Softw Syst Model
DOI 10.1007/s10270-012-0258-8

EXPERT'S VOICE

The quest for runware: on compositional, executable and intuitive models

David Harel · Assaf Marron

Received: 16 October 2011 / Revised: 30 May 2012 / Accepted: 9 July 2012

Abstract We believe that future models of complex software and systems will combine the crucial traits of intuitiveness, compositionality, and executability. The importance of each of these to modeling is already well recognized, but our vision suggests a far more powerful synergy between them. First, models will be aligned with cognitive processes used by humans to think about system behavior and will be understood, and perhaps creatable, by almost anyone. Second, one will be able to build models incrementally, adding to, refining or sculpting away already-specified behaviors without changing most existing parts of the model. Third, there will be powerful ways to execute such intuitive and compositional models, in whole or in part, at any stage of the development. The presence of these three traits in a single artifact will blur the boundaries between natural-language requirements, formal models, and actual software, bringing in its wake a major advance in the way systems are built, and in their cost and quality. We propose the term *runware*[1] to refer to this kind of higher level artifact.

[1] In the process of writing this article, we deliberated among a number of possible terms for this. Our desire was to find something, which, on the one hand, would be a natural third element in the *hardware* → *software* → *??* series of terms, but, on the other hand, would somehow provide a good connotation of the modularity–modality–executability triptych that forms the heart of our paper. Besides *runware* which we ultimately decided to use, these included *specware*, *flexware*, *smoothware*, *genware*, and *compware* (for *comp*ositional, *comp*rehensible, and *comp*uter-executable). If *runware* ends up not "sticking", then perhaps one of the others will…

Communicated by Prof. Jon Whittle and Gregor Engels.

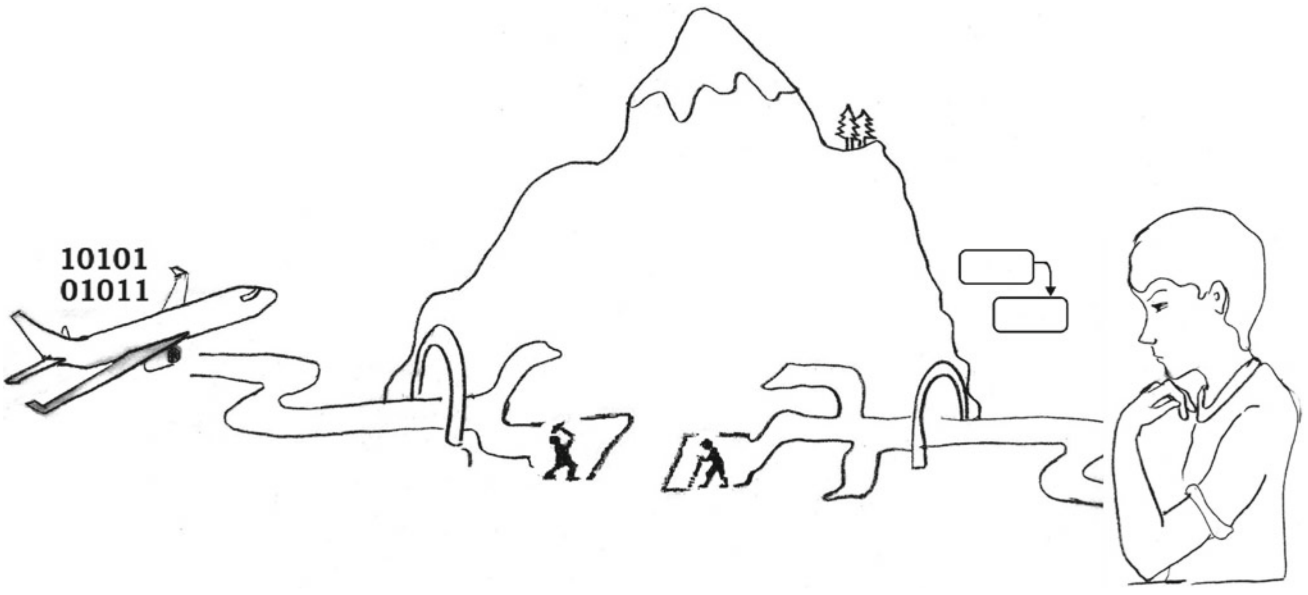
D. Harel · A. Marron (✉)
The Weizmann Institute of Science, Rehovot, Israel
e-mail: assaf.marron@weizmann.ac.il

Keywords Executable specifications · Model-driven engineering · Behavioral programming · Computational methods

1 Background

Since the inception of computerized systems, great efforts have been made to render programming, modeling, and specification languages as close as possible to how humans think about the problem at hand. These efforts have resulted in the invention and evolution of many languages, modeling paradigms, methodologies, and tools. Nevertheless, when engaged in programming, humans are required to think quite differently from the way they would have explained to others what they expect the computer to do. The transition from specifications written in natural language understandable by people all the way to computer-executable artifacts is heavily dependent on assistance from humans—a costly and error-prone process.

