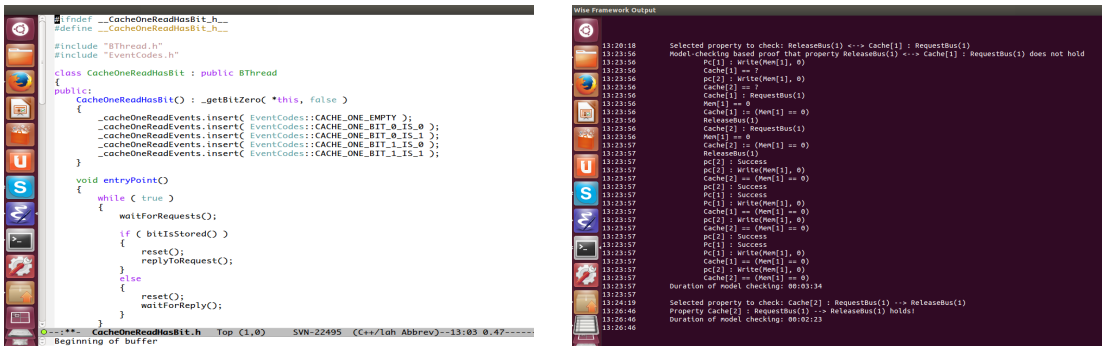

Figure 6: Screenshots of our wise development framework, taken during the cache coherence case-study. The window on the left depicts a standard editor, in which the code of the program is being written. The analysis tools are running in the background, and with every successful compilation of the code they automatically receive a fresh snapshot and analyze it. The window on the right shows output from the analysis — in this case, emergent properties that were examined. One property was proved correct and another was shown not to hold (a counter-example is provided). Most of the time we had these two windows open on separate screens.

```
Checking emergent property:
  Consecutive Cache[3] : RequestBus(2) events
  must have ReleaseBus(2) events between them

Attempting abstraction-based model checking
Abstracting module 1:
  CacheOneUpdate, CacheTwoUpdate,
  CacheTwo, CacheOne,
  CacheTwoReadFetchBit, CacheOneReadFetchBit,
  CacheTwoReadHasBit, CacheTwoWriteFetchBit,
  CacheTwoWriteHasBit, PcTwoRead, PcTwoWrite,
  CacheOneReadHasBit, CacheOneWriteHasBit,
  PcOneRead, PcOneWrite, CacheOneWriteFetchBit

Abstracting module 2:
  CacheTwoInvalidate

Abstracting module 3:
  CacheOneInvalidate

Conclusion: property [holds]
```

Figure 9: Extracts from the logs of the wise development framework, illustrating the autonomous verification of an emergent property that has been identified. The three code modules depicted (each a set of scenario objects) are irrelevant to the property at hand, and are automatically abstracted. Other modules in the program, those that are relevant to the property at hand, are not abstracted. The property is then verified for the resulting over-approximation — leading to improved performance.

Clearly, such speedups allow the framework to cover more properties and present them to the programmer in a timely manner.

6 RELATED WORK

Work related to subject of this paper can be viewed in two perspectives. One is over the individual capabilities of the three sisters, that is, mainly, discovering and proposing candidate emergent properties, and then verifying or refuting these properties. The other perspective is that of the overall view of a wise development environment that accompanies the developer and automatically and proactively carries out these tasks and others, such as requirements analysis, specification mining, test generation, synthesis, and more.

From the first perspective, there is a vast amount of pertinent research, and we focus here on only a few of the relevant papers. The actions performed by Regina, i.e. the dynamic discovery of candidate properties and invariants from program execution logs, is a form of *specification mining* (Ammons et al., 2002). This topic has been studied in the context of scenario-based specification in, e.g., (Cantal de Sousa et al., 2007; Lo and Maoz, 2008), and Regina uses similar techniques. For instance, she looks for emergent properties that have the *trigger and effect* structure of (Lo and Maoz, 2008). However, a key aspect in Regina's operation is the need to conclude the mining phase as quickly as possible, so that she can be seamlessly integrated into the development cycle. This is achieved by employing prioritization heuristics, and putting limits on the number of traces (and lengths thereof) that Regina considers. In the future we intend to enhance Regina with a mechanism similar to the one discussed in (Cohen and Maoz, 2015), where statistical criteria are used to determine when "enough" traces have been considered, hopefully boosting her performance even further.

