Six (Im)possible Things before Breakfast*: Building-Blocks and Design-Principles for Wise Computing

(Work-in-progress report)

Assaf Marron[1], Brit Arnon[1], Achiya Elyasaf[1], Michal Gordon[4], Guy Katz[2], Hadas Lapid[1], Rami Marelly[1], Dana Sherman[1], Smadar Szekely[1], Gera Weiss[3], and David Harel[1]

[1]Weizmann Inst., [2]Stanford U., [3]Ben Gurion U., [4]Holon Inst.

Abstract. Despite many advances in automating system and software engineering, the knowledge, skills and experience of human engineers and domain experts are, and will continue to be, critical factors in the success of complex system development projects. In a new research direction we term *towards wise computing* we aim to make some of these abilities available to all project-team members, at any time, by adding several radically-new capabilities to the development environment. For example, the development environment will be able to, on its own, notice and suggest a required action for, an unexpected, unspecified, emergent system property. A report on the ultimate wise-computing vision and an initial demonstration of the desired functions have been published separately. Clearly, advanced tools, such as formal verification and synthesis, machine learning and automatic reasoning, which were not previously available, are now maturing and will be essential to wise computing. Yet, a comprehensive feasibility plan was not shown until now, and despite the above arsenal of tools, a challenge is often presented: is it indeed possible to automate some critically relevant abilities of humans, like free association, applying general knowledge to a wide variety of specific cases, or "thinking outside the box" to find a solution to a problem. In this report we outline research directions for several software capabilities and design principles that we have begun to work on, and which we believe can remove key remaining obstacles to feasibility of wise computing.

1 The Wise-Computing Challenge

Rich knowledge and sophisticated skills, acquired by experienced engineers over years of training and hard-earned practical lessons, are key ingredients, and even critical success factors, in complex software and system engineering projects. In a new research vision that we term *towards wise computing*, we aim to endow the software development environment with those engineering abilities that are particularly considered unique to humans and make them available to the project team at all times. Some of the relevant capabilities we are most interested in are:

- Free association: i.e., when an event can trigger a distantly related action
- Applying engineering and domain expertise: e.g., to new and varied situations
- Noticing irregularities: e.g., answering "what's wrong with this picture?"

* With apologies to Lewis Carroll

- Creative problem solving: i.e., "thinking-outside-the-box"
- Seeing a bigger picture: e.g., considering problems, causes, work methods, etc.
- Adapting to new demands: e.g., new work methods due to cyber risks

As a specific example, in developing an autonomous car, a wise development environment will be able to do the following spontaneously: (1) identify the modules participating in a communication protocol and apply model-checking to that part of the system only, in order to verify its correctness; (2) notice, in testing or in final use, without it having been specified for the system, that the car occasionally slows down, slightly, but still unnecessarily, and then search for causes and solutions; and (3) use domain-specific information, such as classical mechanics or applicable traffic laws, in code generation, in test generation and in formal verification.

A description of the full vision of wise computing, as well as an initial proof-of-concept demonstration of the capabilities that such a wise development environment will have, has been published separately [4, 5]. Related work is presented later in this section.

Wise computing will rely on advanced computationally-intensive capabilities, such as formal verification using model-checking and SAT/SMT solving, program synthesis, machine learning, pattern recognition, AI planning and reasoning, program static analysis, and automatic test generation. These sophisticated tools, which were not previously available, are now maturing and will play a central role in wise computing. Yet, a comprehensive feasibility plan for wise computing was not shown until now, and despite the availability of the above arsenal of tools, a challenge is often presented: can the human skills described above, and others that are required for accomplishing the wise computing vision, really be automated? Do we know enough about the human mind to even start pursuing such an ambitious goal?

The purpose of this short paper is to describe elements of our present research, which show that in the area of software and system engineering ("SE" hereafter) this is indeed feasible, and that such human activity can be mimicked, independently of the need to understand the biology and psychology behind it.

