Towards Synergistic Integration of Context-Based and Scenario-Based Development

Achiya Elyasaf[1], David Harel[1], and Assaf Marron[1] and Gera Weiss[2]

[1] Weizmann Institute of Science, Rehovot, Israel

[2] Ben-Gurion University of the Negev, Beer Sheva, Israel

Abstract. Scenario-based models of reactive systems divide complex specifications into artifacts corresponding to separate aspects of overall system behavior, as they may appear, e.g., in a robot's requirements document or user manual. The advantages of scenario-based development include intuitiveness and clarity, the ability to execute or simulate specifications of early prototypes and of final systems, and the ability to verify the specification for early detection of conflicts, omissions, and errors. In this position paper we discuss two issues that emerge when applying scenario-based development in complex cases: (a) simple scenarios become unwieldy when subjected to a growing number of conditions, exceptions and refinements, and (b) it is hard to understand and maintain a large 'flat' specification, consisting of many independently-specified scenarios, simple as they may individually be. We address these by basing certain facets of scenario design on *context*, an increasingly popular foundational consideration in software engineering. We first show how one can incorporate context into the graphical language of live sequence charts (LSC) using existing LSC idioms. We then outline two other possibilities: (i) enriching the LSC language, or (ii) embedding LSCs within hierarchical state machines, namely, statecharts. We believe that this research can contribute to the broader goals of developing complex and powerful systems in intuitive and robust ways.

1 Introduction

In *scenario-based programming* (SBP) one develops software and systems such that distinct aspects of overall system behavior are implemented in separate behavioral modules, termed *scenarios*. For example, individual paragraphs of a requirement document for a robotic system, or its user manual, are likely to be implemented as separate scenarios. The approach was first introduced by Damm, Harel and Marelly in [4,9] with the graphical language of *live sequence charts* (LSC). It was subsequently generalized and implemented in procedural languages such as Java, C++ and JavaScript [10] . Strengthened with suitable tools, SBP (a.k.a. *behavioral programming*) was shown to have a broad range of advantages, including intuitiveness, clarity and succinctness of specifications, the ability to directly execute or simulate specifications

of early prototypes and of final systems, and the ability to verify such specifications in order to facilitate early detection of conflicts, omissions, and errors (see, e.g., [6–8] and references therein). Research and development on SBP has tried to tackle what sometimes appear to be fundamental difficulties thereof. An example is whether SBP increases the risk of specification conflicts, the answer being that such conflicts often exist already in the original human-prepared requirements and that the SBP verification tools in fact contribute to their early discovery and resolution.

Two new issues now arise, as research and development of SBP matures and renders the approach suitable for more complex environments and tasks. The first is that scenarios that start out as simple rules, become complex and unwieldy when subjected to a growing number of conditions, exceptions and refinements. Consider for example a (futuristic) home-assistant robot, which needs to automatically detect and clean up dropped or spilled food and drink. Though the sensors and actuators that are needed for such competency are quite elaborate, the behavioral rules themselves seem quite simple. *"When something drops, clean it up immediately"*, etc. However, a long list of exceptions could then be required: *"but not in the middle of the night"* , or *"not when someone is asleep, or on the phone in the kitchen"*, or *" not when the dirt canister is full"*, etc. Without careful design, such many-to-many relationships between conditions and actions can turn a simple specification into a 'spaghetti' of exceptions, refinements and alternative paths.

The second issue is that even if individual scenarios are simple, a large specification may nevertheless become hard to understand and maintain. The powerful runtime composition of scenarios in SBP is not reflected in the code of the individual scenarios. Also, SBP does not provide ways to capture the organizational dimensions that may exist in the engineer's mind. For example, in a use case similar to the DARPA-challenge, a robot that has to drive a car designed for humans, negotiate a pile of rubble, climb a ladder and close a valve may have to deal with a vast number of scenarios. Their interrelationships and inter-dependencies may be handled correctly at run time, but would be difficult for an engineering team to manage during development, where there is need to allocate development tasks, demonstrate partial prototypes, and plan systematic testing. In fact, the intuitiveness of a requirements document or user manual stems not only from the individual sentences (in natural language) but from the overall organization of the document, which eliminates the need to state certain things explicitly, because they are clearly understood from the context, and allows refinement and out-of-context cross-referencing, e.g., via appendices.

Clearly, adequate solutions to these two issues would align nicely with the concept of context and context awareness, which have been addressed extensively in software engineering. In this paper, following a brief introduction to scenario-based programming and a discussion of general view on context awareness, we propose solutions to these two issues, which rely on existing LSC constructs; that is, they do not require additional language idioms or execution infrastructure. The result is an approach that further enables the creation of specifications that are intuitive, expressive and powerful, and, most importantly, are executable by a computer. We then proceed to briefly discuss separate research activities and new language

constructs aimed at further simplifying the incorporation of context awareness into scenario-based programming.

