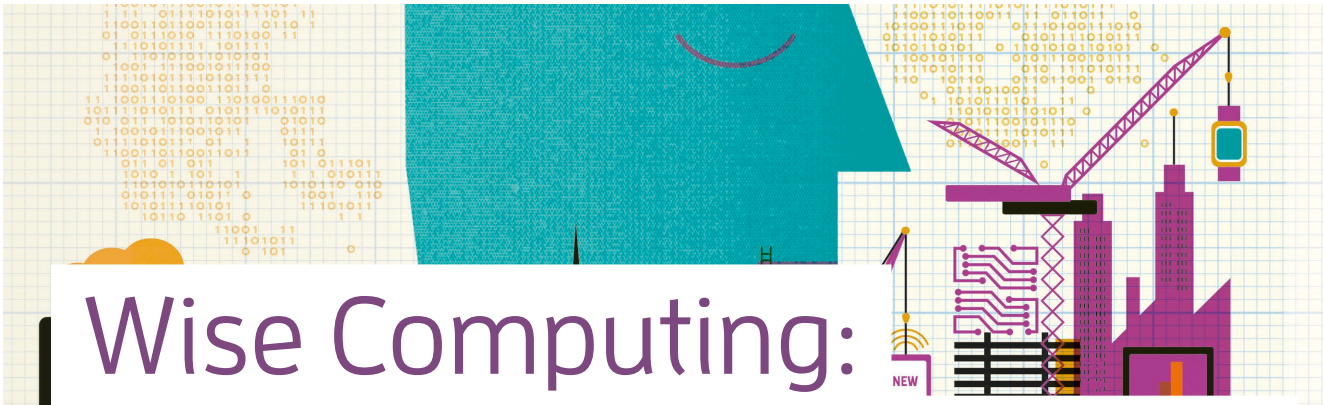

Wise Computing: Toward Endowing System Development with Proactive Wisdom

David Harel, The Weizmann Institute of Science
Guy Katz, Stanford University
Rami Marelly and Assaf Marron, The Weizmann Institute of Science

A broad, long-term research project is described that will lead to the computer becoming an equal member of the system-development team, continuously making proactive contributions, akin to those expected from an experienced and knowledgeable customer or user, a conscientious QA engineer, a strict regulatory auditor, an engineering team leader, or the organization's CTO.

Developing reliable reactive systems is difficult and error-prone. Deliverables can fail, bringing in their wake disastrous results, far worse than exceeding budgets and time schedules. Consequently, a tremendous amount of work has been put into simplifying and automating key system and software engineering tasks. This has included high-level languages and methodologies for modeling and programming, and methods and tools for requirements engineering, test generation, and verification.

However, humans are still limited in their effort to maintain a comprehensive picture of the elements and behaviors of a complex system, and of the relevant domain knowledge. This is exacerbated by the state explosion problem, which prevents exhaustive analysis of all possible behaviors. New algorithms, languages, and tools contribute somewhat to tackling these issues, but the smooth development of reliable complex reactive systems remains a major, and critical, moving target.

The severity of this gap is bothersome not only to engineering and computer-science professionals. Severe problems are found routinely in new versions of popular everyday products, ranging from failing to meet some basic customer need, through malfunctioning

0018-9162/18/$33.00 © 2018 IEEE

in a routine use case, to having a hard-to-find but dangerous security vulnerability. It is particularly disconcerting in view of the fact that such flaws or imperfections are discovered quite quickly by expert critics and reviewers.

The time has come for a major change in how complex systems are developed, providing a major empowerment of the development environment, and shifting the power balance between it and human engineers. Relying on powerful computational tools, on computerized versions of certain human-like competencies, and on knowledge in system development, the development environment will become a lot more creative and proactive, will interact with the developers in more natural ways, and will, ideally, join the development team as an equal partner. We term our vision *wise computing*, and it is built upon an earlier preliminary outline.[1]

PROACTIVE WISDOM

Imagine being invited by a commercial company or a research group to review their latest product, which they deem to be perfect, developed by their best engineers using the best tools and the best methodologies. As a specific example, say the product was a voice-operated medical-assistant robot in charge of home patient care. The intellectual challenge of discovering faults and omissions in the product under review is irresistible, and your mind is busy trying to come up with questions or raise issues that might hide previously unnoticed flaws, such as

- "What accents can the robot deal with? Will it understand a hoarse patient's speech?"
- "Will the noise of a loud TV confuse it? And what if a phrase spoken on the TV show happens to be a robot command?"
- "If I trick it, asking it to fetch a glass that is glued to the table, will the robot or the glass break? Will the robot be confused?"
- "Can the robot see and avoid a thin wire, say, when the patient is walking around with a phone or a medical device connected by a cord?"
- "It seems that voice commands to abandon the current task had to be repeated twice, which was rarely the case with other commands."
- "I have heard about a major security vulnerability in the open-source library that the robot uses for vision. Was the fix applied?"

Indeed, with today's engineering practices, such questions and challenges would be most welcome in review sessions and would become valuable in improving the product; failures associated with them would not necessarily embarrass the designers.

Wouldn't it be nice if the software development environment could help discover such issues itself, and then contribute to addressing them too?

As another example, consider the development of a system for controlling a chemical plant. The wise development environment will be able to identify, on its own, the absence of a requirement to notice the loss of communication with a temperature sensor. Further, it will explain that whenever current temperature is unknown, a certain door will be kept shut to reduce the risk of fire. At runtime, the system will recommend that before opening the door manually, one should check for fire on the other side. This interaction will be carried out using high-level problem-domain abstractions, similar to the English terms used in the list of examples above, independently of the technical terms used within the system model.