In a nutshell, we believe that to harness and integrate into a development environment ("DE" hereafter) the latest analytical tools and the not-yet-formalized knowledge and skills of humans, will require the development of new structural and behavioral principles. These will involve issues in system modeling, in knowledge representation, and in the languages used by humans and machines to converse about systems. As part of our broader research plan for wise computing, this paper is dedicated to these enabling building blocks. In the following sections we briefly present several of these as particular research directions, discuss their merits and feasibility, and describe preliminary results supporting their continued pursuit.

Related Work. Due to space constraints we refer the reader to [11] for a detailed discussion of related work. Automating unique human competencies has been a goal of several research projects, most prominently, the Programmers Apprentice and Requirements Apprentice projects at MIT (ended early 1990s) [10, 9]. There, the computer was intended to automatically generate code from requirements where specific text patterns (clichés) were recognized. A paper published in 2015 by the same team shows a renewed interest in these concepts, and

demonstrates capabilities in natural language processing, domain-specific knowledge, and code generation [2]. Another related recent work is [8], where natural language specifications are used to synthesize and iteratively develop a reactive robot controller, while predicting and discussing desired and undesired behaviors.

The main contributions of the wise computing vision and of the feasibility plan presented here are the breadth of target system applicability, range of human skills addressed, and the search for fundamental game-changing engineering concepts that would streamline this undertaking.

2 Integrative Cross-function Models

Common development practices and methodologies often rely on different models and languages for requirements, specifications, actual system code, test scenarios, simulation results, bug reports, and system documentation. We propose to research a method for modeling all the above artifacts in a consistent way that connects all of them throughout the project life-cycle, in a single model or few closely integrated ones. Combined with extended aggregation and reflection capabilities, discussed in Sec. 3, this will help in application of knowledge in multiple contexts, and in "seeing the bigger picture" of the system, the domain, and the methods and tools used in development. Externally-specified connections between artifacts, heuristic searches and systematic explorations of such models will be able to exhibit behavior that may mimic the human ability for free association.

To that end, we propose to research integration of statecharts [3] and scenario-based programming (abbr. SBP hereafter, a.k.a behavioral programming; see introduction of SBP in Sec. 5 and in [1, 6, 7]). Statecharts have been shown to be an excellent modeling and coding medium from design to final code, and SBP is particularly geared to describing the inter-object scenarios common in requirements specification, testing and the like. Our goal here is a unified visual and textual modeling language (and associated semantics) that includes the capabilities of both and any additional enhancements needed for wise computing.

Specifically, we plan to endow statecharts with the semantics unique to SBP, including *must* vs. *may* transitions, associating statechart states with events that should be considered for triggering and/or are forbidden in the state, and, system-wide enforcement of event order subject to all concurrent scenarios. This work has started and a draft specification of the integration is available in [11].

As an interim step we integrated a statecharts development-and-simulation tool with the PlayGo SBP tool, allowing not only exchange of events between the two systems, but also a common, integrated user-interface, and a well-defined semantics for joint execution [11].

Moreover, such a model immediately suggests a very useful approach to knowledge representation: describe it in scenarios that capture the application of the knowledge, similar to how the program itself is specified. This way, a particular unit of knowledge constantly tries to apply itself wherever it is relevant, perhaps in more than one way. An example of this approach is demonstrated in our preliminary result of a scenario-based biological model for the intra-cellular citric-acid cycle (a.k.a. Krebs cycle) [11]. Disparate chemical reactions are modeled as separate scenarios, oblivious of each other, comparable to how they might be written in a chemistry book. Yet, they all a work together, yielding a model that can

then influence other components, such as a model of a disease or a treatment thereof. Sec. 3 describes an example of scenario-based knowledge representation and aggregation, in a use-case for home-assistant robot.

This kind of knowledge representation can also help in detection of emergent properties or missing elements, complementing machine learning methods, with explicit, much simpler scenarios that look for behavior patterns or checklists of elements that must be present.

