Integrating Interobject Scenarios with Intraobject Statecharts for Developing Reactive Systems

David Harel, Rami Marelly, Assaf Marron, and Smadar Szekely
Weizmann Institute of Science, Rehovot, Israel

Editor's notes:
An important role of cross-layer design is to reconcile model-implementation differences, often stemming from how the two layers are specified. This article shows how a single method and tool can support both the specification and implementation stages, resulting in better closing the "model-implementation gap."
—Samarjit Chakraborty, University of North Carolina at Chapel Hill

When developing software in any discipline, using the traditional waterfall process or any variant of agile and spiral development, all stakeholders are faced with the existence of multiple conceptual layers: requirements, design, and final running code. Throughout the development process, domain experts, system engineers, programmers, and other stakeholders constantly interact to make sure that the transitions across the boundaries of such conceptual layers are indeed correct, and offer acceptable mappings, usually unidirectional refinements. Development tools assist in the process, by introducing artifacts that can be understood and discussed by people of different professional backgrounds, and which can be tested and validated, manually or automatically, against artifacts from other layers.

More specifically, functional requirements describe system behavior and traits, from the point of view of the various stakeholders. They often consist of scenario-based descriptions of sequences of events that reflect desired, allowed, and forbidden behavior. A central characteristic of such scenario-based specifications is their interobject nature. A scenario can contain a flow of events involving any number of objects, internal or external, including subsystems and human users, for example, in the Windows operating system "when the user presses ctrl and then alt and then del, and does not release the pressed buttons, then the task manager screen is displayed." Each scenario can list out for many events, triggering any number of actions, and subjecting all operations to a variety of conditions. Each requirement scenario is self-standing, and with sufficient context can appear anywhere in a requirements document. The scenarios are composed at runtime: the execution environment runs all scenarios in parallel in a synchronous manner, reevaluating all constraints and conditions with every system step and every occurrence of external event. The composition and dependences are well understood in the reader's mind because of the intuitiveness of the compositional idiom.

In addition to natural-language descriptions in requirement documents, such scenarios are often expressed in rigorous languages. A good example

Digital Object Identifier 10.1109/MDAT.2020.3006805
Date of publication: 3 July 2020; date of current version: 28 September 2021.

is the visual language of live sequence charts (LSCs) [1], [2], which evolved from message sequence charts (MSCs). The LSC concepts were adopted in the latter formalization of UML sequence diagrams and in a variety of tools and methodologies. Detailed semantic definitions have made it possible to simulate and execute these scenario-based specifications via runtime concurrent consideration of all scenario constraints and preferences (a process termed play-out). This gave rise to the interobject paradigm of scenario-based programming (SBP), also termed behavioral programming, originally supported by the Play-Engine [2] and later by PlayGo [3]. SBP was later extended to standard programming languages like Java, C++, JavaScript, and Erlang (see [4]), and to domain-specific textual modeling languages like ScenarioTools's SML [5].

Although interobject scenarios are an excellent way to specify and compose requirements, in the common approaches to system design and implementation, system behavior is constructed from intraobject specifications. These object-oriented (or object-centric) descriptions provide for each object separately its behavior as manifested in direct interaction with the environment and with other objects, through events, message exchanges, and internal operations. There are numerous nonvisual procedural languages for object-oriented programming, such as Java and C++, but for a formal visual description of reactive behavior, it is common to use state machines, where each describes all the states of a given object and its reactions, in each state, to all possible external and internal stimuli.

In 1987, the statecharts language [6] was introduced, as a visual formalism that augments conventional state machines with notation and semantic definitions for the concurrency and hierarchy necessary to specify and then directly execute complex behavior. An object-oriented version thereof was described in [7], and among other things this variant became the basis for the state-based language of the UML. Statecharts have been implemented in multiple software engineering tools, such as STATEMATE and Rhapsody (acquired by IBM), MATLAB's Stateflow, SCADE, LabView, Yakindu [8], and others, and have become the visual formalism of choice for intraobject behavior specification in a multitude of industries.

The conceptual duality between the interobject and intraobject approaches is illustrated in Figure 1, originally appearing in [2].

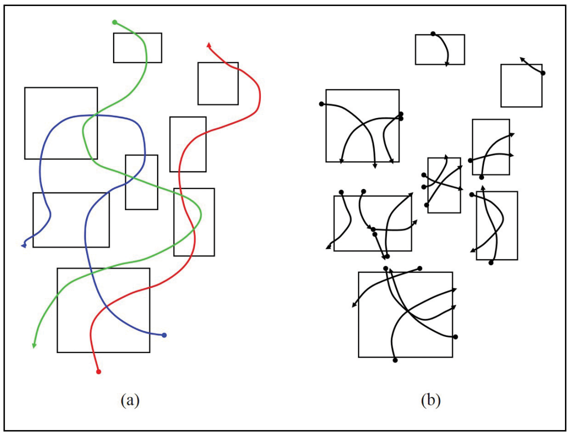

Figure 1. (a) Interobject scenarios cross multiple object boundaries in describing "full stories." (b) Intraobject specifications describe the "full reactivity" of each object. Reproduced from [2], with permission.

The "full story" of the sequence of events in each scenario is provided explicitly in the interobject scenarios, while it is only implicit in the intraobject specification of all objects involved. Conversely, the full reactive behavior of any given object is visible explicitly in the intraobject specification, but in scenario-based specifications it must be derived from multiple scenarios.

Although the intraobject specification approach is directly aligned with classical object-oriented programming, the translation from an interobject, scenario-based specification to implementation is a central issue in software engineering, and constitutes a substantial part of many software development efforts.

In the past, scenario-based behavior specifications were used mainly to help guide the development, and then the testing, of the implementation—given in a conventional intraobject fashion. Testing is often done with the aid of a tool that monitors the execution of the intraobject implementation and confirms that the specified interobject scenarios indeed comply with those specific runs. A key contribution of SBP is the fact that the LSC language and its derivatives have powerful enough syntax and semantics as to render the requirement specifications directly executable. In other words, SBP enables building a working system (or a highly functional simulator thereof) from modules that are aligned with how humans often describe behavior. What happens during the running, or playing out, of the specification is that an SBP execution engine follows all the scenarios in parallel, waits for environment and system-driven events and reacts to them by triggering other events according to the specified behavior, while, significantly, avoiding or delaying the triggering of events that are forbidden (in the current state) by some scenario.