The development environment will be able to answer what-if questions, such as "what happens if the temperature sensor malfunctions?" as well as check a variety of behaviors and properties on its own, proactively alerting developers to problems; for example, that an alarm device previously dedicated to one situation is used also for another situation, possibly confusing human operators. It would also propose changes, explaining their impact and advantages.

During system maintenance, requests for new functionality—such as to deal with a new regulation for remote monitoring—would be stated in intuitive textual or visual interfaces. The development environment would then list affected components and propose required modifications. Indeed, even the mere confirmation that such functionality does not already exist would be extremely valuable.

THE WISE COMPUTING VISION

Wise computing extends the vision of MIT's Programmer's Apprentice (PA) project (circa 1970s–1990), which aimed to assist expert programmers by automatically developing code for requirements that follow known patterns (clichés):

> *The long term goal [of the PA] is to develop a theory of how expert programmers analyze, synthesize, modify, explain, specify, verify, and document programs.*

This is basic research at the intersection of artificial intelligence and system engineering.[2]

In the outlook toward wise computing, we seek to achieve and exceed the above goals via what we term a *wise development suite* (WDS). Specifically, today, human–machine interaction in system engineering relies mostly on a one-way initiative: the humans instruct or query the computer, while the computer "just" makes an effort to understand and do what the humans had in mind. Moreover, the competencies of the computer are presently more limited than those applied by humans when planning new systems or discussing development tasks. In the future, the WDS will initiate actions and suggestions based on deep knowledge and understanding of a broader range of goals and constraints than is possible today; in particular, covering knowledge that is not captured in the specification or code of the system at hand but, rather, comes from domain expertise or general world knowledge and human experience.

This forward leap will be enabled both by new research, described later, and by leveraging and integrating advances in hardware speed; programming and modeling formalisms; verification and analysis algorithms; program synthesis; natural language processing; machine learning; and knowledge acquisition and processing.

While it is only natural to conjecture that the hurdles to wise computing are insurmountable and that the dream of a WDS is just that, a dream, the validity of the current pursuit is supported by various efforts. For example, soon after our January 2015 preliminary technical report on wise computing was published in arXiv.org,[1] a paper by the PA team showed a renewed interest in the PA line of research, describing new capabilities in natural language processing, domain-specific knowledge, and code generation.[3] Another example is the work by Hadas Kress-Gazit's group,[4] in which a reactive robot controller was synthesized from natural language specifications, and its desired and undesired behaviors were automatically predicted and discussed with developers. In industrial software development, automated interactive analysis is becoming more common, with notable examples including Google's Tricorder, Facebook's Infer, and VMWare's Review Bot tools. Using static analysis, these tools automatically check for common types of errors and for violations of coding practices. Key tenets of these efforts, which must also be carried over to a WDS, are smooth workflow integration, efficiency, and scalability. The design of our proof-of-concept framework[5] is indeed aimed at achieving these properties. A more general call for action in this area was also expressed in a 2015 column in the *Communications of the ACM* by Vint Cerf (then-president of the ACM).[6]

IN INDUSTRIAL SOFTWARE DEVELOPMENT, AUTOMATED INTERACTIVE ANALYSIS IS BECOMING MORE COMMON.

Further discussion of related work can be found in the supplementary material at www.wisdom.weizmann.ac.il/~harel/IEEE.wisecomputing.

The wise development suite

We now describe how a WDS will be involved in the development of complex, yet typical, reactive systems. (Extending the ideas to other kinds of systems will be described as part of future research.) The WDS will be proactive throughout the development process, often assuming roles typically performed by humans (see Figure 1), as follows:

- *Requirements*. It will participate in the elicitation, formalization, validation, and iterative enrichment of requirements. Most notably, it will be able to notice requirements that were omitted—whether missed, lost, assumed to be obvious, or excluded intentionally but without adequate documentation.
- *Design*. During design and implementation, it will dynamically observe and analyze the system, using symbolic executions, simulations, and actual executions in controlled environments.
- *Testing and verification*. Throughout development, it will attempt to incrementally test and verify the system, constantly exploring

behaviors, both exhaustively and via suspicious use cases that it will proactively propose.

- *Detection of emergent properties*. It will detect undesired and conflicting behaviors, as well as behavior patterns that were not explicitly specified. For the latter, it will assess whether they are problems or advantages.
- *Cause and effect analysis*. It will be able to hypothesize about causes of otherwise unexplained behaviors and, where needed, automatically offer fixes or constraints that avoid them.
- *Documentation*. It will create models and documentation to share its growing knowledge with humans at appropriate abstraction levels.
- *Enabling runtime capabilities*. Per meta-requirements, or constraints added late in development, it will enhance the target system to monitor itself during execution, constrain behaviors, report problems, look ahead, and learn from its own past experience. It will also interact with users and with other systems to explain its actions and to accommodate external needs; for example, displaying on a door in a chemical plant why it is closed and what action can cause it to open.
- *Mode of interaction*. It will proactively initiate many of its interactions with the human engineers, using visual representations, natural language, pseudocode, and conventional code, and exploiting its ability to employ multiple levels of abstraction suitable for a wide range of human and machine stakeholders. As an example, consider the following sentence, which combines low-level code elements, system-level behaviors and specifications, and external user needs: "When this bit is turned on, that system-busy light starts blinking, telling users they cannot enter new commands."