The efforts to bridge the gap between the way people form and express their thoughts about the system under development and the possible formats these thought can take that can eventually be executed by a computer, can be viewed as digging a tunnel through a mountain while working from both ends: in one direction, we seek ways to make human wishes and instructions more understandable by a computer, and in the other we try to advance the state-of-the-art in programming languages to better align them with human cognition. In this paper, we outline our vision for the future of complex system modeling, with this two-way tunnel-digging analogy in mind.



“The Quest for Runware”—We envision means for smooth system development, which would be discovered by tunneling through the barriers that separate human wishes and expectations from the executable artifacts that satisfy those expectations in full. These means would eventually enable developers to painlessly and intuitively transform their informal ideas into the final system

2 Abstraction and ambiguity need not imply vagueness

In system development, abstraction often implies “abstracting something away”—removing details to better see the large picture. While such abstraction is needed to deal with human constraints (e.g., limitations on our field of vision or short term memory), a collection of abstractions may be arbitrarily detailed and unequivocal. Additionally, in contrast to accepted ways of dealing with abstraction, a system model that is sufficiently detailed to be executable, but at the same time contains abstractions that enable human comprehension, does not have to be hierarchical in nature. It should be possible to create executable specifications from collections of abstract views, each of which is conveniently and intuitively created and manipulated by humans. The abstractions will, on their own, appear quite high level, and may be oblivious to many practical details or even to the existence of other abstract views.

For example, instructing an unmanned aircraft to fly from New York to Vienna via Paris provides an abstract view of a route that is at the very same time both specific and executable. Additional “pieces” of behavior, such as an unplanned stop in Dublin or on-the-fly changes in altitude to accommodate weather conditions, are not hierarchically related to the original route, and are driven by other views of the same trip. The same applies to behaviors guided by the need to manage fuel consumption or safety.

Another relevant facet of the way humans express ideas is ambiguity. While computers already deal quite often with ambiguous reality, we have been conditioned to believe that the instructions given to a computer must be pedantically unequivocal and complete, and that the consequences of ambiguous programs can be disastrous. In helping to dig the tunnel from the rigid, executable-by-computer side to the apparently more hazy, natural-for-humans side, we suggest that this premise may hamper progress and innovation, and that general techniques should, and can, be developed to counter it.

Efforts in this direction can be seen in areas such as constraint programming, fuzzy logic, machine learning, heuristic and probabilistic approaches to problem solving, and artificial intelligence in general. In the world of control systems it is common to use feedback loops to enable taking actions in a world where not everything is known. This feedback can be further enhanced by relying on the ability of software to recover from failures, and the ability to remotely repair and automatically download new software versions.

We thus believe that the mission of digging the system development tunnel connecting human cognition and computer-executability is not inherently impossible.

3 The vision

We believe that future models of complex software and systems will be far richer than today’s. They will be far easier to build and far more useful. In fact, they will feature a combination the crucial traits of intuitiveness, compositionality, and full executability. The importance of these traits to modeling is already well recognized, especially in the realm of reactive systems, which are the ones we concentrate on here

[20]. However, our vision sees them living together in full and powerful harmony in virtually all application areas.

No one knows exactly what models will actually look like in the future; will most of them still be mainly textual, or perhaps a lot more symbolic or graphical? Perhaps they will be captured by three-dimensional holographic artifacts or other as-of-yet unknown media… Still, we expect that both the model and the process by which it is constructed will be, first and foremost, intuitive. Modeling will be aligned with the cognitive processes naturally used by humans when thinking of, or describing, dynamic behavior. Second, we believe one will be able to build models incrementally, with little or no changes to existing components as new ones are added, thus “sculpting” the desired behavior to one’s desire. It appears that this is the way we humans learn: as new independent experiences are incurred, we enhance, refine, or replace what was previously learned without having to actually manipulate the memories that drove that learning. Mysteriously, new behaviors emerge from a collection of new and old experiences, instructions, rules, and goals. Third, there will be powerful ways to execute in full the resulting intuitive and compositional models—directly, without further human interpretation. Executability will be possible not only during the final system’s operation, but also in a variety of contexts during the entire process of system development and maintenance.

We propose the term *runware* to refer to the kinds of higher level artifacts that this vision would lead to. And in contrast to hardware and software, we believe that in the realm of runware the abilities of even very non-computer-savvy people relevant to the system under development will grow considerably. Most will be able to control the execution of runware models, possibly piecemeal, to ensure that the computer’s understanding indeed matches their own, and many will be able to actually construct and modify the models too.

4 Why bother?

What is the problem that would be solved by such runware, and why do we hope that scientists and engineers will make great efforts to resolve it?

Since the 1960s, when the term “software crisis” was coined, it has been well accepted that software development is difficult and expensive. Even with the recent overabundance of applications (e.g., for smartphones and tablets, and in web and cloud computing), it is clear that the demand for new applications and enhancements of existing ones far exceeds the supply. One need think only of the long lists of wishes pending for practically every corporate business application, or remember one’s own long-standing pet peeves regarding commonly used software packages.