Checking whether properties mined from traces indeed hold for the model in general brings us to the broad field of program and model verification. Many powerful and well known tools exist, such as SPIN, SLAM, BLAST, UPPAAL, Java Pathfinder, ASTRÉE, ESC/Java and others, and they utilize many forms of explicit and symbolic model checking, static analysis, deductive reasoning, and SAT and SMT solving (see (Alur et al., 2015) for a brief survey of the application of such methods in practice). In our framework these tasks are handled by Athena, and she uses tools specifically optimized for behavioral models (Harel et al., 2011; Katz, 2013; Katz et al., 2015).

As to the second perspective, successful attempts at automatic property discovery and subsequent verification appear, e.g., in (Nimmer and Ernst, 2001; Zhang et al., 2014). There, the Daikon tool is used to dynamically detect candidate program invariants which are then used to either annotate or instrument the program. In (Nimmer and Ernst, 2001) these guide ESC/Java in verifying the properties, and in (Zhang et al., 2014) they help guide symbolic execution in the discovery of additional or refined invariants. The motivation and approach of Daikon are very close to ours, but we aim at constructing a fully integrated, proactive and interactive environment, built upon the highly incremental paradigm of behavioral modeling.

Providing an interactive analysis framework that is tightly integrated into the development cycle/environment has become quite widespread in the industry over recent years. Some noticeable examples are Google's *Tricorder* (Sadowski et al., 2015), Facebook's *Infer* (Cristiano et al., 2015) and VMWare's *Review Bot* (Balachandran, 2013) tools. These tools use static analysis to automate the checking for violations of coding standards and for common defect patterns. Lessons learned from these projects indicate that, in order to be successfully accepted by programmers, an integrated analysis framework should have the following properties: (i) it needs to seamlessly integrate into the workflow of developers; (ii) it must produce results quickly; and (iii) it has to perform its analysis in a modular manner, so that it can scale reasonably well to large projects. The design of our framework is indeed aimed at achieving these properties. In particular, for the modular analysis part, Athena attempts to leverage the special properties of scenario-based models and reason about individual objects. In (Harel et al., 2015b), it is shown that objects in behavioral models often have very small state spaces; and this allows Athena to effectively compare these objects to her stored meta-model and identify object patterns that can later be used for analysis.

7 CONCLUSION

In this paper we contribute to the effort of simplifying and accelerating development of robust reactive systems, by proposing a development framework along the lines raised in e.g., (Cerf, 2014; Harel et al., 2015c). In a nutshell, the idea is to start with a modeling/programming formalism that is expressive, modular and relatively simple, and integrate quick, continuous, and easy-to-use analysis into the development process. This entails extending and adjusting existing analysis techniques in order to render them more interactive and proactive.

Our development framework is currently comprised of three main elements: specification mining and initial semi-formal analysis for generating candidate system properties, abstraction-assisted formal analysis for verification of detected properties, and run-time debugging. When integrated into the development cycle, these elements can often draw developers' attention to subtle bugs that could otherwise be missed. We carried out initial evaluation of the framework by iteratively developing a cache coherence protocol, and saw that it was successful in discovering and reporting bugs.

In the future we plan to carry out a more extensive, empirical comparison between our development framework and related tools, such as Tricorder (Sadowski et al., 2015) and Infer (Cristiano et al., 2015). We also plan to enhance Regina's specification-mining capabilities with learning techniques (Ammons et al., 2002), allowing her to learn over time which emergent properties are most valuable to programmers and should be checked first.

While our work so far is but an early step towards the vision of the computer acting as a wise, fully-fledged proactive member of the development team, we hope that it contributes to demonstrating both the viability and the potential value of this direction.

ACKNOWLEDGEMENTS

This work was supported by a grant from the Israel Science Foundation, by a grant from the German-Israeli Foundation (GIF) for Scientific Research and Development, by the Philip M. Klutznick Research Fund and by a research grant from Dora Joachimowicz.

REFERENCES

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

Alur, R., Henzinger, T. A., and Vardi, M. Y. (2015). Theory in practice for system design and verification. *ACM Siglog News*, 2(1):46–51.

Ammons, G., Bodik, R., and Larus, J. (2002). Mining Specifications. *ACM Sigplan Notices*, 37(1):4–16.