This allows for direct execution and dynamic testing of requirements in early prototypes and simulations, and/or for programming a system using its multimodal requirement scenarios (see [9] paper 174). This can save the developers and engineers part of the efforts associated with transforming requirements into design. Solutions for key design considerations (or concerns, etc.), such as detecting conflicts between independently specified scenarios, or efficient parallel execution of thousands of scenarios, are emerging from research on SBP (see [4] and in [9] papers 230, 233, 257).

SBP supports agile, or spiral, development methodologies, in that when new requirements or refinements are introduced, one can often specify them incrementally in new stand-alone scenario with little or no change to existing ones (see [9] paper 230). The naturalness of programming with scenarios has been further discussed in several studies and in observing how children learn to program.

However, SBP has its limitations. While early in development external system behavior is usually conveniently described using scenarios, there are many inner mechanisms and details that are less amenable to such specification and require an object-oriented method. Together with constraints of pre-existing OO software components, and ingrained programming tradition, this often causes developers to make the entire design intraobject.

In this article, we present an overall development philosophy, which supports a natural integration of interobject and intraobject approaches. It offers a gradual and coherent transition from the former to the latter, allowing the coexistence and coexecution of “completed” intraobject statecharts with interobject scenarios that have not yet been captured in statecharts, or which have been purposely retained; for example, for verification and validation (see Figure 2).

Specifically, we have extended the PlayGo tool for LSCs and have integrated it with the Yakindu Statechart tool (available from itemis corporation; at the time of writing this article, the license for noncommercial use is free) [8]. The integrated tool supports beginning with an executable model of the requirements and incrementally adding implementation details by object-oriented statecharts, and then optionally removing already-implemented requirements. Thus, the proposed approach and tool support the smooth back-and-forth transition across boundaries of the conceptual layers of requirements elicitation, formal specification, design, and implementation.

Introducing the Statecharts and LSC languages

Statecharts

Three key concepts that the statecharts formalism added to classical state machines are: 1) concurrency, i.e., separate state components that are active simultaneously and can carry out transitions

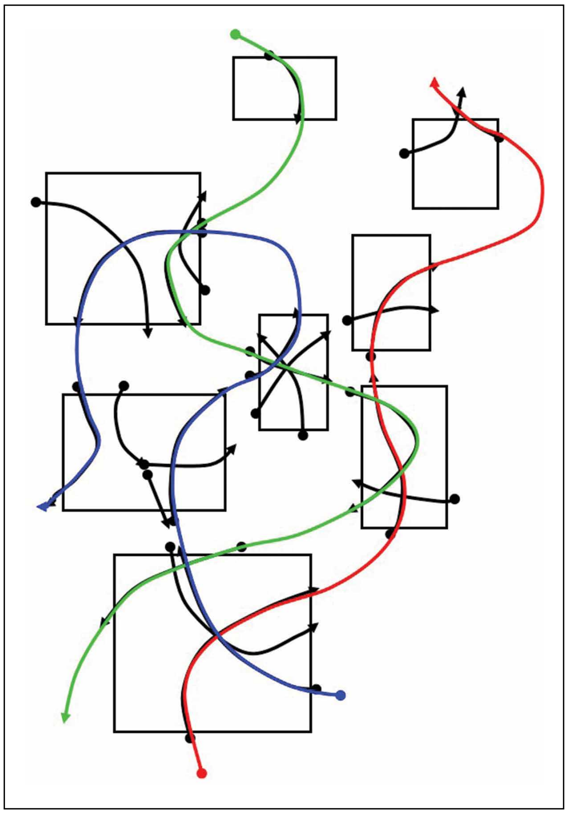

Figure 2. Modest illustration of our vision: the interobject and intraobject views of system behavior are cohesively integrated, both superimposed upon each other and complementing each other.

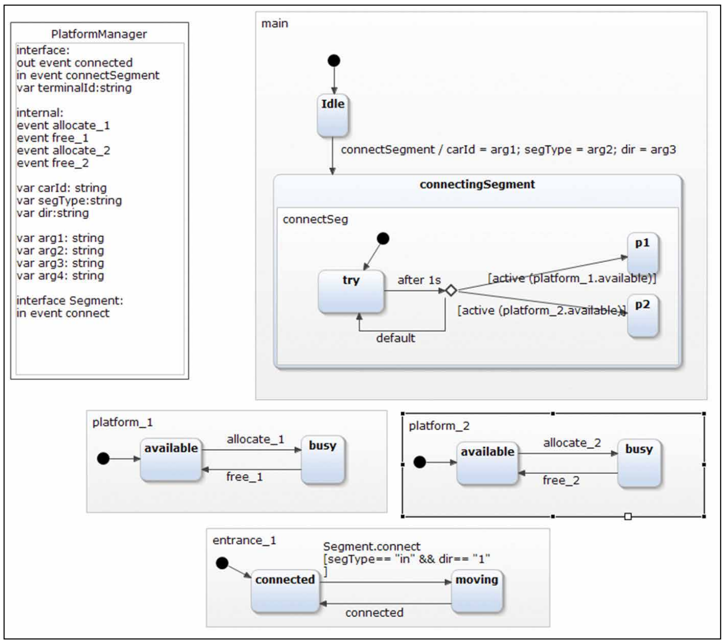

Figure 3. Statechart of the platform-manager object of the railcar example, showing parts of its behavior in four concurrent states. The two station platforms, Platform 1 and Platform 2, can be allocated (or freed) for an incoming (resp., departing) railcar. The Entrance 1 state represents the current status of the rail segment that connects Entrance 1 with the platforms. The superstate main moves from Idle to connectingSegment upon the triggering of the connectSegment event, in which case its three arguments, arg1, arg2, and arg3, are stored in the internal variables carID, segType, and dir, respectively. When entering the connectingSegment superstate, the platform manager tries to allocate a platform to the incoming railcar, by checking which platform is available, in intervals of 1 second, until successful. This is done using the choice construct, and the active function checks if another region is in a certain (sub-) state. (All statechart images are from the Yakindu statechart tool, and are produced here with the permission of itemis Corp.)

at the same time as others; 2) hierarchy, i.e., the ability to specify that one state contains multiple other states and associated transitions, with an unbounded containment depth; and 3) the ability to condition a local behavior on the fact that another region is in a particular (sub-)state. States are drawn as rountangles (rectangles with rounded corners). Hierarchical containment is depicted by the physical containment of states within another state. Concurrent state components, also termed orthogonal states, are drawn either as a partition of states into regions, using dashed lines, or depending on the supporting tool, as free-floating states on the top level of the hierarchy. See example in Figure 3 (taken from the railcar application described in the “Integration semantics and implementation” section).