To the quest for accelerated delivery and reduced costs we may add the natural creative and innovative drive of us humans. Our colleague Chen Keasar once claimed that the software crisis stifles innovation, and it indeed seems that the difficulty in building software (or computerized systems in general) becomes a conspicuous obstacle to technological and scientific progress. We believe that breakthroughs in software and system development will enable a growing circle of groups of people to fulfill their tech-related dreams on their own: from minor things like changing the way one’s home entertainment system is set up to record a TV program, via creating a popular web site or an order-tracking system for one’s business, and all the way to building a next-generation flight control system or integrating vast biological knowledge into a prediction-generating model.

Other important driving factors are the existence of software in almost all conceivable devices, and the broad realization of the near omnipotence of software; i.e., that any well-understood process that is doable within the limits of computability and computational complexity, can be carried out by an actual computer, usually driven by software.

Last, in a highly computerized society there will be a desire for customizability and personal innovation (say, in communications, navigation, office work, etc.). With fast-growing computer-oriented education, it will also be easier for humans to articulate what they would like their software to do as part of their real-world concerns, and then to either do the programming themselves or communicate the requirements for others to implement.

5 Elements of the solution

Casting the broad vision in a more tangible form, we will now describe some of its manifestations, as well as the technological elements that could help bring it about. Thus we try to turn a vision/dream into a set of predictions. And since these predictions are based to a large extent on existing research and available technology, we extend the discussion in this section with a brief review in the next, of where our field stands today with regard to these capabilities.

5.1 Intuitive executable specifications of behavior

Specifications and models will be similar to standard computer programs in that they will be fully executable. They will be different from most computer programs in that they will be a lot more intuitive. Thus, a record generated by humans reflecting their perception of how a certain environment behaves, and what they desire their system to do in that environment, will serve as part of the final application of the system. High-level compilation or interpretation will of course be required, but these will be automated, and in general will not require human guidance as to the semantics of what was intended by the model. Human intervention will be required to clarify one's wishes when the computer discovers irreconcilable conflicts or gaping holes in the original specification.

The specifications and models will be expressive and highly intuitive both to the computer professionals who create them (making their task easier) and to humans in general—enabling effective communication between programmers, architects, testers, end-users, customers, sales representatives, customer-service representatives, managers, etc. Note that customers and end-users include both lay ones and highly professional ones like technicians, operators, engineers, and scientists, among others. The expressiveness and intuitiveness of the specification/modeling languages will apply both to the description of the system's desired behavior as well as to that of its environment. Furthermore, the increasing proximity between the languages of people and machines will contribute to redefining the scopes and boundaries of traditional human roles in system development.

Intuitive specification idioms will continue the progression of programming languages—from machine language through assembly and high level imperative languages, to visual modeling, functional programming, logic programming, constraint programming, and domain-specific languages. Thus, even when coming very close to natural language, specification idioms will be associated with precise semantics, enabling repeatable machine execution within a given platform and across platforms, as well as consistent human comprehension of models and code. Note that precise semantics does not imply that the specification is deterministic—it only requires that all nondeterministic choices are well defined [21].

5.2 Small independent specification and modeling units

In contrast to standard programs, the intuitive yet executable specifications we have in mind will be comprised of relatively small units that will often be independent or semi-independent of each other. This mutual independence will be manifested in several dimensions.

- *Creation* During the development phases, one will be able to create units of specification to reflect the desired (or corrective) behavior without even looking at other parts of the specification.
- *Comprehension* During the planning, maintenance, or customization stages, one will be able to contemplate a unit of the specification and understand its purpose without having to see other units (despite the obvious fact that overall system behavior will continue to depend on all participating units).
- *Execution* At run time, a single specification or modeling unit, or any sub-collection of units, will be fully executable, yielding meaningful results.

These are in contrast with many conventional kinds of software components, which often cannot be created, understood, or executed without the explicit (and often cryptic) interfaces with the other components. In fact, this element of our vision stems directly from observing how people prefer to express requirements, as opposed to the final implemented parts of the system. Examples are abundant. For example, requirement FR-2.3 in a US Department of Agriculture system for managing data about water, soil, management, and economy for assessment of conservation practices [45], reads as follows:

> Browse, query, and download individual sampling station data and metadata. Provide access to the data via browsing of sites, stations, and instruments; allow for simple queries to individual datasets; provide a metadata search tool to query dataset parameters; and allow for downloading of datasets (full or partial).

Or, requirement UAV7 in the document for unmanned-aerial-vehicle compliance with air-traffic control [15]:

> S&A system (i.e., "sense and avoid") should notify the UAV pilot-in-command when another aircraft in flight is projected to pass within a specified minimum distance. Moreover, it should do so in sufficient time for the UAV pilot-in-command to manoeuvre the UAV to avoid the conflicting traffic by at least that distance or, exceptionally, for the onboard system to manoeuvre the UAV autonomously to miss the conflicting traffic.

These requirements are largely self-explanatory. Where they are not, they do not depend on other requirements and use-case scenarios, but on the common vocabulary, e.g., the definition of the terms "sampling station" or "the on-board system". Moreover, large parts of these two requirements could readily serve in the requirements document of other systems, processing different kinds of data. Indeed, if specifications become programs, there is no reason why common features of present-day software packages, such as reusability and portability, will not be observed in the kinds of intuitive and executable specifications discussed here.