Balachandran, V. (2013). Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. In *Proc. 35th Int. Conf. on Software Engineering (ICSE)*, pages 931–940.

Cantal de Sousa, F., Mendonca, N. C., Uchitel, S., and Kramer, J. (2007). Detecting Implied Scenarios from Execution Traces. In *Proc. 14th Working Conf. on Reverse Engineering (WCRE)*, pages 50–59.

Cerf, V. (2014). A Long Way to Have Come and Still to Go. *Communications of the ACM*, 1(58):7–7.

Clarke, E., Grumberg, O., Hiraishi, H., Jha, S., Long, D., McMillan, K., and Ness, L. (1995). Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6(2):217–232.

Cohen, H. and Maoz, S. (2015). Have We Seen Enough Traces? In *Proc. 30th Int. Conf. on Automated Software Engineering (ASE)*.

Cristiano, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. (2015). Moving Fast with Software Verification. In *Proc. 7th. Int. Conf. on NASA Formal Methods (NFM)*, pages 3–11.

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

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

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

Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., and Weiss, G. (2013b). On Composing and Proving the Correctness of Reactive Behavior. In *Proc. 13th Int. Conf. on Embedded Software (EMSOFT)*, pages 1–10.

Harel, D., Kantor, A., Katz, G., Marron, A., Weiss, G., and Wiener, G. (2015a). Towards Behavioral Programming in Distributed Architectures. *Science of Computer Programming*, 98(2):233–267.

Harel, D. and Katz, G. (2014). Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures. In *Proc. 4th Int. Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!)*, pages 95–108.

Harel, D., Katz, G., Lampert, R., Marron, A., and Weiss, G. (2015b). On the Succinctness of Idioms for Concurrent Programming. In *Proc. 26th Int. Conf. on Concurrency Theory (CONCUR)*, pages 85–99.

Harel, D., Katz, G., Marelly, R., and Marron, A. (2015c). Wise Computing: Towards Endowing System Development with True Wisdom. Technical Report. `http://arxiv.org/abs/1501.05924`.

Harel, D., Katz, G., Marelly, R., and Marron, A. (2016). An Initial Wise Development Environment for Behavioral Models: Supplementary Material. `http://www.wisdom.weizmann.ac.il/~harel/Modelsward.wisecomputing`.

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

Harel, D., Katz, G., Marron, A., and Weiss, G. (2014). Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs. *Transactions on Computational Collective Intelligence (TCCI)*, 16:1–33.

Harel, D., Katz, G., Marron, A., and Weiss, G. (2015d). The Effect of Concurrent Programming Idioms on Verification. In *Proc. 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 363–369.

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

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

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

Katz, G. (2013). On Module-Based Abstraction and Repair of Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 518–535.

Katz, G., Barrett, C., and Harel, D. (2015). Theory-Aided Model Checking of Concurrent Transition Systems. In *Proc. 15th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 81–88.

Lo, D. and Maoz, S. (2008). Mining Scenario-Based Triggers and Effects. In *Proc. 23rd Int. Conf. on Automated Software Engineering (ASE)*, pages 109–118.

Lo, D., Maoz, S., and Khoo, S.-C. (2007). Mining Modal Scenario-Based Specifications from Execution Traces of Reactive Systems. In *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE)*, pages 465–468.

Nimmer, J. W. and Ernst, M. D. (2001). Static Verification of Dynamically Detected Program Invariants: Integrating Daikon and ESC/Java. *Electronic Notes in Theoretical Computer Science*, 55(2):255–276.

Reubenstein, H. and Waters, R. (1991). The Requirements Apprentice: Automated Assistance for Requirements

Acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240.

Rich, C. and Waters, R. (1988). The Programmer's Apprentice: A Research Overview. *Computer*, 21(11):10–25.

Sadowski, C., van Gogh, J., Jaspan, C., Söederberg, E., and Winter, C. (2015). Tricorder: Building a Program Analysis Ecosystem. In *Proc. 37th Int. Conf. on Software Engineering (ICSE)*.

Tsay, Y., Chen, Y., Tsai, M., Wu, K., and Chan, W. (2007). GOAL: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 466–471. Springer.

Zhang, L., Yang, G., Rungta, N., Person, S., and Khurshid, S. (2014). Feedback-Driven Dynamic Invariant Discovery. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 362–372.