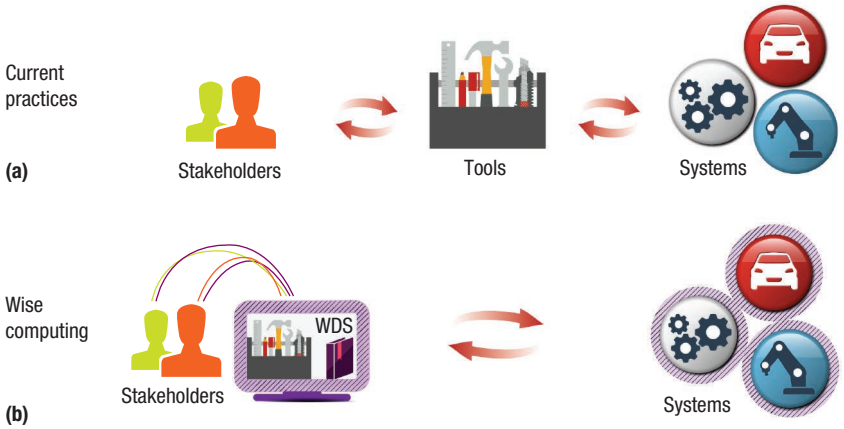

FIGURE 1. System development lifecycles. Compared to current practices (a), (b) the wise development suite (WDS) joins the stakeholders and developers as an equal, knowledgeable, proactive partner throughout the lifecycle of system and software development projects.

Benefits

The most immediate benefits of having wise development suites will be, of course, a significant reduction in the development time and cost of complex systems, and much improved system quality. This will also increase user and regulator confidence in systems, further expanding development and adoption of innovative solutions. Moreover, we believe that in the further future, we will experience new dimensions of innovation as rich new capabilities and new ranges of safety will be initiated (and often invented!) by wise development environments rather than by humans only.

TOWARD GETTING IT DONE

How will such a WDS be built? We believe that the dream does not require a yet-to-be-discovered quantum leap from the state of the art. Instead, it can be accomplished incrementally, with research that, considered as a whole, might very well be of breakthrough nature, but which will also rely and build upon many existing ideas and tools.

The creation of a powerful initial WDS is based on three main cornerstones:

- collecting, representing, and structuring knowledge about systems and problem domains through a *common formalism (CF)*;
- conducting human-computer discourse about systems in natural and appealing ways through a special *interaction language (IL)*; and
- rigorously analyzing and drawing conclusions from these knowledge items and interactions using a dedicated *analysis engine (AE)*.

Naturally, these three research directions will have to be integrated in a comprehensive, generic, scalable, easy-to-use, and open tool, together with a corresponding methodology. We elaborate below, and also refer the reader to our article "Six (Im)possible Things before Breakfast: Building-Blocks and Design-Principles for Wise Computing" for an additional discussion.[7]

The common formalism: Representation and structuring

Wise computing will require us to develop application-agnostic means, accompanied by rigorous semantics, for representing structure and behavior of systems and their environment in executable specifications. This should cover all facets of relevant information, such as behavioral and structural, discrete and continuous, deterministic and stochastic, object-centric and interobject, mainline behaviors and exceptions thereto. The representation will use multiple levels of abstraction, modularity at several levels of granularity, and separation of concerns. It will also have to address the need for intuitive human–computer interaction, for fast and automated mathematically oriented analysis of all artifacts, and for easy incremental refinement and enhancement of specification elements.

The CF that we propose (and have begun to develop) will enable the WDS to see the "big picture" during all development stages, via constant automated analysis of the entire project. For example, for an autonomous car project, using knowledge that includes, among other things, physics and traffic laws, it will be able to discover all possible causes for the car to overturn, including road-related issues, weight distribution, and driver actions. Central to the CF are extensive, formally annotated, connections between entities, enabling rigorous application of external knowledge and free association, where earlier interesting observations can readily drive a more focused automated exploration of related aspects.

> THE CF THAT WE PROPOSE (AND HAVE BEGUN TO DEVELOP) WILL ENABLE THE WDS TO SEE THE "BIG PICTURE" DURING ALL DEVELOPMENT STAGES.

Examples of the search for such new formalisms can be found in the introduction of hierarchical, concurrent state machines, namely, Statecharts (see, for example, Harel's early work on Statecharts and Statemate[8,9]), and in our group's research on scenario-based programming (SBP), where the programming idioms are aligned with the interobject scenarios that humans often use in describing reactive systems in requirement documents.

To further elaborate on the latter, in a 2008 paper, it was suggested that programmers can be liberated from the constraints presently imposed by specification, design, and verification, and that programming computers can become much closer to the way we communicate with other persons, such as children, employees, or students, when trying to bring about desired behavior in others.[10] This dream of liberating programming has yielded a large body of work on SBP, which originated in the language of live sequence charts (LSC),[11,12] later being extended into procedural languages such as Java and C++ (see, for example, "Behavioral Programming"[13] and www.b-prog.org). It allows natural playing-in of desired and forbidden scenarios, and techniques and tools such as runtime look-ahead, formal verification, automatic program repair, and more.

Regardless of the specific formalism chosen, it seems that to be useful for knowledge representation in a wise computing environment, the CF would have to allow specifying directly the execution scenarios that guide the application of such knowledge to the system in question. For example, an expert's knowledge of open-source libraries suitable for a certain application might be represented by the WDS as a combination of a data store of open-source solutions with an analysis and recommendation procedure. This approach might also blur the boundary between a system and its environment. For example, in the patient-care assistant robot case, a WDS's growing knowledge of what new obstacles could look like might be stored in the same way as the system's scenarios that actually recognize and navigate around such obstacles. Since the Statecharts and SBP formalisms

deal adequately with many of the ways humans describe behavior in requirement documents and system specifications, we believe that with certain extensions (Assaf Marron's paper,[14] for example, describes a possible approach for combining them for wise-computing purposes), they can prove valuable in specifying many kinds of external knowledge and the meta-heuristics needed for a WDS.