2 Scenario-Based Programming with Live Sequence Charts

The LSC language extends classical message sequence charts with rich syntax and semantics that enable intuitive event-based abstraction of behavior to serve both as formal specifications and as the running code in the compositional execution (termed *play-out*) of the final system. The PlayGo tool provides an interactive development and simulation environment, as well as a stand-alone, independent, system run-time infrastructure for LSC. Similar syntax and semantics were adopted in UML sequence diagrams (SD). Each LSC chart (see example in Fig.1 depicts a scenario of system behavior. Behavior is represented as event arrows between vertical lines representing objects, with time flowing from top to bottom. Blue and red distinguish events that may happen from those that must happen, and solid arrows represent requests to execute/trigger events while a dashed arrow depicts events that should be merely waited for; i.e., monitored. Other notations can specify forbidden events, control flow elements like if-then-else and loops, and more. The play-out algorithm runs all scenarios in parallel, in a fully synchronized, lockstep manner. Following an environment event, all affected scenarios advance; their declarations of what event must, may , or must not be triggered are consolidated, and a non-forbidden event is selected according to a prescribed strategy (random, priority, or based on look-ahead). All scenarios are notified of this selection and affected ones proceed accordingly. When all system reactions are exhausted, the next external, environment-generated event can be dealt with.

3 Context-based Specifications

There are many approaches to context-oriented programming and to endowing procedural programs with context awareness (see, e.g., [1,3,11,12]. We will not delve into the relevant definitions here, nor will we discuss how these context-related approaches differ from dealing with the various common conditions that can affect program execution. Instead, we hope that readers will find that the way we deal here with intuitive executable specifications that can be subject to complex conditions fits a variety of needs and design patterns that could be fairly included under the general term of being context based. Nevertheless, to set expectations right, additional context-related examples of the kind we would like to handle include: whether a client or server in a distributed application is initiating an interaction (in sending mode) or listening out for notifications (in receiving mode); how presence of (or engagement with) a human affects an industrial robot's entire operation; how the location (in orbit or on the ground) affects an autonomous satellite's handling of events and conditions; how battery charge level of a mobile phone affects autonomous features; how an

autonomous car's speed is to be affected by a narrow and/or curved road and by lighting conditions.

A contextual condition is not necessarily external and uncontrollable: a robot encountering poor lighting conditions might be able to turn on additional lights and change the context. We also ignore the fact that particular contextual information ('battery is low'), may also be part of a very particular condition (''battery is now 7% full");

An additional motivation for context-based designs is the ability to incrementally constrain the system or accelerate verification. Thus, the design and testing of a readyto-ship home-assistant robot could have assumed typical indoor lighting, when a lastminute concern is raised that the robot may not work properly in a dark room or a sunlit porch. A makeshift solution would limit the entire robot behavior to certain lighting conditions, and when these are not met activity is paused. Similarly, when verification or extensive testing are to be carried out, the size of the state space or the extent of test-coverage goals can be dramatically reduced using context awareness to enforce simplifying assumptions.

4 Context-based Design in Native LSC

A key methodological point we propose is that contexts should play a primary role in initial analysis and design. Entities that would otherwise be modeled as object properties or emergent inter-object behavior (someone asleep in the house; the collaboration of two robots), should be identified as objects very early on.

The main principle underlying our approach to context-based design in native LSCs (abbreviated CBLSC) is that instead of refining scenarios by adding context conditions locally, just prior to triggering the actions that depend on them, the activation or relevance of entire scenarios should be made subject to the presence of desired contexts. Here are the main language features and design patterns for doing this (see Fig. 1):

Dynamic objects. In LSC, objects of all types can be created and destroyed dynamically. This can be done from any scenario by executing an appropriate event.

Binding expressions. The binding of a lifeline to an object instance can be subjected to a binding expression, that specifies one, or all, the instances to be bound (in the latter case the scenario is replicated).

Dynamic binding. By default a scenario is not active. When a monitored event appearing as the very first in a scenario occurs (triggered by the environment or by another scenario), the scenario is activated and a new *live copy* thereof participates in play-out until its termination. During this, lifelines are bound to objects but when no instance satisfies the binding expression, the live copy terminates.

Context objects. It is a common to infer a current context by checking the values of object properties, like babyIsAsleep==true or batteryPercent<10. The relevant objects usually persist despite changes in these values. By contrast, in CBLSC, whether

a context holds or not could depend on whether or not an object like babyIsAsleep or batteryLow is instantiated.

Scenario-driven creation of context objects. The examination of possibly complex conditions and events that determine whether a context holds is done in one or more dedicated scenarios. Composite contexts can be similarly created by monitoring conjunctions, disjunctions and other relationships of other contexts.

Subjecting scenarios to context objects. Scenarios specify context dependencies by having lifelines for relevant context objects (even if no events occur therein). When contexts apply to only a small part of a scenario, one can split the scenario into its parts, or replicate, or use condition constructs instead.

Context-termination handling. Graceful context termination is still the developer's task. Either all affected scenarios should be immediately terminated, activating scenarios to handle the new situation (including completing certain actions as in exception handling), or use events to notify active scenarios that they need to terminate ASAP.