Statechart transition arrows can be (optionally) labeled with: 1) events that trigger the transition; 2) a guard condition (in square brackets) that must be true to enable it; and/or 3) actions that are to be carried out when the transition is taken (specified following a “/”). Additional actions can be specified to be taken upon entering or exiting a state, or while in a state.

The statecharts language contains additional features (see also Figure 3), including specifying the raising or triggering of events; richer specification of conditions and time, the ability to re-enter a super-state directly to the inner state in which it was when the super-state was previously exited, dealing with synchrony and parallelism/simultaneity (like the ordering of events that become enabled “at the same time”), reference to other objects and states within the statecharts of those objects, and more.

In the Yakindu statechart tool, which we use in our implementation, every statechart specification contains also a list of interfaces representing the class to which this statechart belongs, and objects or classes with which the statechart can communicate and the related events and variables.

A key contribution of our integration in this article is that the object model used by LSC is the very same one that is used by the statecharts infrastructure.

Live sequence charts

Figure 4 shows several LSCs [the acronym LSC is both the language name and a noun for a single scenario (plural: LSCs)]. Each scenario describes one aspect of system behavior—typically its response to an event or a sequence of events under certain conditions. The events are messages (depicted as horizontal arrows) exchanged between (vertical) lifelines. Each lifeline is associated (labeled) with an object (symbolically by class, or concretely using a particular instance thereof). In a given lifeline, events are ordered, with time flowing from top to bottom. The order among events that appear on different lifelines is partial and can be constrained by other language constructs that synchronize those lifelines.

The LSC language distinguishes events that are executed, i.e., triggered by the runtime infrastructure when enabled (marked by solid lines), from events that are merely monitored, i.e., waited for in the particular scenario (marked by dashed lines). The language also distinguishes events, which, once enabled, must eventually occur (colored red), from events that only may occur (colored blue).

The LSC language supports additional constructs, such as conditions, including ones that can cause interrupts in scenarios, variable assignment, flow control (e.g., loops), nested subcharts, and more.

The PlayGo development environment for LSC provides a rich GUI for class/object model specification and scenario specification, which can be done both by drawing and by using natural language (English text). It supports full execution (play-out), including simulation and debugging, and play-in (translating GUI-based user-controlled event triggering into scenarios).

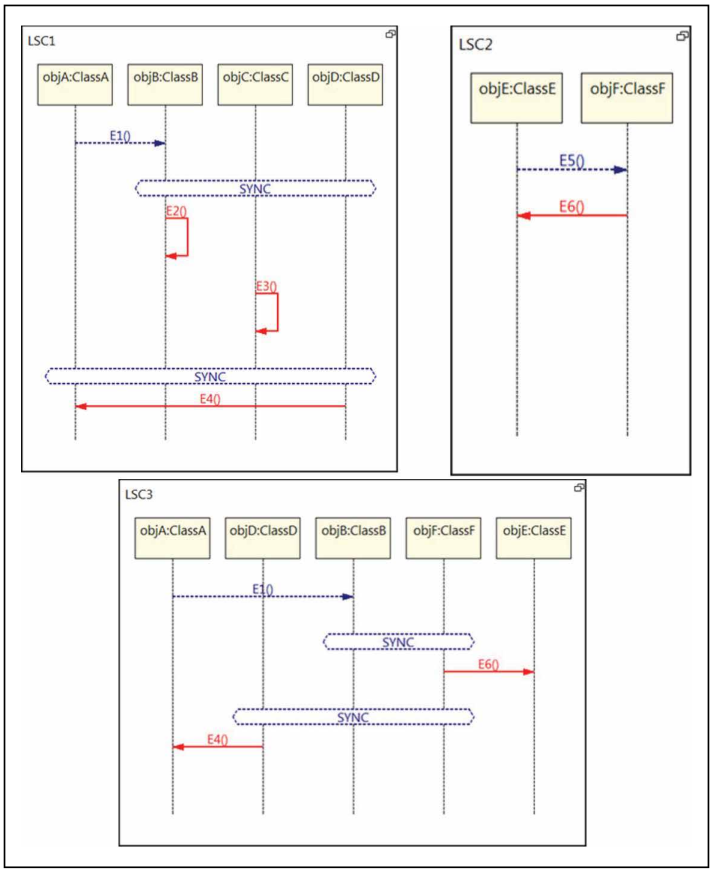

Figure 4. LSCs example: Scenario LSC1 specifies that after event E1 occurs, events E2 and E3 must occur, in any order, and, after both of them occur (enforced by the SYNC construct), E4 must occur. LSC2 specifies that after E5 occurs E6 must occur, and LSC3 specifies that once E1 occurs, E4 cannot occur until E6 occurs. Hence, when executing these LSCs, after E1 is triggered E4 will be delayed until E5 is triggered (by the environment or by some other scenario), subsequently triggering E6 and enabling E4.

Integrating LSCs and Statecharts: A simple example

In this section, we motivate our integration of scenarios and statecharts, and illustrate it using an extremely simple example. We focus on its value in terms of the development process that integrates the two models, and do not deal with problem-specific nuances. A more elaborate example is given later, in the "Revisiting the railcar system" section.

The front end of our example system consists of a simple GUI of a switch and a light shown in Figure 5a.

The only requirement is that whenever the switch is set to on the light turns on, and when it is set to off, the light turns off. This is coded as a single scenario, shown in Figure 5b.

Note the use of the same variable name, state, and values on and off for the switch and the light, to create the intuitive scenario logic. In this phase, the user does not care or know how the system implements the requirement; for example, how the information about the switch's state is transferred to the light.

This LSC is executable, and using PlayGo the user can already test the specification by turning the switch on and off via the GUI and checking the light's reaction. In the next step, the developer starts to incorporate design considerations, by introducing a controller. The controller receives a toggle message from the switch and sends a toggle message to the light. This LSC (shown in Figure 5c) is executable as well, and while running the two the developer can track the sequence of events between the switch, the controller and the light.

Now that the design is completed, the developers can proceed to the implementation phase. The first thing they might like to do is to implement the switch's logic, which can be done by the statechart of Figure 6a. This moves the responsibility for the switch's behavior from the LSCs to the statechart, and the corresponding message in the LSC is changed from executed to monitored (shown in Figure 6b).