An important part of the research needed for the CF component of the WDS involves ways to incorporate and use general software-engineering and domain-specific knowledge (see www.wisdom.weizmann.ac.il/~harel/IEEE.wisecomputing for discussion of prior research in this area, including knowledge-based software engineering,[15] domain-oriented design,[16] Wolfram Alpha,[17] and Wolfram Language[18]). The necessary knowledge can be found in industry standards and reference documents, electronic education materials, simulators and games containing world knowledge, and more. For example, to mimic the acute observer's question regarding wires disrupting the patient-care assistant robot, one could

- use machine learning to be able to identify, from documents or images, when a system includes a robot that has to move around in a typical indoor environment; and
- specify a fixed scenario that asks whether the robot would trip over invisible obstacles like wires or unevenly carpeted floors.

Even when not directly applicable to the case at hand (for example, this is a factory floor with no wires or carpets, and the robot is already programmed to handle the relevant obstacles), in many cases, just posing the question can create a flurry of development actions, such as enhancing and refining the test cases or documenting operational constraints and user responsibilities more conspicuously.

The construction of such knowledge bases for the WDS will benefit from collaborative efforts led by humans and by software agents. Centralized knowledge sources will likely be complemented by distributed search-and-recommend techniques, where advice regarding the system at hand is discovered from the automated, just-in-time examination of code, models, and documentation of similar systems available on the Internet (see, for example, work by Robillard et al., Yadid and Yahav, and Zagalsky et al.,[19–21] and the discussion at www.wisdom.weizmann.ac.il/~harel/IEEE.wisecomputing).

The interaction language: human–computer discourse

In addition to natural representation of the CF, a fully capable WDS requires a new IL that enables human–computer collaboration on all system artifacts and on all relevant knowledge, similar to the way project team members communicate among themselves. For example, when using the WDS to design an autonomous vehicle, the following conversation between a human (H) and the WDS (W) should be possible, almost verbatim:

H: "When the driver presses the brake pedal the brake light should turn on."

W: "Does this apply also to when the engine is off?"

H: "No."

W: "Done."

And later:

H: "I saw the car rolling downhill and stopping; how did it stop?"

W: "The driver pressed the brake pedal."

H: "Why didn't the brake light go on?"

W: "The engine was off."

H: "The light should have turned on anyway."

W: "But earlier you said no."

H: "Ah, right. So please remove this exception."

Such interactions deal simultaneously with requirements, programming, what-if and causation scenarios, and self-reflection, and they use the right level of detail for each (as opposed to, for example, showing an arcane and excessively detailed log of events). The IL will have formal extensible semantics, and its underlying engine will interact with the user to resolve ambiguities.

It is important to distinguish between the CF and the IL. Ideally, the common formalism stores everything that we know about the system, in a computer-analyzable manner. The interaction language is what humans will use to modify this storage and retrieve information from it. For example, a portion of the CF might include a large automaton, or graph, with many state nodes and transition edges. In

addition to merely viewing visual representations of this structure, the user will be able to interact about it with the CF via queries and commands such as "add an edge labeled E1 from state S1 to state S2," or "is there a path from S1 to S3 that does not visit S2?" The IL might also serve as a presentation mechanism for the CF, so that when such a graph becomes unwieldy, for example, it might be displayed as a list of text lines, or as table entries that can be sorted and traversed in various ways.

Although the above conversation script might seem like general futuristic machine intelligence, we believe that our far more restrictive subject matter—system development—renders it within reach. The piecewise assertions about complex structures like those about properties of states, transitions, and labels thereof can combine with modular compositional techniques like scenario-based programming to formally define artifacts not readily available in existing languages, such as "a possible cause," "a possible effect," "an exception to a rule," "a synonym," "a situational context," and so on. Once these entities become well-defined, recent advances in programming using controlled natural language, visual formalisms, and programming by example would facilitate the first steps. In subsequent steps, machine learning could be used to translate from a wider (but still limited) range of human-provided natural language sentences to IL idioms, and to transform events and emergent properties in system behavior, as observed in simulations and in real execution (such as a car having stopped, or a light not having turned on, or the existence of an unexpected noise) into CF entities that can be presented, analyzed, or queried.

The analysis engine: The core of the WDS

Although features such as naturalness, unbounded abstraction, and separation-of-concerns of the CF and the IL will enable many fast knowledge-dependent analyses, we propose to create an AE to serve as the core of the WDS. This part of our vision, the "brain" of the WDS, will utilize and enhance the most advanced computational techniques available (for example, model-checking, program synthesis, constraint solving, and machine learning), to proactively and interactively mimic intricate human processes, such as free association, initiative, prioritization, and even certain kinds of creativity. The AE will constantly analyze all project artifacts, relevant world knowledge and relationships therein as encoded in the CF. It will proactively look for anomalies (for example, deadlocks, race condition, undue delays), interesting unexpected properties (for example, event correlations), and desired and undesired patterns suggested by experience and by external knowledge. When it identifies relevant issues, the AE will drive deeper investigation, and suggest fixes.

Using the IL, the AE will present its findings clearly, concisely, and using appropriate abstractions, and will regularly receive work guidance, new requirements, simplifying assumptions, and so on. For example, the AE will be able to detect that an autonomous car is not aware of a new traffic sign or that it sometimes slows down unnecessarily, creating, in both cases, a collision risk. The AE will then be able to automatically offer program repairs, such as turning the legal requirement associated with the traffic sign into program code, or detecting and removing the reason for the slow down.

> **A CHALLENGE TO ALMOST ANY KIND OF ANALYSIS IS THE COMPUTATIONAL COMPLEXITY THAT RESULTS FROM THE STATE EXPLOSION PROBLEM.**

An obvious challenge to almost any kind of analysis is the computational complexity that results from the state explosion problem. The WDS and the AE will be specifically designed as follows to help cope with, or circumvent, this challenge in many cases.