Note that some of the individual sentences and clauses comprising each requirement paragraph could serve as stand-alone requirements, refining previously stated specifications. For example, the exception in the last sentence of the UAV requirement could be left absent only to emerge in a later paragraph, or an appendix, or in some software maintenance task description, with phrasing as follows:

> In addition to, and regardless of, other timing requirements and safety margins, the S&A system notification must be early enough to allow time for the onboard system to manoeuvre the UAV autonomously in order to avoid conflicting traffic.

5.3 Behavioral compositionality

As mentioned earlier, it will be possible to represent new requirements, or changes thereof, in a new behavioral component or module, with minimal change to the previously specified parts of the model, and without sacrificing executability and manageability. Such modules will be simply added to the existing model, virtually 'piled-atop' it, with no component-specific interface, connectivity, or ordering requirements.

In our vision, the units of the specification and models are not assembled in detail like resistors or chips on a computer board, or methods and fields in an OO-programming object class. The interweaving of behavioral modules will be facilitated by their reference to common aspects of system behavior described using shared vocabularies (for example, common events), and not via mutual awareness and direct communication between components. From the point of view of such a module, the other modules could be transparently replaced by new ones. In fact, the implementation of any part of the specification or a model should be replaceable by some kind of 'invisible box', whose implementation remains a mystery, and the effectiveness of the remaining modules and the integrity of the overall system behavior will be preserved. For example, consider requirement SMJ-02 in a US government requirement document for an acquisition and vendor management system [44], which reads as follows:

> Document record change audit trail - Generate an audit trail of document record changes (origination, update, deletion). Audit trail records must include: Document number, Change made, User ID/Name, Date/time stamp.

This requirement speaks purely of system behavior: of the change audit trail itself and of changes that are to be audited. It does not mention the many processes that could cause such changes, or the programmatic ways by which they may be effected. When adding new specification units, collective execution of the revised specification will smoothly incorporate the changes. The semantics for composing behavioral modules will be standard and application-independent.

To further illustrate behavioral compositionality, consider a present-day situation, in which a customer provides her detailed requirements to a programmer purely orally. That is, the former only speaks (writing nothing down), and the latter only listens and remembers everything that has to be done. Naturally, all refinements and changes coming from the customer as she remembers forgotten points or is questioned about emerging conflicts, will be appended as more spoken sentences. These sentences will refine or replace entire ideas, not words in a previously spoken sentence.

We believe that the ability to support this kind of process is very much in line with how humans store and process instructions and learning. There is a growing body of research and knowledge on the way human memory and learning are processed and stored, but this is outside of the scope of the paper. Still, we observe that this is how people communicate requirements and changes thereto, in requirement-specification documents, and in issue-tracking systems.

5.4 Multi-modality

The methods we advocate will give rise to the ability to specify executable behavior using a variety of modalities; enabling one to specify, e.g., what *must* happen, what *may* happen, as well as what *must not* happen. This is in contrast to most contemporary programming approaches, which are usually of a single modality (*do this*) often guarded by conditions, etc. Hinting at the implementation of our vision, specifying what may happen will provide the system with options and possibilities for things to execute, and specifying what must be done and what may not be done will constrain these options.

Of particular interest is the negative modality, which in lighter versions is sometime referred to as an exception: the ability to specify what is not allowed to happen (possibly under particular conditions). This alleviates the need to consider every part of the specification or model that may cause the undesired behavior. The constraints should be specifiable even when affected generators of the undesired behavior are not yet specified at all. Consider, for example, requirement SMB-05 in the acquisition management system, which states:

> Manual updates to Central Contractor Registration (CCR)— Prevent the agency from manually updating CCR.

In standard programming approaches, the introduction of such a requirement at a late stage may require changes to many existing modules. Aspect-oriented programming presently helps code such a requirement as a cross-cutting concern to be added to a set of existing base modules. In our vision, this negative requirement (like any other behavior) will be coded as a dedicated, independent module and be guaranteed to influence overall system behavior.

The use of generalization and exceptions in natural language interactions between humans offers convenient means of succinctness, and we believe that the same will emerge in system modeling and programming.

The kinds of programming idioms we have in mind will allow development by "sculpting" or "carving out" behavior, where some behavioral modules attempt (possibly nondeterministically) to generate new behaviors, while others eliminate undesired behaviors. Note that this approach also exists in nature, where the dynamics of biological and biochemical processes are given by excitation and generation processes but are very often also restricted by inhibition.

In addition to the standard impact on flow, such constraint specification will be usable broadly to avoid disasters, allowing the risk-taking often associated with innovation. In way of analogy, consider the learning potential and creative power of a human who is allowed to freely experiment with a variety of behaviors, except those that are forbidden (e.g., the illegal, expensive, or risky ones), figuring out if and when any of allowed actions produces valuable results.