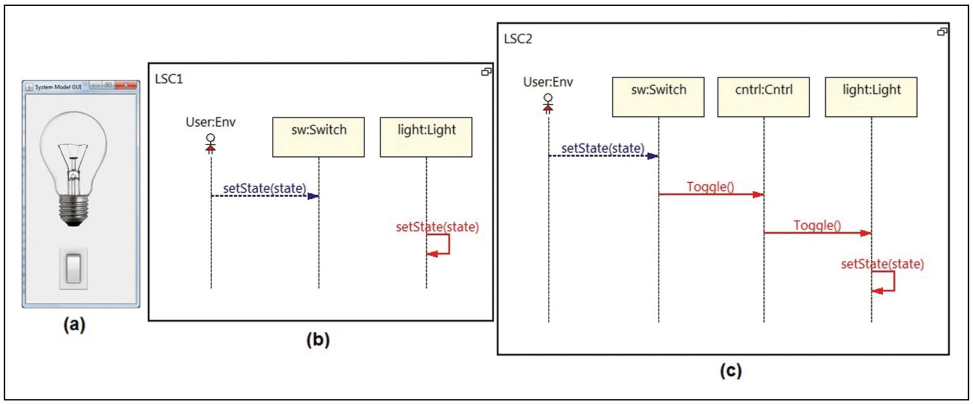

Figure 5. Switch–light system: requirements and design.

This integrated model, consisting of an LSC and a statechart, is also executable: when the user clicks the switch in the GUI, the statechart reacts to the event by calling the toggle method of the controller. This event is "caught" by PlayGo and is unified with the monitored message, thus allowing PlayGo to proceed, executing the next toggle message, and then turning the light on.

Gradually continuing with the implementation, the statechart of the controller can also be added, and then the one for the light. Each time a statechart takes responsibility for the actual triggering of events, the corresponding events in the LSCs are modified to be monitored, and can even be removed. Figure 7 shows the statecharts of the three components, with the original requirement now in monitored mode. This model is actually the final implementation of our system, since all components are fully implemented as statecharts. The LSC can now be omitted if we so choose, or it can be run together with the statecharts as a monitor for the requirement, confirming at runtime that the execution is indeed consistent with the requirements.

Although the example described here is extremely simple, it demonstrates the main idea of our proposed development cycle, and the fact that although the implementation of the various components is incremental, the system can be executed in full at any time during the development cycle.

Integration semantics and implementation

Semantics

Integrating any two runtime platforms calls for many decisions; for example, mapping the concepts of one platform into that of the other, concurrency

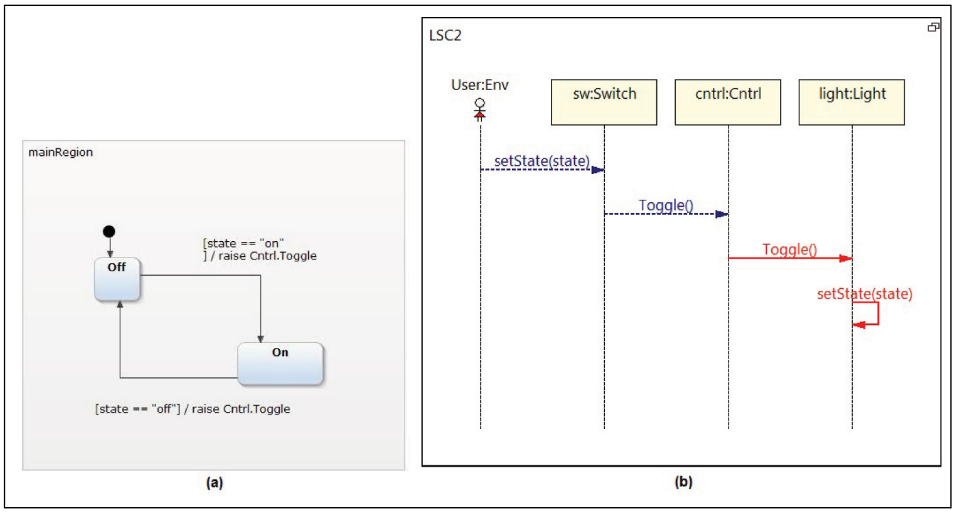

Figure 6. Switch–light system: implementing the switch.

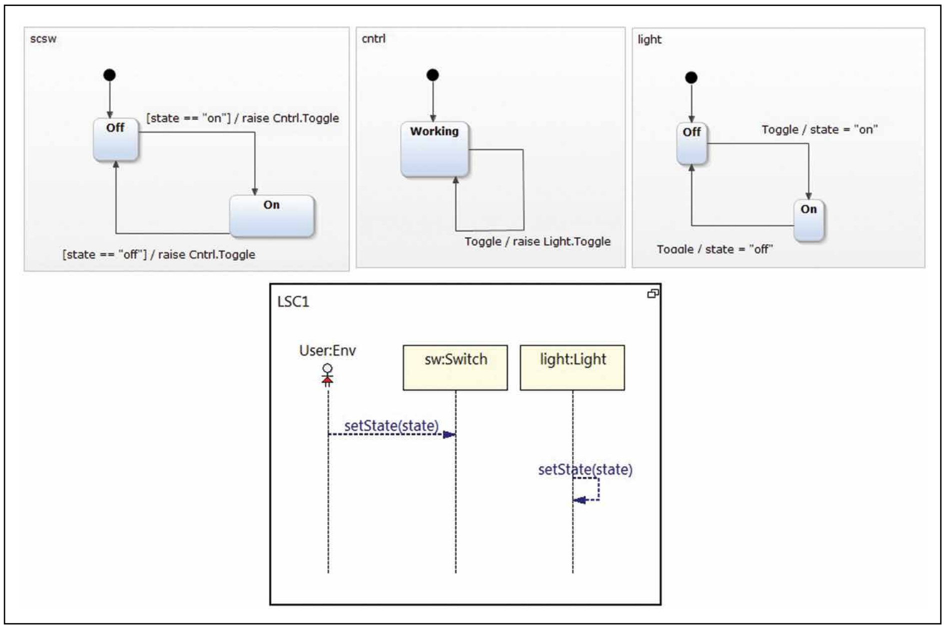

Figure 7. Switch–light system: fully implemented.

and priority in execution, data sharing, messaging protocols, synchronization, etc. The details of the semantics we adopted for integration and its implementation are provided in the supplementary material, at http://www.b-prog.org/sctlsc/sctlscsupp.pdf. Briefly, the key decisions we made were as follows.