The CF will be used to encode useful heuristics, much like the means human developers would use to cope with present tool limitations in ordinary projects. For example, if the WDS is unable to verify a commodity product that uses face recognition in all possible lighting combinations specified in the CF, the stakeholders might specify that a certain helpful lighting arrangement can be assumed, substantially simplifying the verification. Further, existing machine learning techniques can be readily applied to

extend such knowledge. One might learn the conditions under which the system has a low success rate or takes longer than would be allowed in production for its recognition task, or even learn the environment settings under which the verification process requires more than the time allotted to it. The WDS can then propose constraints to be added to the CF and to the system's documentation. Machine learning can also be used to find commonalities between automatically synthesized scenarios and CF elements, and to provide useful generalizations and abstractions. Nevertheless, vast ground can be covered even with heuristics that are coded manually, since they can be immediately shared across all engineering projects; the quantity of such heuristics is manageable because they emanate from "human-scale" development activities, such as a paragraph in a requirement document, a discovered runtime bug, or a comment from an expert reviewer; and, like humans, the WDS would generalize such heuristics (for example, if the robot tripped on a nearly invisible wire, the heuristic is likely to look for other kinds obstacles to physical motion that are not readily detectable by the robots various sensors, such as visual, lidar, or sonar). Other facilities of the WDS will help add such requirements to the user manual, and help implement a runtime alert for when such conditions are not met. Furthermore, the abstractions, hierarchies and compositional traits of the CF will enable limiting certain analyses to appropriate components or subareas of the system. (We have already obtained some promising preliminary results in applying SMT solvers to the task of such compositional verification; see, for example, "Theory-Aided Model Checking of Concurrent Transition Systems"[22] and references therein.)

The AE will not have to be an expert on everything. It will be designed to support multiple concurrent targeted analyses, each of which might be limited, or guided by specific heuristics, in a way that mimics the special limited-but-focused attention a reviewer might apply to a particular aspect of the system. Such analysis skills will be added to the AE incrementally, and could benefit from collaboration and reuse.

The AE will not be expected to carry out all manner of analysis. Thus, it would not always be required to find highly complex scenarios that could lead to a deadlock, or detect complex repeating patterns among a large number of events. Discovering these very often involves advanced verification or machine learning techniques, and only rarely can be discovered merely by applying what one would call human wisdom. Instead, the AE will have algorithms that are geared to being able to ask simpler, but unexpected, questions, as in the examples throughout this article.

The previous point notwithstanding, the modular nature of the AE will enable it to be continuously enriched with the latest state-of-the-art techniques in verification, synthesis, and machine learning.

An interim discussion

Now that we have discussed the CF, IL, and AE, one might ask whether success in the research needed to build them would be sufficient for achieving the high aspirations of wise computing. Also, are they necessary? For example, could other techniques that are being developed, such as deep learning and program synthesis, combine to accomplish the same? And finally, are we actually saying that much of these components already exist today in some other guise?

As for novelty and necessity, at present no common methods or methodologies can capture, automate, generalize, and continuously enhance competencies that mimic an expert reviewer's ability to make quick, yet pithy, observations (even if only in a narrow area) regarding a system about which they have only partial information, and using terms and abstractions that are understood by all stakeholders. Moreover, there are no ways to exceed the abilities of such experts by applying these competencies also in exhaustive analysis of all system artifacts.

As for the sufficiency of the AE, IL, and CF, we note that wise computing does not aim for automated design, synthesis, or verification of entire systems, but rather to provide expert validation and advice continuously from the beginning of, and throughout, the development process.

A smaller-scale question is whether the search for "one language for everything" is realistic. We believe that with a liberal enough definition of "language," the answer is positive. First, in a way, such languages do exist, such as Turing machines and English, in which reactive systems can be described with as much or as little detail as desired. Second, clearly a particular language might not be sufficiently intuitive to humans or sufficiently formal for computer interpretation. However, these criteria are fluid, and because there is a plethora of excellent languages and idioms that collectively cover the relevant notions from knowledge, logic execution, sensing and actuation, coordination and development meta-operations, and so on, it seems

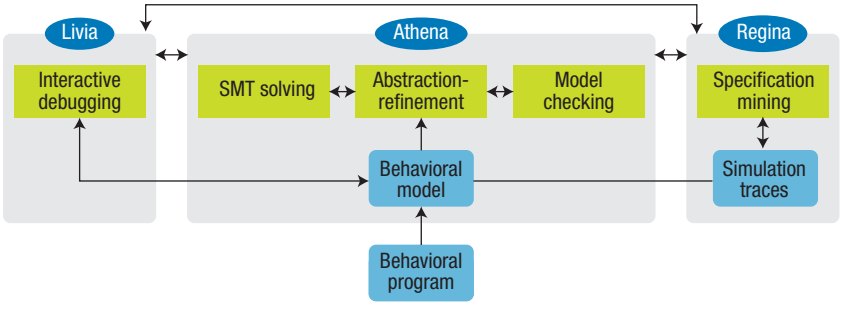

FIGURE 2. A high-level overview of the three sisters. The developer provides a behavioral program, from which Athena extracts a behavioral model. She then analyzes this model using abstraction-refinement, model checking, and satisfiability modulo theory solving. Athena also shares the behavioral model with her sisters—with Regina for the purpose of specification mining and with Livia for interactive debugging. The sisters also exchange information with one another. For instance, Regina might ask Athena to attempt to formally prove an emergent property that she found.

that the raw materials for creating a syntax and semantics for a useful general CF are indeed already available.

Note that wise computing does not require breakthrough advances in general natural language processing and in code-generation from casually written requirement documents, for example. The WDS will be able to understand humans with regard to system development activities, using only state-of-the art natural language processing, by relying on things like controlled subsets of natural language,[23] domain-specific phrasing templates and itemized checklists, interactive disambiguation, model annotations, tables or graphs of technical and natural language synonyms and richer semantic networks (like abstractions and refinement relationships), vocabularies that are extracted from actual models, and more. All of these can be gradually acquired by programming, by extracting information from documents and models, or by machine learning. Additionally, we believe that smart human users will be able to quickly learn from examples and experience, and adopt as "natural" the idioms that are more readily understood by the WDS.