5.5 Reliance on heavy-duty computational methods

Present scientific knowledge about brain function suggests that the processes by which humans interpret, store, and apply knowledge to behavior are very complex. In analogy, we believe that a computing infrastructure that enables intuitive, compositional, and executable representation of programs will also rely on complex algorithms and heavy-duty computation. These processes may or may not mimic, or be aligned with, the corresponding cognitive processes, but like the human brain, they too will be very complex. We envision them employing deep mathematical and computer science knowledge to manipulate the vast amounts of available information and specifications. Some underlying principles may be simple, but the explosive growth in the number of connections and composite states will require (at least initially) the most innovative approaches and the latest computer-processing power to support them.

Among the areas we believe will be represented in the approaches we foresee are heavy-duty verification and synthesis techniques, theorem proving, constraint solving, search techniques, logic, inference, optimization, machine-learning, planning, data mining, natural language processing, and more.

5.6 Adaptivity

Whether subsumed by "intuitiveness" or as a property in its own right, system adaptivity is an essential ingredient of intuitive executable specifications. As specifications aim to prepare systems for dealing with the world (as opposed to serving as an unconditional "cooking recipe"), a way must be found to deal with new conditions. No level of compositionality and incrementality in software development can make up for the vast range of conditions that systems encounter in the real, constantly changing world. Therefore, the system must be able to adapt by a variety of means, including, among others execution look-ahead and learning.

Many execution environments already include some look-ahead—for example, at the low level in processor optimization, and in applications like workload scheduling and autonomous vehicles. We believe that look-ahead can become standard in executing specifications at a far higher level and for a far broader range of reactive systems. Specifically, we predict that this will happen in resolving under-specification, where, when forced to choose among multiple paths, the execution environment will be able to optimize its choice, or at least choose a path that does not lead to disastrous results in the scope of the look-ahead. The actual depth, or horizon, of the look-ahead will vary based on technology and the application at hand.

Another mechanism that will enable intuitive (under-) specification will be the ability of a system to learn. Where look-ahead is not applied, or fails, the system will be able to learn and change its decisions based on its own experience. As in look-ahead, learning will influence the selection of the next step among multiple available options.

5.7 Awareness of good and bad states

Both look-ahead and learning functionalities require the support of *good versus bad* distinctions, as well as designation of finer qualitative and quantitative grades to system states. These designations can be offered by application-specific behaviors, or be inferred in an application-agnostic way from the execution conditions, as in the case of deadlocks or infinite loops.

5.8 Advanced development tools

It is only natural to expect that the actual process of modeling and programming will also benefit from major advances in runware. Rich interfaces will enable the expression of intuitive specification and modeling idioms textually, visually, by physical/tactile manipulation, or by other means. The development environments themselves will be specified in this manner too, allowing new requirements for assistive tasks to be readily developed. The development tools will be aligned with a deep behavioral understanding and decomposition of system development, with tasks ranging from the narrowest selection and sequencing of individual operations to broad activities, such as program refactoring, software package reuse, and the design of graphical interfaces and databases.

6 Where are we in the quest?

6.1 Initial work motivating the vision

The vision we describe here was ignited and fueled by the *scenario-based* programming approach [11,17]. We have recently expanded this work, and have started to use the term *behavioral programming* to refer to it [18,19]. The first-listed author, with his research group, has been involved in this paradigm since his initial work with Werner Damm on live sequence charts (LSCs) in 1998, and several additional groups have contributed to its evolution. Behavioral programming indeed focuses on the quest for intuitive, compositional, executable specifications, and besides LSCs it has been implemented in a number of conventional programming languages. For example, in a game-playing application built using behavioral programming, each rule and strategy may be considered as a different behavior. In an autonomous vehicle controller, each travel itinerary, maintenance schedule, obstacle avoidance rule, or physical stabilization technique may be coded as a separate behavior. Each unit of the specification controls system events (and sequencing thereof) as they must, may, or must not occur in a particular behavior of the system, while recognizing and reacting to the environment events that are beyond the system's control. At runtime, these independently specified behavioral specification units are interwoven into a combined execution of the system, subject to well-defined semantics. Thus a collection of behavior specifications constitutes a cohesive, executable system.

6.2 High-level languages

The quest for intuitive executable specifications started with the development of assembly languages to replace coding in machine languages. Work on higher level languages can be claimed to have started with Backus' work on FORTRAN [1]. The development of programming languages ever since has provided additional major steps in the direction of digging the tunnel from the computer side, continuously providing programmers with ever-higher abstractions to work with. A general survey of the history and current state of programming languages is far beyond the scope of this article; the reader may refer, e.g., to [28].