- PlayGo is the host environment, controlling the two development environments, with smooth switching between the two, and the runtime environment, with the coordinated execution, data sharing and message exchanges. The runtime architecture is depicted in Figure 8. It relies on Execution Bridge to be able to interface with multiple kinds of models (Yakindu and others) and with multiple instances of any given model.
- The internal clocks of the two systems are synchronized.
- The generated Java code (of both PlayGo and Yakindu) can run without the development environments.
- Triggered statechart events have priority over LSC events that are enabled and ready to execute at the

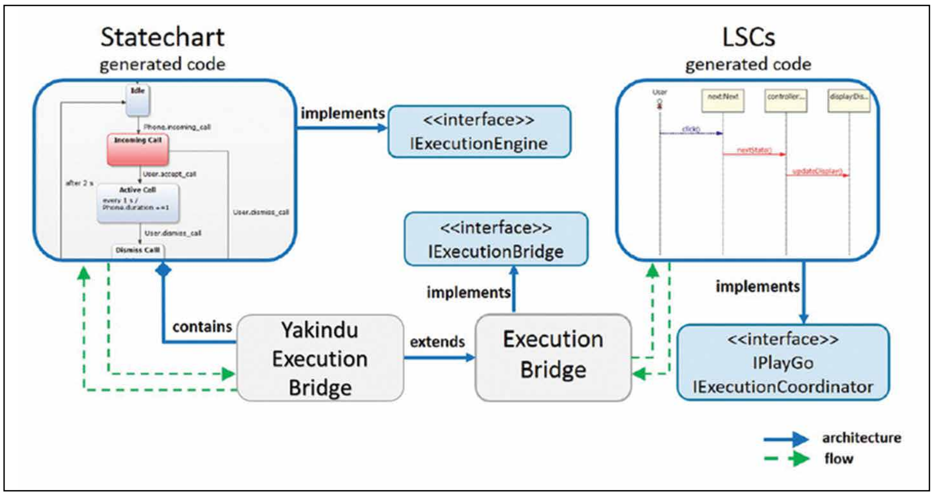

Figure 8. High level architecture of our implementation.

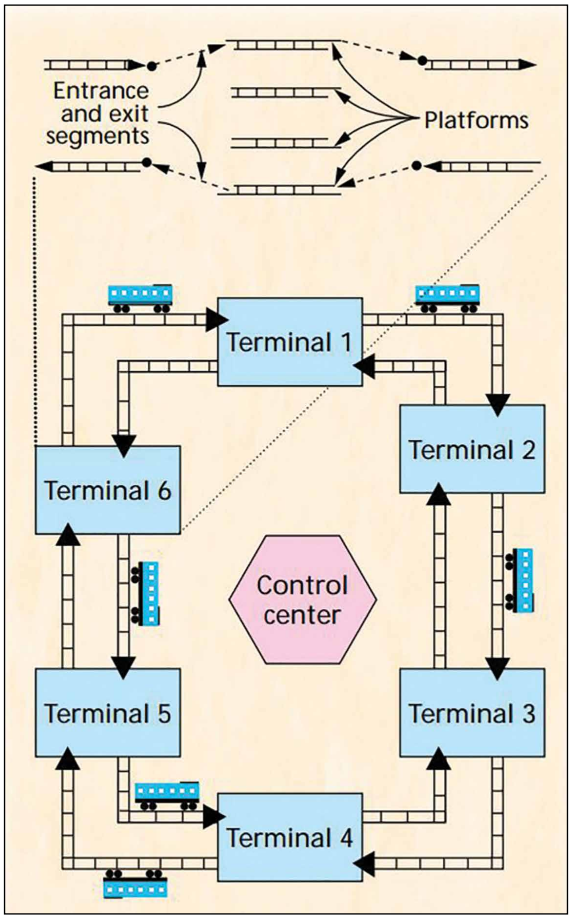

Figure 9. Railcar system. (Reproduced from [7], with permission.)

same time. This choice stems from the desire to allow the implementation, which is commonly developed later, and must run, to refine, and if needed, override the specification.

- Statechart events that are forbidden in an LSC will nevertheless occur and the resulting violation will be reported—as opposed to deferring the event until it is allowed, as would be the case with forbidden LSC events. The rationale for this choice is the same as that of the previous item.
- The object model is shared between the two platforms (see supplementary material at http://www.b-prog.org/sctlsc/sctlscsupp.pdf).[1]

In addition to the above, we provide a detailed mapping between LSC events (associated with messages or parameterized method calls) and the corresponding statechart event names defined under object interfaces in Yakindu (see supplementary material at http://www.b-prog.org/sctlsc/sctlscsupp.pdf).

Revisiting the railcar system

We now proceed to illustrate the capabilities of the methodology, and its semantics and the supporting tools, via the example appearing in the paper that introduced object-oriented statecharts [7] (see Figure 9), bringing the interobject versus intraobject duality to some kind of closure. For lack of space, our account here is rather brief, and a more detailed description appears in the supplemental material at http://www.b-prog.org/sctlsc/sctlscsupp.pdf.

The setting is as follows. Multiple terminals are connected by a cyclic path, consisting of two rail tracks, one for each direction of travel. Several railcars (abbreviated cars hereafter) transport passengers between terminals. A control center coordinates all activities. Each terminal has multiple platforms, and the incoming and outgoing rail segments are each connected to a short adjustable rail segment, within the terminal, which can be linked to any of the platforms.

Here are some requirement scenarios. They clearly illustrate the standalone, interobject "story" nature of scenarios in general.

- *Car approaching terminal:* When the car is 100 yards from the terminal, the system allocates a platform and an entrance segment, and, if the car is only passing through, also an exit segment. If the allocation is not completed when the car is within 80 yards from the terminal, the car must stop.
- *Car departing terminal:* A car departs the terminal 90 seconds after arrival. The system connects the platform to the outgoing track via the exit segment, engages the car's engine, and turns off the destination indicators on the terminal's destination board. The car can then depart, unless it is behind another car and within 100 yards of it.
- *Passenger in terminal:* When a passenger is in a terminal and no car in the terminal is traveling in the desired direction, the passenger can push a destination button and wait until a car arrives. If the terminal contains an idle car, it is assigned to that destination, otherwise a car is sent in from another terminal. The system indicates car availability on the destination board.

[1]In the future, we plan to make it possible for the user to choose semantic variations via plug-in code for event selection and execution-order policies.