We do assume, though, that the system under development is engineered in ways that are amenable to composition, abstraction, requirements-tracking, verification, and automated repair. This aligns with our group's active research on such system engineering goals over several decades. Indeed, the initial work described in the next section was carried out using a particular architecture that aims at such purposes.

SOME INITIAL WORK

We end this vision article by briefly describing a very modest and preliminary, proof-of-concept, wise development suite (a mini-WDS, or mWDS), which illustrates some of the ideas we aim at. A more detailed description of the mWDS and a case study we have carried out using it appears in "An Initial Wise Development Environment for Behavioral Models."[5] Also, up-to-date versions of the mWDS and the case study themselves, as well as prerecorded video clips demonstrating its main principles, can be found at www.wisdom.weizmann.ac.il/~harel/IEEE.wisecomputing. We particularly recommend that the reader take the time to view the two narrated demo clips therein.

A modest wise development suite

The mWDS is focused on accompanying the development of reactive systems using scenario-based programming, where system components are scenarios (also known as behavior threads). The system behavior is based on events that serve as abstractions of physical interactions with the environment, which, in turn, are implemented, say, with classical sensors and actuators. Scenarios control the behavior by requesting, blocking, and reacting to such events,[13] and they are executed in parallel, yielding cohesive system behavior. We focus on this model because it is simple and general; its idioms appear in other formalisms, such as certain kinds of publish–subscribe systems and supervisory control; and most importantly, it can sometimes combine intuitiveness with amenability to fast automated analysis (see, for example, the work of Katz et al.[22] and Harel et al.[24] and references therein), which are at the center of the wise computing vision.

Functional components of the mWDS: the three sisters

The present demonstration focuses on a human competence that appears most elusive: the uninitiated noticing and handling of emergent properties, that is, observed properties that did not necessarily appear in the requirements

or documentation—either as desired or as forbidden ones. To this end, we have constructed a mini-AE built from three new components, which we call "the three sisters"—Athena, Regina, and Livia—as shown in Figure 2.[5] We have also made some initial steps toward creating basic versions of the CF and IL, but these are omitted here due to lack of space.

Very briefly, Athena, named after the Greek goddess of reason, uses formal tools (model-checking, satisfiability modulo theory [SMT] solving) to prove observed system properties—properties that should be valid for all runs.

The more "regal" Regina can initiate executions of the system and control the other two wise-computing components. She runs multiple executions (actually, simulations in a safe sandbox), collects statistical information, and, in a form of specification mining, attempts to present interesting conclusions to the user. These conclusions might not be valid for all runs, but they still reflect the system's behavior during a large number of executions and can thus capture what will happen in typical runs.

Finally, Livia works in a live online fashion, monitoring the system as it runs and prompting the user when it appears to be acting in undesired ways—for example, if a thread seems to be "stuck." She can also monitor the system for the occurrence of specific scenarios (such as those marked by the user as undesired). In problematic or suspicious cases, Livia can launch a formal localized runtime analysis (using, for instance, bounded model checking), checking whether an error has truly occurred and why, or, conversely, finding a sequence of events that would allow the system to proceed correctly from its current state.

Workflow

The offline components, Athena and Regina, constantly, automatically, and proactively analyze the code and run simulated executions thereof, looking for emergent properties and assessing them. Such properties might be relevant even if they do not hold for all runs, as they can reconfirm correct behavior or draw attention to problems.

Athena and Regina run continuously as background processes. Whenever they detect a fresh compilation of the system, they take a snapshot of the code and begin to analyze it. Athena performs basic verification for general properties (like the absence of deadlocks), and system-specific ones. Checks for additional properties, such as loops of generating system events without reacting to environment events, can be added incrementally in a modular manner. Mimicking the earlier insightful observation about having to repeat certain commands can be generalized as follows: look for scenarios or behavior patterns of "always after E1 do E2," and "always between E1 and the next E2 there is an occurrence of E3." If there is no explicit specification of "always before E2 do E3," this might be a problem (and even if it is specified, this might be an error). She then verifies properties suggested by Regina, as described further below.

To accelerate the verification, and provide a safe, isolated simulation environment, Athena first generates an abstract model of each thread, based only on its external communication events and points of synchronization with other threads, "abstracting away" unshared internal thread states, and external interface actions (see a specific example in the next section). Athena then searches these models for certain patterns—such as semaphores, shared memories, sensors, and actuators—and partitions the threads into functionally related modules. For additional acceleration of subsequent property verification, Athena employs module-based abstraction-refinement techniques.[25] She temporarily abstracts away elements that seem unlikely to affect the outcome. If the verification fails, the counterexamples are checked against the full model, and, if they are spurious, namely false negatives, the abstraction is refined and the process repeats. We are currently in the process of integrating SMT-based (theory-aided) techniques into Athena (see, for example, Katz et al.[22]) to further improve performance.

Regina searches for system properties via specification mining. She runs multiple simulations of Athena's abstract model and looks for patterns, such as events that always (or never)

WE HAVE CONSTRUCTED A MINI-AE BUILT FROM THREE NEW COMPONENTS, WHICH WE CALL "THE THREE SISTERS"—ATHENA, REGINA, AND LIVIA.