6.3 Software composition

The composition of independent units of specification has been addressed in several areas, including aspect orientation [24], rule-based systems (see, e.g., [12]), behavior-based robotics (e.g., [9]), feature-oriented development [37], software product lines [2], and correct-by-construction compositional methods [5]. Software compositionality is also prominent in architectures for concurrent and distributed computing (see, e.g., [13,36]), in decentralized control, in agent- and actor-oriented architectures (e.g., [7]), and in feature distribution [22]. Such conceptual distribution and decentralized control can be implemented in a single system, in specifically designed systems, or in generalized ones, say, web-based platforms. A key criterion for evaluating such compositional approaches is the level of inter-dependency that is mandated by the composition itself. The same applies to assessing relevant language constructs, such as methods in object oriented programming (see, e.g., [32]), or monads and monad transformer stacks in functional programming (e.g., [46]). Thus, we are interested in the extent to which a developer, aware of the need to reduce coupling or cohesion among modules, can minimize the changes to existing code (or model) when adding, removing or refining behavior.

6.4 Specifying modalities

Certain kinds of behavioral modalities can be found, e.g., in logic programming (see, e.g., [43]) and in constraint programming (e.g., [40]), where the system searches through a non-deterministic solution space, subject to declared specification of goals and limitations. Application-specific handling of modalities can be found in AI applications too. There, the program has to decide and choose its next action from among multiple available options, subject to explicit constraints and optimizations, based on the current state or on planning and look-ahead. Design by contract and the assertions used in the technique [32] provide means for programmers to specify preconditions, post-conditions, and invariants that are specified as being required or not allowed, and which are monitored at development time and at run-time, creating exceptions when actual behavior is not as required.

6.5 Languages for modeling and requirements

Digging into the mountain from the other side, requirements engineering (see, e.g., [33,47]) focuses on what requirements and models should contain, and how domain-specific knowledge and "world-behavior" should be described. Modeling languages, most prominently the grand collection of modeling notations—the UML and its sub-languages and various types of diagrams [35]—offer formalization for abstraction of the *what* and the *how* of a system and its environment. Model-driven architecture and model-driven engineering provide methodologies, guidelines, and tools for building working components and systems from formal abstract models with little or no lower level programming (see, e.g., [30,34,41]). Many tools and languages were proposed over the years to help end-users and domain experts develop application components, such as customized reports, database queries, process-automation scripts, business transactions, or rule-based systems, forming the category of domain-specific languages [31]. In addition, a variety of languages evolved to enable children to program with minimal programming education (see, e.g., [38]). An ambitious vision for transforming intentions into working software is described in [42].

6.6 Adaptivity

Software adaptivity is of great interest in designing systems with increased autonomy. It alleviates some of the burden that lies on the programmer's shoulders during software design, of predicting possible situations in the program's future runs. A variety of approaches have been proposed for this (see, e.g., [8,29]). Adaptivity goes hand-in-hand with advancements in areas of artificial intelligence, such as planning and reasoning algorithms (see, e.g., [27]) and machine-learning (e.g., [4]), as well as in genetic algorithms (e.g., [26]).

6.7 Verification and validation

Significant advances have been made in many directions of work on verification and validation, that is, on the profound issue of determining whether a system meets its specification. An entire paper would be needed to review this area of research, even superficially. In the present context, it suffices to mention two main thrusts, model-checking (see, e.g., [10]) and synthesis (e.g., [6]), both of which are directly relevant to many facets of the quest described here.

6.8 Human interfaces

The use of advanced human interfaces in the system development process itself is gaining prominence. We feel that visual languages can play a central role in making this far more useful (see, e.g., [30]). Programming and system modeling can benefit from advances in speech recognition and language processing, both directly and indirectly (e.g., [3]), and from technologies in machine vision (e.g., [16]), and in tactile and motion interfaces (e.g., [25]). In addition, there are specific development environment features that can help make modeling more natural and intuitive. They include powerful context-aware auto-completion and refactoring of methods and field names, which attempt to guess what the programmer was about to do or the objects that are relevant to the task at hand (e.g., [14]).

6.9 Processing power

Needless to say, the immense processing power needed for such intelligent operations is becoming increasingly available with more powerful computers, massively parallel systems and clusters, pervasive computing power in every device, and readily-available access to web-based cloud computing.

6.10 Formalized logic

Besides our own motivating work, mentioned briefly at the beginning of this section, which we consider as a very partial feasibility test, but with positive results, two phenomena from outside of computer science inspire us and increase our confidence that the quest described here is indeed feasible. First, some modest parts of the extraordinarily powerful, yet mostly mysterious, human faculty of reasoning have been formalized over the years by philosophers and logicians, and many parts thereof were eventually algorithmicized and made executable by computer (see, e.g., [39]).We find it particularly impressive that logic can be manifested both in the processing of the smallest units of information, e.g., electronic gates processing 0s and 1s, and in the solutions of complex reasoning problems. We predict that system behavior and its development will reap very similar benefits.

6.11 The brain

The other phenomenon is the successful manifestation of these ideas in the human brain. As mysterious as the brain still is, the problem of intuitive but compositional executable models seems to be largely solved by the elaborate machinery inside our skulls. A continuous flow of sensory stimuli is internalized and stored in a manner that drives our future choices and behaviors. In normal, healthy people there is apparently no modification of existing initial memories—no insertions, no cut-and-paste—only more and more experiences. Images seen, sentences heard, pain felt, are all amassed as new memories and connected to existing memories in more ways than we can imagine today. Some of these, of course, explain, refine, correct, reorganize, or completely replace things that were previously experienced (or seen or heard or read) in how they affect future behavior (see, e.g., [23]). However, we know very little about how this is done in the brain and cannot predict whether the computational capabilities required for such feats will be available in the foreseeable future. Still, the magic of the brain provides us with a goal and inspiration in the search for a practical man-made solution.