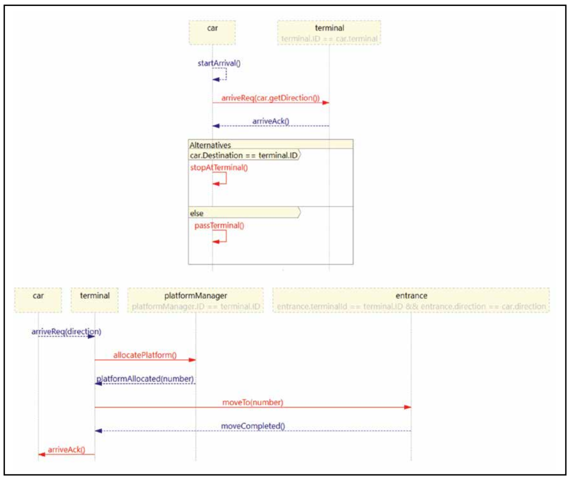
car
terminal
terminal.ID == car.terminal
startArrival()
arriveReq(car.getDirection())
arriveAck()
Alternatives
car.Destination == terminal.ID
stopAtTerminal()
else
passTerminal()
car
terminal
platformManager
platformManager.ID == terminal.ID
entrance
entrance.terminalId == terminal.ID && entrance.direction == car.direction
arriveReq(direction)
allocatePlatform()
platformAllocated(number)
moveTo(number)
moveCompleted()
arriveAck()

Figure 10. Top LSC: Car arrival at a terminal. The car first calls startArrival. It then sends an arriveReq message to that terminal and waits for acknowledgment. Depending on whether the next terminal is its destination or not, it stops or passes through. Bottom LSC: Arrival request. The terminal asks the platform manager to allocate a platform and waits for an approval containing the allocated platform's number. Then the car is sent to the corresponding entrance. Note in both LSCs the symbolic lifelines, which are concretized (instantiated) to specific objects via a binding expression; for example, in the top LSC, the car's terminal property is compared with the ids of all terminals.

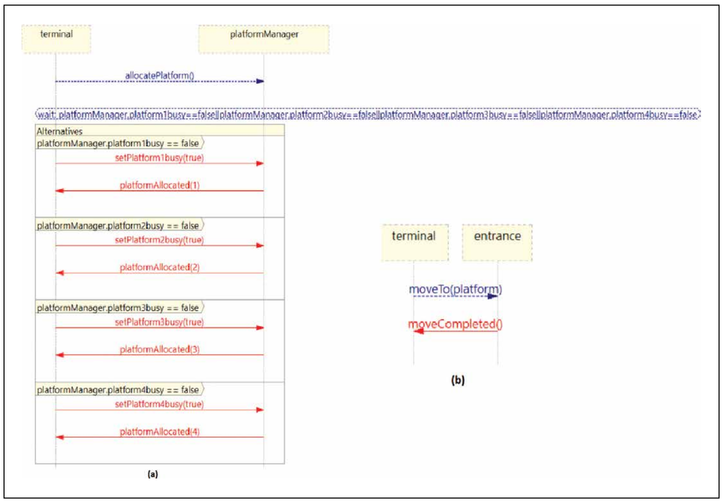
terminal
platformManager
allocatePlatform()
wait: platformManager.platform1busy==false||platformManager.platform2busy==false||platformManager.platform3busy==false||platformManager.platform4busy==false
Alternatives
platformManager.platform1busy == false
setPlatform1busy(true)
platformAllocated(1)
platformManager.platform2busy == false
setPlatform2busy(true)
platformAllocated(2)
platformManager.platform3busy == false
setPlatform3busy(true)
platformAllocated(3)
platformManager.platform4busy == false
setPlatform4busy(true)
platformAllocated(4)
(a)
terminal
entrance
moveTo(platform)
moveCompleted()
(b)

Figure 11. (a) Platform allocation: the platform manager waits for some platform to become available, allocates the first available platform and marks it as busy. (b) Simple LSC handling the handshake between the terminal and the entrance.

In [7], this system was programmed using object-oriented statecharts; see Figure 3 for one of those. In our approach here, we start by formalizing the requirements as LSCs, as exemplified in Figures 10 and 11.

The LSC in Figure 12 specifies that every time a car moves, each terminal checks whether the car is moving in its direction and has passed the minimal distance. Note the use of symbolic lifelines with multiplicity, indicating a scenario that applies to multiple terminals. Clearly, the implementation will differ

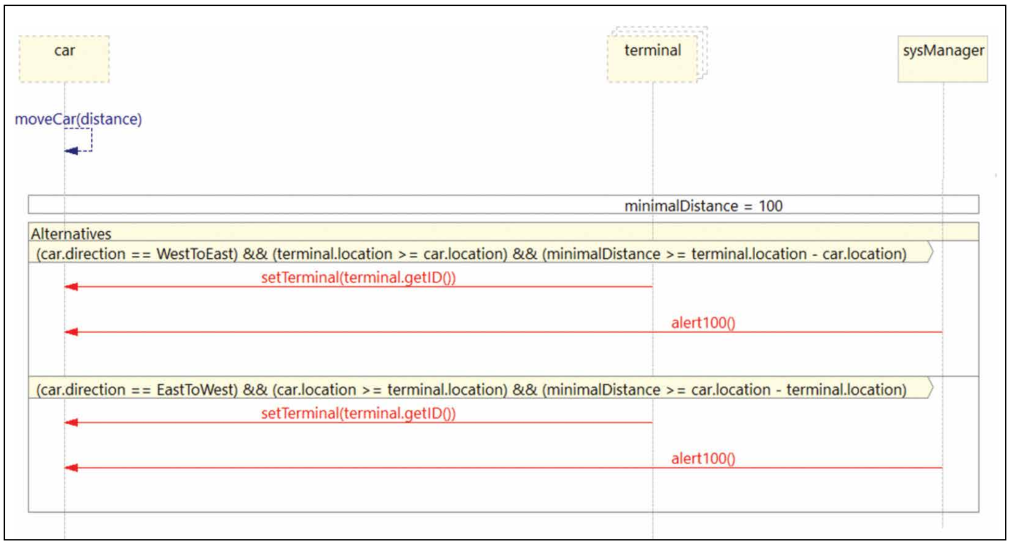
car
terminal
sysManager
moveCar(distance)
minimalDistance = 100
Alternatives
(car.direction == WestToEast) && (terminal.location >= car.location) && (minimalDistance >= terminal.location - car.location)
setTerminal(terminal.getID())
alert100()
(car.direction == EastToWest) && (car.location >= terminal.location) && (minimalDistance >= car.location - terminal.location)
setTerminal(terminal.getID())
alert100()

Figure 12. Alert 100.