3 Enhanced Aggregation and Reflection

Two of the most attractive features of statecharts are their intuitive and powerful aggregation of states, and the ability to reference states of any object from anywhere. We propose to extend these two capabilities to other system entities, such as scenarios, and to also allow a single entity to be contained in multiple containers, which themselves are first-class citizens in the model.

Flexible hierarchy and reflection will help in creating contexts for scenarios, and then applying rules to entire contexts—e.g., day-time vs. night-time behaviors of a home-assistant robot, or separating the robot's self-maintenance activities from house duties. They will also simplify the application of external knowledge: e.g., the knowledge that a baby needs a certain number of hours of sleep, can, among others things, be reflected in a scenario that forbids the robot from doing *any* of the actions previously tagged or identified as noisy, when the baby is asleep. External reference to scenario states will also be valuable in refinement and repair of existing behavior.

Aggregation may also enable the DE to exhibit what may be perceived as creative behavior, or "thinking outside the box". As is often advised in innovation workshops, one should try to expand his or her "box", namely, by having a list of generic solutions whose applicability should be considered. Flexible aggregation enables the categorizing of certain system elements, as, e.g., "concurrency related", or "safety-critical", thus enabling automated examination of generic solutions such as a variety of locking and serialization mechanisms for the former, or added sensors-and-alerts or human supervision for the latter.

We have already prepared an initial specification for adding hierarchy and aggregation to SBP in the context of *live sequence charts* (LSC) [11]. An interesting observation is that, when SBP is finally fully integrated within statecharts, as proposed in Sec. 2, aggregation of inter-object scenarios and the ability to externally refer to scenario states will be readily implementable.

4 An Extensible Text-based Interface

In addition to any visual modeling and classical programming tools for the above, we propose to provide an extensible textual language with precise, unambiguous semantics for creating and discussing the models. A key feature of the language is that its statements are aligned with how human experts would prefer to converse, with people and with the DE, about the above systems—focusing on one fact at a time, in arbitrary order. Each of the statements can thus be viewed as a stand-alone entity, and their order or organization will not carry additional semantics.

A highly relevant preliminary result is our group's work on programming scenarios in controlled natural language with precise semantic rules—which may

thus be considered as a form of a domain-specific language (DSL). We have also created interfaces for displaying LSCs as English sentences. Our experience has shown that while a picture (say, of an LSC) is "worth a thousand words", a concise statement such as "when the driver presses the brake pedal, the brake light turns on" is both useful and convenient. Such an approach, based on stand-alone statements, is common also in declarative programming (e.g., Prolog), in text encoding of UML models (e.g., UML/P), or in assertions to SAT/SMT solvers (e.g., CVC4 and Z3). As for a DSL for statement-based description of statecharts, extending the above is straightforward, especially in the presence of existing text-based encoding of statecharts such as SCXML.

Thus, since all system entities, environment knowledge and meta information can be described in statecharts and scenarios, we are confident that we now have a good basis for a natural interface for the two-way text-based interaction of a wise DE and its users.

5 Self-Integrating Behavior

We introduce a design principle, termed *self-integrating behavior* (SIB), which we believe can dramatically accelerate the endowing of DEs with new knowledge and analytical abilities. We propose that it be adopted for specifications, actual system modeling and code, DEs, domain knowledge and other SE artifacts.

SIB-designed components "take care of themselves". They seek out their inputs, and make their outputs available to all other components. But, most importantly, they are also prepared to comply with constraints specified by other components. They can thus join or leave complex processes in an incremental, self-standing way with little or no change to existing components. The latter would be more difficult in common OO designs, where existing modules have to be modified with calls of relevant methods or for discovery of configuration changes[1].

One implementation we propose for SIB-design is that of *scenario-based programming* (SBP) [1, 6, 7], which we feel is particularly apt as a basis for it. It was first introduced with the visual language of *live sequence charts* (LSC), and later implemented in standard languages such as Java and C++. SBP calls for specifying system behavior as inter-object scenarios, aligned with how people often describe such behavior in, say, a requirements document or a user manual. Each scenario specifies some facet of system behavior with the events that may, must, or must not occur at each point. During execution, the scenarios run concurrently: the execution environment selects the next event to be triggered, by considering the entire specification according to well-defined composition semantics, and then advances all the scenarios accordingly. The capabilities and advantages of SBP have been demonstrated in a variety of sample application areas, including autonomous vehicles, industrial automation, a web server, and biological modeling.