7 Implications and conclusion

We felt that it would be nice to find a fitting term for powerful kinds of intuitive and composite models envisioned here, which despite their high-levelness are still fully executable. We found the term *runware* most appropriate for this (although we considered several others; see the footnote at the beginning of the article), since it somehow alludes to models being computer-executable/runnable and also serves as a natural continuation of the almost tactile connotations present in the terms *software* and *hardware*.

Terms and connotations aside, the real question revolves around the implications of having such runware at our disposal. What will happen if and when the human way of expressing requirements for systems will become almost indistinguishable from the way this is done with computer programs? And what will be the result of a level of compositionality that would allow humans to add capabilities to a system with much less dependence on that system than is possible today?

First, of course, software development will become cheaper, faster, and more pervasive, accelerating the progress of technology. Second, this may transform the task of maintaining legacy applications so that it will not be substantially different from initial development. Maintenance may actually change from being a task often regarded as tedious and undesired by professionals, into a rewarding assignment, where speedy identification and resolution of real pain-points of end users, or of failures in production applications, will provide satisfaction and recognition.

System customization and personalization will become a new major phase in the life cycle of systems. End-users will be able to add specification units to control desired system behavior. These may be coded by the end users themselves or by the programmers supporting them, and perhaps even downloaded from large pools of vendor-provided and community-developed behaviors for similar kinds of systems.

Another software engineering task that may be drastically transformed when our vision is realized is the need to update models and documentation based on changes made to production systems. As the specification of each behavior and change thereto, in its most basic form, will become part of the final system, this task may be all but eliminated.

When a collection of specification units grows over time, accumulating an unmanageable collection of patches, exceptions, and enhancements, it is likely developers will call for merging or refactoring them into much more concise artifacts. The new modules will replace the existing collection in all its uses (final executing system, official record of the specification, and means for communication between humans) without changing other parts of the specification. Such refactoring will be acceptable, even if it ends up being done manually, as it will focus on capturing the human's revised perception of the affected behavior.

Still, the most significant implication of advanced runware of the kind envisioned here may appear in totally unexpected places. As more and more areas of everyday life benefit from intelligent systems, perhaps a significant acceleration of system development may lead to a wave of innovation in a variety of fields—from physics and biology to economics and communications.

The goals of connecting and unifying cognitive models of systems with their executing code will be challenging, to say the least. We expect that the "tunnel digging" will proceed in parallel, by many teams of many disciplines, and in many different ways. We can only hope that these efforts will be successful and will yield a variety of approaches for all to choose from.

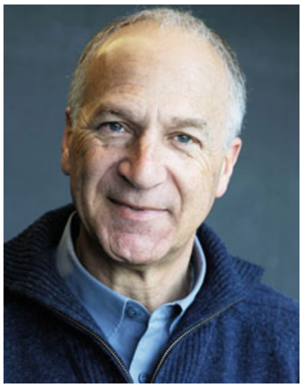
Acknowledgments We thank the anonymous reviewers for their constructive and valuable comments. This research was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science and was partly funded by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

References

1. Backus, J.W., Beeber, R.J., Best, S., Goldberg, R., Haibt, L.M., Herrick, H.L., Nelson, R.A., Sayre, D., Sheridan, P.B, Stern, H., Ziller, I., Hughes, R.A., Nutt, R.: The FORTRAN automatic coding system. In: Papers presented at the February 26–28, 1957, western joint computer conference: Techniques for reliability, pp. 188–198. ACM (1957)
2. Batory, D.: Product-line architectures. In: Smalltalk and Java Conference (1998)
3. Begel, A., Graham, S.L.: An assessment of a speech-based programming environment. In: IEEE Symposium on Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006, pp. 116–120. IEEE (2006)
4. Bishop, C.M.: Pattern Recognition and Machine Learning, vol. 4. Springer, New York (2006)
5. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: CONCUR (2008)
6. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3) 2011
7. Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F.: Multi-Agent Programming: Languages, Tools and Applications. Springer, Berlin (2009)
8. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, pp. 28–33. ACM (2004)
9. Brooks, R.: A Robust Layered Control System for a Mobile Robot. IEEE J. Robot. Automat. 2(1) 1986
10. Clarke, E.M. Jr., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
11. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. J. Form. Methods Syst. Des. **19**(1), 45–80 (2001)
12. Date, C.J.: What Not How: The Business Rules Approach to Application Development. Addison-Wesley Professional, Reading (2000)
13. Davis, II J., Goel, M., Hylands, C., Kienhuis, B., Lee, E.A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J. et al.: Overview of the Ptolemy project. Department EECS, University of California, Berkeley, CA, ERL Technical Report UCB/ERL (M99/37) (1999)
14. Eclipse Foundation. Eclipse inetgrated development environment documentation. http://help.eclipse.org/indigo/. Accessed May 2012
15. European Organisation for the Safety of Air Navigation. EUROCONTROL Specifications For The Use Of Military Unmanned Aerial Vehicles As Operational Air Traffic Outside Segregated Airspace. EUROCONTROL (2007)
16. Forsyth, D.A., Ponce, J.: Computer vision: a modern approach. Prentice Hall Professional Technical Reference, Englewood Cliffs (2002)
17. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Berlin (2003)
18. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM **55**(7), 90–100 (2012)
19. Harel, D., Marron, A., Weiss, G.: Programming coordinated scenarios in java. In: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP), pp. 250–274 (2010)
20. Harel, D., Pnueli, A.: On the Development of Reactive Systems. NATO ASI Series, vol. F-13. Springer-Verlag, New York (1985)

21. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of semantics? Computer **37**(10), 64–72 (2004)
22. Jackson, M., Zave, P.: Distributed feature composition: a virtual architecture for telecommunications services. IEEE Trans. Softw. Eng. **24**(10), 831–847 (1998)
23. Kandel, E.R., Schwartz, J.H., Jessell, T.M. et al.: Principles of Neural Science, Vol. 4. McGraw-Hill, New York (2000)
24. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming (ECOOP) (1997)
25. Kortum, P.: HCI beyond the GUI: design for haptic, speech, olfactory and other nontraditional interfaces. Elsevier/Morgan Kaufmann, Amsterdam (2008)
26. Koza, J., Poli, R.: Genetic programming. Search Methodol, pp. 127–164 (2005)
27. LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006)
28. Louden, K.C., Lambert, K.A.: Programming languages: principles and practices. Course Technology (2011)
29. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. Computer **37**(7), 56–64 (2004)
30. Mellor, S.J., Balcer, M., Foreword By-Jacoboson, I.: Executable UML: a foundation for model-driven architectures. Addison-Wesley Longman Publishing Co. Inc, Boston (2002)
31. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. (CSUR) **37**(4), 316–344 (2005)
32. Meyer, B.: Object-Oriented Software Construction. Prentice hall, New York (2000)
33. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering, pp. 35–46. ACM (2000)
34. Object Management Group. OMG Model-Driven Architecture. http://www.omg.org/mda/. Accessed May 2012
35. Object Management Group. OMG Unified Modeling Language Superstructure version 2.4. Accessed May 2012
36. OSCI. Open SystemC Initiative. IEEE 1666 Language Reference Manual. http://www.systemc.org. Accessed Sept. 2011
37. Prehofer, C.: Feature-oriented programming: a fresh look at objects. In: ECOOP (1997)
38. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. et al.: Scratch: programming for all. Commun. ACM 52(11):60–67 (2009)
39. Robinson, A., Voronkov, A.: Handbook of Automated Reasoning, vol. 2. Elsevier, Amsterdam (2001)
40. Saraswat, V.A., Rinard, M., Panangaden, P.: The semantic foundations of concurrent constraint programming. In: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 333–352. ACM (1991)
41. Schmidt, D.C.: Guest editor's introduction: model-driven engineering. Computer **39**(2), 25–31 (2006)
42. Simonyi, C.: The death of computer languages, the birth of intentional programming. In: NATO Science Committee Conference (1995)
43. Sterling, L., Shapiro, E., Eytan, M.: The Art of Prolog, vol. 94. Wiley Online Library, London (1986)
44. US General Services Administration. Acquisition System Requirements Draft. http://www.acquisition.gov/fas_reqdraft042808.pdf. Accessed May 2012
45. USDA-Agricultural Research Service. CEAP/STEWARDS System Requirements Specification. USDA (2006)
46. Wadler, P.: Monads for Functional Programming. Adv. Funct. Program. LNCS, vol. 925, pp. 24–52 (1995)
47. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. (TOSEM) **6**(1), 1–30 (1997)

Author Biographies



David Harel is the William Sussman Professorial Chair in the Department of Computer Science and Applied Mathematics at the Weizmann Institute of Science, Rehovot, Israel, and has served as Dean of the Faculty of Math & CS. He has worked in logic and computability, software and systems engineering, modeling biological systems and more. He invented Statecharts and co-invented Live Sequence Charts. He was co-founder of I-Logix. Among his books are "*Algorithmics: The Spirit of Computing*" and "*Computers Ltd.: What They Really Can't Do*". His awards include the ACM Karlstrom Outstanding Educator Award, the Israel Prize, the ACM Software System Award, and four honorary degrees. He is a Fellow of ACM, IEEE and AAAS, and a member of the Academia Europaea and the Israel Academy of Sciences.

Assaf Marron is a researcher at the Weizmann Institute of Science, Department of Computer Science and Applied Mathematics. He is a member of the research group of Prof. David Harel. His research interests include behavioral programming and scenario-based programming, machine learning and information visualization. Assaf holds a PhD in computer science from the University of Houston. Assaf worked for many years in research and development of innovative products and technologies at leading companies including IBM and BMC Software. He is the inventor or co-inventor of several patents.