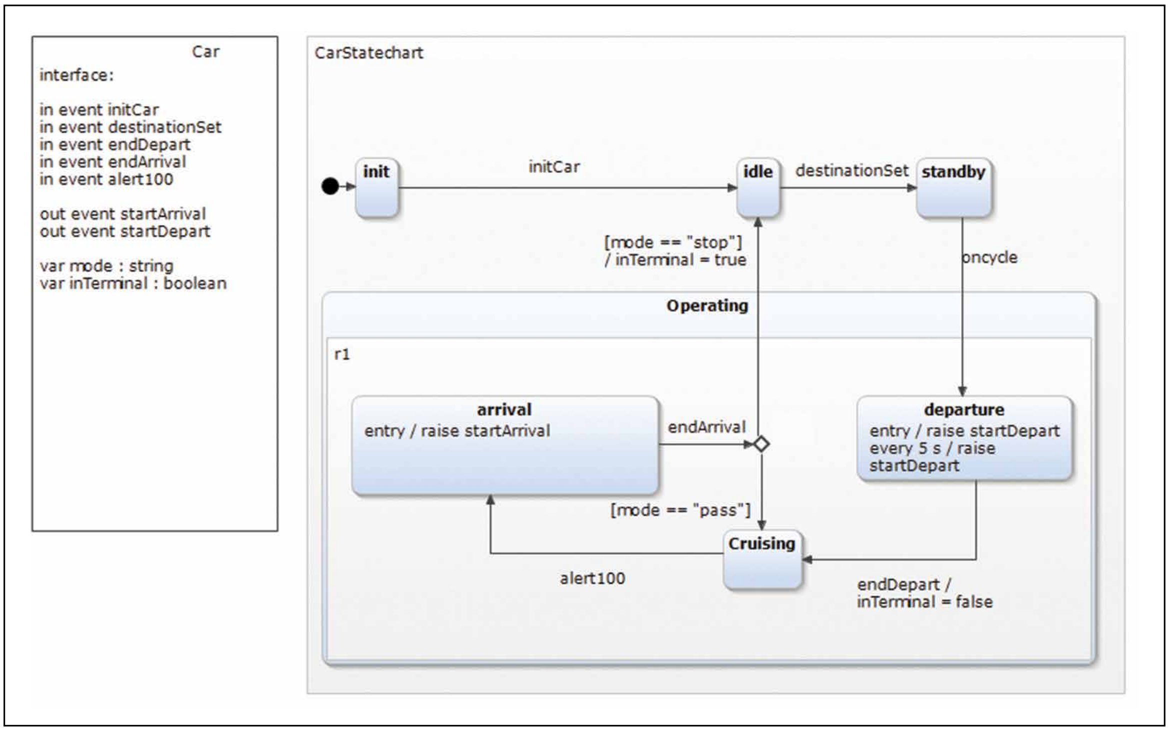

Figure 13. Car statechart.

from what is described in the scenario, but since we are in the requirements phase, we keep our scenarios abstract and ignore implementation and efficiency issues. If there is a terminal that meets the conditions of direction and proximity, it informs the approaching car and sends it its specific terminal number (following this action, the car sets its terminal variable to that number) and then the system manager sends an alert100 signal to the car.

In our implementation, we chose to implement the car on the intraobject level, while leaving the other objects (the car's "environment") at the requirements interobject level. Therefore, the car's statechart (Figure 13) reacts to all the input signals that are sent to the car and raises the output signals expected by the other objects.

The car's statechart is quite straightforward, so we will focus on the interaction with the LSCs. When the car is in state cruising, it waits for the alert 100 signal and reacts by moving to the arrival state and raising the startArrival signal (event). This signal belongs to the car's default interface and is therefore handled in the LSC as a self-method call. This event triggers the LSC in Figure 10. The car then waits for the endArrival event and moves to the idle or cruising state, depending on whether or not it should stop at that terminal. The endArrival event is raised by the LSC that handles the car's passing through the terminal and the LSC that handles its stopping at the terminal (Figure 14).

Related work

Transitions from scenario-based specifications to code, via state machines, can be found

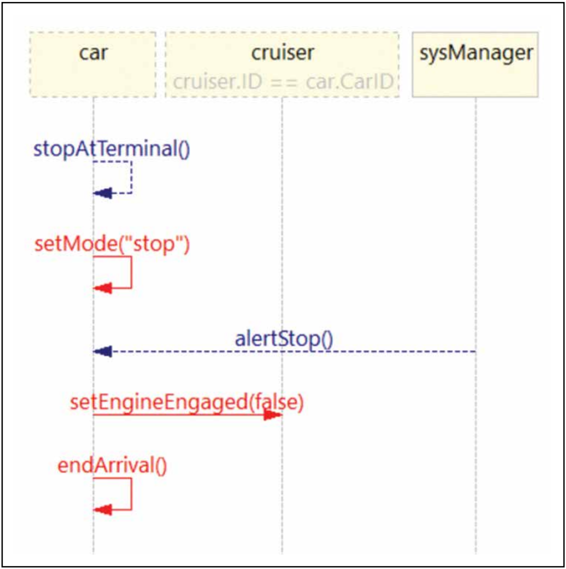

Figure 14. Stop at terminal.

in the large amount of work on synthesizing finite automata from MSC, sequence diagrams and LSCs (see [9] paper 212, [10], [5], and references therein). However, synthesis is often impractical, as the size of the resulting composite automaton grows exponentially with the number of scenarios and the allowed range of variable data and event parameters. To help combat state explosion, synthesis solutions often constrain the supported scope of certain expressive features in the original scenario language. Execution of final systems by playing out scenario-based specifications (as envisioned, e.g., in [9] paper 174), is not ready yet to fully materialize.

More robust solutions are needed for dealing with legacy code and engineers' preferences of languages. We also need adequate approaches for decomposing distributed systems. Hence, we believe that there is a need for the kind of integration mechanism and methodology proposed here, which allow human engineers to conduct a well-controlled, gradual transitioning from requirements to system implementation.

Bohn et al. [11] discuss showing different system views at various abstraction levels, verifying statecharts against LSC specifications and using LSCs to generate test vectors for the statecharts. However, the LSCs and statechart models in [11] are separate and their execution is not directly integrated. The Rhapsody tool (https://en.wikipedia.org/wiki/Rational_Rhapsody) supports monitoring statechart execution against the sequence diagrams specifications, but the sequence diagrams cannot influence the execution.

InterPlay (in [9] paper 136) is a tool developed in our group to link statecharts and LSCs. Its motivation was similar to that of the this article, and it provides a gateway that propagates and translates events between independently running LSCs and statechart engines. The contributions of the present research over Interplay include: a fully shared object model between the LSCs and the statecharts; an integrated and synchronized execution semantics, and a supporting mechanism that can also operate without the presence of the development environments (PlayGo and Yakindu, in this case).