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

FIGURE 3. Displays from the mini wise-development suite (mWDS) during development of a cache-coherence protocol application, showing emergent properties as observed by Regina, followed by Athena's conclusion regarding whether they hold or not. The arrows indicate event implication—that the occurrence of one event implies the occurrence of the other a short time earlier or later. For example, the first property observed by Regina is "The event of 'releasing Bus 1' implies, and is implied by, the event of 'Cache 2 requests Bus 1'." Then, using model checking, Athena determined that this does not always hold.

appear together; events that cause, prevent, or suspend the occurrence of other events; producer–consumer patterns, and so on. This is done under various assumptions, such as fair versus unfair event scheduling, that is, allowing and disallowing starvation of certain system or environments behaviors. We plan to enhance this capability with techniques and tools for test case generation guided by coverage goals.

At present, Regina looks only for simple properties, and as with Athena, additional ones can be readily added. Published techniques for mining scenario-based specifications from runtime event sequences are prime candidates for such extensions. Observed properties are immediately displayed to the user, who can immediately prune trivially correct ones ("Of course! This is what the system was programmed to do!") or trivially incorrect ones ("This pattern only applies to these particular runs, and I know why Regina incorrectly tried to generalize this, so there is no need to pursue it"), or follow up on hints of undesired behavior ("Never mind the generality of the observed pattern, even in these runs only this shouldn't have happened").

Regina then proceeds to check whether these properties hold in general, either on her own by checking statistically for additional simulations, or by passing them over to Athena for formal verification. Here, too, the user can intervene by dictating the kinds of verification to be carried out and their order, as well as additional properties to be checked. Figure 3 shows simple examples of the results displayed by Regina and Athena.

As the mWDS might continue its analysis indefinitely, the user is informed of verification results, including relevant counterexamples, as soon as they are available. Future validations can be enriched by the verified properties to ensure they survive system changes, or to study why other properties that seem desirable do not always hold.

Since exhaustive model checking or extensive statistical checking of many properties will very often be infeasible, we complement the user's selection and prioritization with heuristics. For example, the mWDS identifies groups of logically equivalent, symmetrical, or syntactically similar properties, and analyzes only one representative from each. Further, higher priorities are assigned to properties associated with facets that are prone to error, such as concurrency.

When Athena's verification runs out of memory or exceeds programmer-specified time limits, Regina can take over, using her more efficient but less accurate statistical methods.

When an undesired safety or liveness issue is discovered, the user may request Athena to synthesize a code fix in the form of an additional scenario.[25,26] Athena can also synthesize a monitor thread and add it to the program, in order to report at runtime if a certain property is violated. For example, recalling the example of tricking the home care robot into trying to lift a glass that is glued to the table, one such scenario can simply block the application of a pulling force that is greater than a certain threshold, thus preventing the breaking of the glass or the robot, and another can report when motion-related actions have not been completed within a given time limit, avoiding the appearance of the robot being stuck.

Athena also supports applying its ability to focus on behaviors that matter, toward thread optimization, such as the removal of unreachable code.

As described, the online part of the mWDS, Livia, is a runtime debugging assistant and is launched manually to accompany test and production runs of the final system.

A modest case study

We have evaluated our mini-WDS by using it to develop several programs, including the one we will present here: a cache-coherence protocol. Such protocols are designed to ensure consistent shared memory access in a set of distributed processors. Each processor

caches its results of memory reads, and when it writes a new value to the shared memory other processors invalidate their corresponding cache entries. Cache-coherence protocols are notoriously susceptible to concurrency related bugs—rendering them a prime candidate to benefit from a wise development environment. Figure 3 depicts a simple extract from the mWDS's listing.

An important question we considered when preparing the case study was whether programming a complex system while being aided by a proactive framework is convenient and/or useful. While such issues are highly subjective, we can report that we found the process useful, natural, and even enjoyable. Also, although the mWDS implements only a very modest portion of the wise computing vision, its integration into the development environment increased our confidence in the implementation's correctness. Specifically, the mWDS reported several concurrency-related issues that we had overlooked and had to repair. And having Regina and Athena identify properties that indeed seemed natural and expected but were neither obvious nor redundant, reassured us that we were indeed on the right track. Again, we recommend that the reader view the recorded demos referred to earlier.

The verification-acceleration techniques also helped. For example, in verifying the mutual exclusion property "Cache 3 cannot acquire Bus 2 repeatedly without first releasing it," Athena explored about 1 million states in about 27 minutes. In automatic and proactive abstraction-refinement, it abstracted away other buses and verified the property, exploring only 21,000 states in under 31 seconds.

ABOUT THE AUTHORS

DAVID HAREL is vice president of the Israel Academy of Sciences and Humanities, and a professor at the Weizmann Institute of Science. His research interests include logic and computability, software and systems engineering, modeling biological systems. Harel invented Statecharts, co-invented Live Sequence Charts, and has authored several books, including *Algorithmics: The Spirit of Computing* and *Computers Ltd.: What They Really Can't Do*. He received the ACM Karlstrom Outstanding Educator Award, the Israel Prize, the ACM Software System Award, the EMET Prize, and five honorary degrees. Harel received a PhD in computer science from MIT. He is a Fellow of ACM, IEEE, and AAAS, a member of the Academia Europaea and the Israel Academy of Sciences, and a foreign member of the US National Academy of Engineering and the American Academy of Arts and Sciences. Contact him at dharel@weizmann.ac.il.

GUY KATZ is a postdoctoral research fellow at Stanford University. His research interests lie at the intersection of software engineering and formal methods, in particular, modeling paradigms that are useful and friendly to programmers while also amenable to formal analysis, verification, and program repair. To this end, he has been studying the properties of various concurrency idioms, and how these properties can be leveraged by advanced program analysis tools such as SMT solvers. Katz received a PhD in computer science from the Weizmann Institute of Science. Contact him at guyk@cs.stanford.edu.

RAMI MARELLY is co-founder of Cue, a consulting firm in system engineering, business development, and project management. Before his retirement, he held key positions in the Israeli Air Force technological directorate. Marelly's research focused on specifying and executing behavioral requirements by co-developing the Play-in/Play-out approach. He received a PhD in computer science from the Weizmann Institute of Science. Contact him at ramimarelly@gmail.com.