Summary. With the above constructs, each scenario can thus specify separately and concisely both the required behavior and the contexts in which the specified reactivity applies.

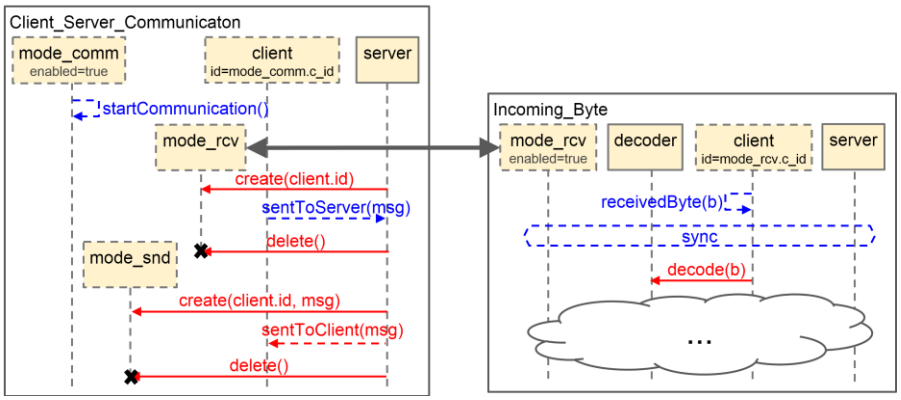

Fig.1: Creating a context object (left) and subjecting a scenario to that context (right).

5 Towards Intuitive Organization of Context-based Specifications

To streamline the management of 'large flat collections of independently-specified scenarios, we propose to add the following support to specification management in LSC tools:

Textual scenario views. Optionally hide the graphical chart view, and show select details thereof, such as name, text comments, affected objects, relevant contexts, key events, or a textual description of the scenario's flow (now available automatically).

Navigable Specifications. Navigate specifications according to function, structure and dependencies implied by the above key details, starting with contexts. At present this can be done manually, and can be automated by adding required indexing. Views would include queries and filters, sorted lists, hierarchical trees, graphs, rectangle-containment, etc.

Feature-model-like Design of Context Awareness. We propose to design context/scenario relationships to resemble feature models [2] and software product lines, aligning contexts with key user requirements, system functions, or target environments.

Multi-Hierarchies of Contexts. We believe that humans find it easier to understand and manage contexts that are hierarchical, with sub-division of properties such as time and location. As orthogonal hierarchies intersect, they can still be navigated and understood using the above idioms. Intuitive visual representations, such multi-hierarchies, include a forest of tree hierarchies. Intersections can be shown with directed edges between trees while keeping the entire graph acyclic, or by connecting context nodes from different trees to a common set of scenarios.

6 Research on New Language Idioms

We are pursing two additional lines of work related to context orientation. One is adding specific syntax and semantics to LSC for creating and destroying context objects, subjecting scenarios to contexts, and other features related to context based design (see a report in www.wisdom.weizmann.ac.il/ harel/morse17s)

Another direction is based on embedding LSC within the intuitive hierarchical structure of statecharts [5], which allows both state containment and orthogonality, which in turn align well with contexts. Contexts will be associated with statechart states, and LSC scenarios that are associated with a context state will participate in play-out only when the system transitions into that state. This will be complementary to our work on incorporating statecharts within an LSC scenario (see a report in above location). Another advantage of statecharts is that their concurrency feature, namely, the ability to condition a transition on whether other orthogonal parts of the system are in a certain state, is an excellent basis for implementing multi-hierarchies.

7 Conclusion

We believe that contexts are a central concept in system analysis, and have shown how context-based design can be incorporated into executable, scenario-based specifications, using existing LSC idioms. We have also outlined approaches for making the entire specification easier to understand via navigational features, new language idioms, and integration with statecharts. We hope that our work will contribute to the search for languages that can produce intuitive models that are also powerful, executable programs.

Acknowledgments

This work has been supported in part by a grant to David Harel from the Israel Science Foundation, the William Sussman Professorial Chair of Mathematics, and the Estate of Emile Mimran.

References

1. G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a betterunderstanding of context and context-awareness. In *International Symposium on Handheld and Ubiquitous Computing*, pages 304–307. Springer, 1999.
2. S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.
3. M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. Contextj: Context-oriented programming with java. *Information and Media Technologies*, 6(2):399–419, 2011.
4. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 2001.
5. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
6. D. Harel. Can Programming Be Liberated, Period? *IEEE Computer*, 41(1):28–37, 2008.
7. D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On Composing andProving the Correctness of Reactive Behavior. In *EMSOFT*, 2013.
8. D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the Succinctness of Idioms forConcurrent Programming. In *CONCUR*, pages 85–99, 2015.
9. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
10. D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Comm. of the ACM*, 2012.
11. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.
12. R. Keays and A. Rakotonirainy. Context-oriented programming. In *MobiDE'03*, 2003.