In recent years, a number of efforts have been made to enable the joint simulation and analysis of models developed in different formalisms. These include Ptolemy II, with its multiple models of computation (MoC); ModHel'X, which combines semantics of multiple languages; the Epsilon Merging Language (EML), which provides a rule-based language for merging models of diverse meta-models and technologies; reusable aspect models (RAMs), which integrates structural models, message views, and state views using an aspect-oriented modeling technique; and the GEMOC-based BCOoL coordination language, which allows the specification of diverse semantics and integration between multiple languages. We have not been able to find a system development environment where the execution (or simulation) of (LSC-like) multimodal scenarios and (statechart-like) state machines can be truly integrated, with well-defined semantics.

In separate but related work, we have amalgamated statecharts with SBP, by extending Yakindu to allow associating individual states with requested and blocked events, and then enhancing the Yakindu event-triggering mechanism to deal with such specifications [12]. While this development allows an engineer to specify both scenarios and state-based reactivity in a single formalism, it is yet to be seen how the intuitiveness of the scenario's "story" and the clarity of the roles played by the participating objects, which are key tenets of sequence diagrams and LSCs, can be accomplished in statecharts. Is this an issue of design patterns, or of visual formatting? Perhaps it is another issue altogether.

We have presented a development environment and a methodology for incremental system development, starting with intuitive requirement scenarios and ending with object-oriented state machines, where throughout the process all artifacts are analyzable and executable, enabling simulation and validation at all stages. In addition, the availability of powerful versions of the two modeling approaches implemented in a single integrated tool simplifies developers' choice of the most suitable and naturally fitting language for the various parts of the system.

Future directions of work for enhancing the integration include: 1) finding a more straightforward mapping between parameterized LSC events and statechart events; 2) enabling semantic variations via user-supplied code; 3) enabling integration also with components written in standard procedural languages; and 4) incorporating into the integrated platform important techniques that have been developed for SBP or statecharts, such as formal verification, context-awareness, natural language input, execution with look-ahead (smart play-out), runtime learning, and more (see in [9], publications 230, 190, 112, 217).

We believe that a single tool and methodology for developing executable models in both interobject and intraobject approaches—supporting both requirement specification and implementation phases, and with means for smooth and semantically consistent transition between the two—can have a dramatic impact on the cost and quality of complex systems development. ■

Acknowledgments

The work of David Harel was supported in part by a grant from the Israel Science Foundation, the William Sussman Professorial Chair of Mathematics, and the Estate of Emile Mimran. We thank Axel Terfloth and Andreas Mülder of itemis corporation for their dedicated support of this project, and the anonymous reviewers for valuable comments and suggestions.

■ References

[1] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *J. Formal Methods Syst. Des.*, vol. 19, no. 1, pp. 45–80, 2001.

[2] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Berlin, Heidelberg: Springer, 2003.

[3] D. Harel, S. Maoz, S. Szekely, and D. Barkan, "PlayGo: Towards a comprehensive tool for scenario based programming," in *Proc. ASE*, 2010, pp. 359–360.

[4] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *ACM*, vol. 55, no. 7, pp. 90–100, 2012.

[5] J. Greenyer et al., "Scenario-based modeling and synthesis for reactive systems with dynamic system structure in ScenarioTools," in *Proc. MoDELS Demo Poster Sessions, Co-Located ACM/IEEE 19th Int. Conf. Model Driven Eng. Lang. Syst. (MoDELS CEUR)*, 2016, pp. 16–23.

[6] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[7] D. Harel and E. Gery, "Executable object modeling with statecharts," in *Proc. 18th Int. Conf. Soft. Eng.* Berlin, Germany: IEEE Press, Mar. 1996, pp. 246–257.

[8] Itemis Corporation. *Yakindu Statechart Tool Web Site*. Accessed: Oct. 2015. [Online]. Available: http://www.statecharts.org/

[9] D. Harel. (2019). *Personal Publication List.* Accessed: Jan. 2019. [Online]. Available: http://www.wisdom.weizmann.ac.il/harel/papers.html

[10] H. Liang, J. Dingel, and Z. Diskin, "A comparative survey of scenario-based to state-based model synthesis approaches," in *Proc. Int. Workshop Scenarios State Mach., Models, Algorithms, Tools*. New York, NY, USA: ACM, 2006, pp. 5–12.

[11] J. Bohn et al., "Modeling and validating train system applications using statemate and live sequence charts," in *Proc. IDPT*, 2002, pp. 1–9.

[12] A. Marron et al., "Embedding scenario-based modeling in statecharts," in *Proc. MORSE Workshop MoDELS*, 2018, pp. 443–452.

David Harel has been with the Weizmann Institute of Science, Rehovot, Israel, since 1980, where he was the Dean of the Faculty of Mathematics and Computer Science. He currently serves as the Vice President of the Israel Academy of Sciences and Humanities, Jerusalem, Israel. He has worked in logic and computability, software and systems engineering, and modeling biological systems. He invented Statecharts and co-invented Live Sequence Charts. 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, the Eme"t Prize, and five honorary degrees. He is a Fellow of ACM, IEEE, AAAS, and EATCS, a member of the Academia Europaea and the Israel Academy of Sciences, a foreign member of the U.S. National Academy of Engineering, the U.S. National Academy of sciences, and the American Academy of Arts and Sciences, and a Fellow of the Royal Society (FRS).

Rami Marelly held key technology leadership positions in the Israeli Air Force including the Head of C4I Systems Department and the Head of Aerial ISR Systems. As the Head Engineer, he led the IAF IT transformation toward network centric warfare and was responsible for the development of networking, avionics, simulators, C4I, and security systems. After retiring (Col. res.) from the IAF, he co-founded Cue, Israel, a consulting firm. He teaches advanced academic courses in systems engineering and volunteers as a mentor in various

FIRST robotics projects. His research was about specifying and executing behavioral requirements using the Play-in/Play-out approach. He has a PhD in computer science from the Weizmann Institute of Science, Rehovot, Israel.

Assaf Marron is a Researcher with the Weizmann Institute of Science's Computer Science and Applied Mathematics Department, Rehovot, Israel. Prior to joining the Weizmann Institute, he held senior management and technical positions in leading companies including IBM and BMC Software. He is the inventor or co-inventor of several patents. His research interests include software engineering and artificial intelligence. He has a PhD in computer science from the University of Houston, Houston, TX.

Smadar Szekely was the Head of software team in the research group of Prof. David Harel from 2009 to 2018. Prior to this, she worked as the R&D Director at SunGard Corporation, Herzliya, Israel, where she created complex mission-critical software products and defined corporate software development processes. She has a BA in computer science from Tel-Aviv Yaffo College, Tel Aviv-Yafo, Israel.

■ Direct questions and comments about this article to Assaf Marron, Weizmann Institute of Science, Rehovot 76100, Israel; assaf.marron@weizmann.ac.il.