As an example of SIB, consider a game-playing program where rules and playing strategies are each specified as a separate scenario, much as they would be when described verbally, or appear, say, in paragraphs of the printed instructions without having any program explicitly invoking all these modules in some order

[1] Aspect-oriented programming (AOP) provides limited, and somewhat fragile, SIB capabilities in standard programming languages, and indeed, some SBP implementations use AspectJ internally. For additional discussion of SBP and AOP see [7].

(see, e.g., our group's paper in EMSOFT'13 on compositionality in SBP). Another SIB example, focused on knowledge representation, is our preliminary-result modeling of the Krebs cycle, described in Sec. 2.

We propose to further investigate SIB. E.g., see whether any additional semantics are required for accomplishing it, beyond the native SBP principles. One such candidate is adding a context scope to the event-blocking idiom, limiting its otherwise global and unilateral effect. Another direction is to identify any special properties SIB may have, such as its communication or module-complexity costs, its possibly-improved succinctness under certain conditions, or its allowance of accelerated formal verification using, say, assume/guarantee techniques. A most intriguing research direction would be to measure the contribution of SIB and aggregation to tackling state explosion.

As mentioned above, wise-computing will incorporate, and appropriately extend, a broad range of SE ideas and tools. We intend to find ways to allow existing tools and automated techniques to be embedded in the framework of SIB, in order to allow DEs to harness their power smoothly and efficiently.

Finally, as in parts of the PoC in [5] we propose to integrate the DE and the system, using SBP and SIB, thus enabling a variety of intelligent ways to monitor systems and proactively guide their execution. This can be done both in development, say, in testing and simulation, and as part of the deployment and run-time environment, for, say, diagnosis and recovery from failures.

6 Conclusion

Since the early days of computing, the software engineering community has been aspiring for a development environment that can automate some of the more sophisticated tasks carried out by human experts. We presented four research directions: a single cross-function model for all aspects of development, flexible and extensible aggregation and reflection, a text-based interface based on stand-alone statements, and building models, systems, and development environments from units of self-integrating behavior. We believe that these can form important aspects of future system and development-environment design, and help integrate the most advanced tools and human knowledge and skills in SE and related disciplines, thus beginning to suggest the feasibility of the wise-computing dream.

Acknowledgement

This work is partly supported by grants from the Israel Science Foundation and from the German Israeli Foundation (GIF), by the Philip M. Klutznick Fund at the Weizmann Institute. We thank Gal Kaminka and Moshe Vardi for valuable comments on ideas expressed here.

References

1. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 2001.
2. H. Davis, B. Shrobe, and R. Katz. Towards a Programmer's Apprentice (Again). *Center for Brains, Minds and Machines*, 2015. CBMM Memo No. 030.
3. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

4. D. Harel, G. Katz, R. Marelly, and A. Marron. Wise Computing: Towards Endowing System Development with True Wisdom, 2015. Tech Report. http://arxiv.org/abs/1501.05924.
5. D. Harel, G. Katz, R. Marelly, and A. Marron. An Initial Wise Development Environment for Behavioral Models. In *MODELSWARD*, 2016.
6. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
7. D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Comm. of the ACM*, 2012.
8. C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit. Provably Correct Reactive Control from Natural Language. *Autonomous Robots*, 38(1):89–105, 2015.
9. H. Reubenstein and R. Waters. The Requirements Apprentice: Automated Assistance for Requirements Acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, 1991.
10. C. Rich and R. Waters. The Programmer's Apprentice: A Res. Overview. *IEEE Computer*, 1988.
11. Supplementary Material. http://www.b-prog.org/Models16PosterSupplemental.