ASSAF MARRON is a researcher in the Weizmann Institute of Science's Computer Science and Applied Mathematics Department. Prior to joining the Weizmann Institute, he worked in senior management and technical positions at leading companies, including IBM and BMC Software. Marron's research interests include software engineering, scenario-based programming, machine learning, and information visualization. He received a PhD in computer science from the University of Houston. Contact him at assaf.marron@weizmann.ac.il.

The notion of machines that invent, design, and build other machines exists mostly in the realm of science fiction. However, we believe that computers can definitely help humans in doing so, and the feasibility of the wise computing vision lends support to that belief. We are encouraged by our preliminary results on the common formalism, interaction language, and analysis engine, as well as by the mini-WDS integrative prototype.

Clearly, a tremendous amount of work remains to be done, and we hope that researchers and practitioners will be inspired to undertake major efforts that would help bring the vision to fruition.

ACKNOWLEDGMENTS

This work was supported by a grant from the Israel Science Foundation, by the Philip M. Klutznick Research Fund, and by a research grant from Dora Joachimowicz. We thank

Yishai Feldman, Orna Kupferman, Shahar Maoz, Moshe Vardi, Gera Weiss, Amiram Yehudai, and the anonymous reviewers for their valuable comments.

REFERENCES

1. D. Harel et al., "Wise Computing: Towards Endowing System Development with True Wisdom," arXiv, 23 Jan. 2015; arxiv.org/abs/1501.05924.
2. C. Rich and R. Waters, "The Programmer's Apprentice: A Research Overview," *Computer*, vol. 21, no. 11, 1988, pp. 10–25.
3. H. Davis, B. Shrobe, and R. Katz, *Towards a Programmer's Apprentice (Again)*, CBMM memo no. 030, Center for Brains, Minds and Machines, 2015.
4. C. Lignos et al., "Provably Correct Reactive Control from Natural Language," *Autonomous Robots*, vol. 38, no. 1, 2015, pp. 89–105.
5. D. Harel et al., "An Initial Wise Development Environment for Behavioral Models," *Proc. 4th Int'l Conf. Model-Driven Engineering and Software Development* (MODELSWARD 16), 2016, pp. 600–612.
6. V. Cerf, "A Long Way to Have Come and Still to Go," *Comm. ACM*, vol. 1, no. 58, 2015, p. 7.
7. A. Marron et al., "Six (Im)possible Things before Breakfast: Building-Blocks and Design-Principles for Wise Computing," *Proc. 19th Int'l Conf. Model Driven Engineering Languages and Systems* (MODELS 16), 2016, pp. 94–100.
8. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, 1987, pp. 231–274.
9. D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.*, vol. 16, no. 4, 1990, pp. 403–414.
10. D. Harel, "Can Programming Be Liberated, Period?," *Computer*, vol. 41, no. 1, 2008, pp. 28–37.
11. W. Damm and D. Harel. "LSCs: Breathing Life into Message Sequence Charts," *J. Formal Methods in System Design*, vol. 19, no. 1, 2001, pp. 45–80.
12. D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer, 2003.
13. D. Harel, A. Marron, and G. Weiss, "Behavioral Programming," *Comm. ACM*, vol. 55, no. 7, 2012, pp. 90–100.
14. A. Marron, "A Reactive Specification Formalism for Enhancing System Development, Analysis and Adaptivity," *Proc. 15th ACM-IEEE Int'l Conf. Formal Methods and Models for System Design* (MEMOCODE 17), 2017, pp. 161–164.
15. K. Benner, "Knowledge-Based Software Assistant," *Knowledge-Based Systems*, vol. 1, no. 4, 1988, pp. 221–226.
16. G. Fischer, "Domain-Oriented Design Environments," *Automated Software Eng.*, vol. 1, no. 2, 1994, pp. 177–203.
17. S. Wolfram, "Wolfram Alpha," 2018; www.wolframalpha.com/about.html.
18. S. Wolfram, "Wolfram Language," 2018; www.wolfram.com/language.
19. M. Robillard, R. Walker, and T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software*, vol. 27, no. 4, 2010, pp. 80–86.
20. S. Yadid and E. Yahav, "Extracting Code from Programming Tutorial Videos," *Proc. Int'l Symp. New Ideas, New Paradigms, and Reflections on Programming and Software*, 2016, pp. 98–111.
21. A. Zagalsky, O. Barzilay, and A. Yehudai, "Example Overflow: Using Social Media for Code Recommendation," *Proc. 3rd Int'l Workshop on Recommendation Systems for Software Eng.* (RSSE 12), 2012, pp. 38–42.
22. G. Katz, C. Barrett, and D. Harel, "Theory-Aided Model Checking of Concurrent Transition Systems," *Proc. 15th Int'l Conf. Formal Methods in Computer Aided Design* (FMCAD 15), 2015, pp. 81–88.
23. D. Harel and M. Gordon, "Steps Towards Scenario-Based Programming with a Natural Language Interface," *Proc. ETAPS Workshop: From Programs to Systems*, 2014, pp. 129–144.
24. D. Harel et al., "On the Succinctness of Idioms for Concurrent Programming," *Proc. 26th Int'l Conf. Concurrency Theory* (CONCUR 15), 2015, pp. 85–99.
25. G. Katz, "On Module-Based Abstraction and Repair of Behavioral Programs," *Proc. 19th Int'l Conf. Logic for Programming, Artificial Intelligence and Reasoning* (LPAR 13), 2013, pp. 518–535.
26. D. Harel et al., "Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs," *Trans. Computational Collective Intelligence*, vol. 16, 2014, pp. 1–33.