Towards Integration of Context-Based and Scenario-Based Development

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

[1] Weizmann Institute of Science, Rehovot, Israel
assaf.marron@weizmann.ac.il

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

Abstract. In scenario-based models of reactive systems complex specifications are divided into artifacts corresponding to separate aspects of overall system behavior, as they may appear, e.g., in a robot's requirements document or user specifications. 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 an unorganized list of independently-specified scenarios, simple as they may individually be. We address these issues 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 reactive 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, both desired and forbidden, are implemented in separate behavioral modules, termed *scenarios*. For example, individual paragraphs of a requirement document or of a user manual for a robotic system, 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

© Springer International Publishing AG 2018
M. Seidl and S. Zschaler (Eds.): STAF 2017 Workshops, LNCS 10748, pp. 225–231, 2018.
https://doi.org/10.1007/978-3-319-74730-9_21

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). The research and development of SBP has regularly tried to tackle emerging new challenges that appear (at least at first) to be particular to SBP. An example for such research question is whether the SBp call for independent specification of scenarios increases the risk of specification conflicts. The answer turns out to be that not only do such conflicts often exist already in the originally-stated requirements, but also that the abstractions provided by SBP, and the verification tools that were developed in fact contribute to the early discovery and resolution of such conflicts.

As research of SBP matures and renders the approach suitable for more complex tasks, two new issues arise. 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, coming from all stakeholders, as well as from standards and regulations, as is commonly expected in real-world applications. Consider for example a (futuristic) home-assistant robot which needs to automatically detect and clean up dropped or spilled food. Though the required sensors and actuators are quite sophisticated, the behavioral rules themselves seem simple: *"When food is dropped, clean it up."*. However, many exceptions can then emerge: *"but not late at night"* (due to vacuum-cleaner noise), or *"but not when anyone is asleep or is on the phone in the same room."*, or *" but not when the dirt canister is full."*, etc. Without careful design, such many-to-many relationships between environment 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 become hard to understand and maintain. SBP's powerful scenario composition is not visible in the individual scenarios, and there is no direct way to capture the organization 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, walk over a pile of rubble, climb a ladder and close a valve has to deal with many scenarios. The scenarios' dependencies may be handled correctly at run time, but, during development, it may be difficult for engineers to allocate development tasks, demonstrate partial prototypes, and plan systematic testing. In fact, the intuitiveness of requirement documents and user manuals stems not only from the natural language of individual sentences but from the document organization, which allows both omission of what is understood from the context, and out-of-context cross-referencing (as is done, e.g., in appendices).

Clearly, solutions to these two issues would align well 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 and do not require new language idioms or run-time 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 even greater simplification of adding context awareness to scenario-based programming.

2 Scenario-Based Programming with Live Sequence Charts

The LSC language extends *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, and a stand-alone LSC run-time infrastructure. 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 dashed arrows depict events that should be merely waited for, i.e., monitored. Other notations specify forbidden events, if-then-else conditions, 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 events *must*, *may*, or *must not* be triggered are consolidated, and an event is selected according to a prescribed strategy (random, priority, or based on look-ahead). All scenarios are notified of this selection and the affected ones proceed accordingly. When all system reactions are complete, 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 context-related approaches differ from dealing with environment conditions in standard programming. Instead, we hope that readers will find that our proposals for how to subject intuitive executable specifications to complex conditions fit a variety of needs and design patterns that can qualify as being context based. Nevertheless, to properly set readers' expectations, below are additional context-related examples of the kind we would like to handle: e.g., 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 a human, or collaboration with one, affects an industrial robot's operation; how the location

(in orbit or on the ground) affects an autonomous satellite's handling of events; how battery- charge level of a mobile phone affects its autonomous features; or, how an autonomous car's speed is to be affected when the road is narrow and/or curved and/or poorly lit.

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 here the fact that particular contextual information ("battery is low"), may also be part of a very particular condition ("battery is now 7% full");

Context-based designs also allow one to incrementally constrain the system. E.g., if the design (and testing) of a home-assistant robot assumed only typical indoor lighting, and a last-minute pre-shipping concern questions its functioning in a dark room or in a sunlit porch, a makeshift solution can be to physically limit the entire robot behavior to indoor lighting conditions, and when these are not present to pause all activities. 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 reduced using context awareness, to enforce simplifying assumptions throughout.

4 Context-Based Design in Native LSC

First, a key methodological point we propose is that contexts should play a primary role in initial analysis. Hence, entities that may otherwise be modeled as properties or as inter-object behavior (e.g., the facts that someone is asleep in the house, or that two robots are collaborating) may need to be modeled as objects in their own right.

Second, for context-based design in native LSCs (abbr. CBLSC), we propose that instead of refining scenarios as one would refine ordinary programs, i.e., by adding context conditions locally prior to triggering the actions that depend on them, the activation (namely, the very relevance) of entire scenarios should be subjected to the presence of desired contexts, as follows (see also 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 the instance(s) to be bound (if more than one, the scenario is replicated).

Dynamic binding. By default a scenario is not active. When a monitored event that appears as the first one in a scenario, is triggered by the environment or by the system, the scenario is activated and a new *live copy* thereof participates in play-out until its termination. Lifelines are then bound dynamically as specified. But, if there is a lifeline that cannot be bound, as no object satisfies its binding expression, the live copy terminates.

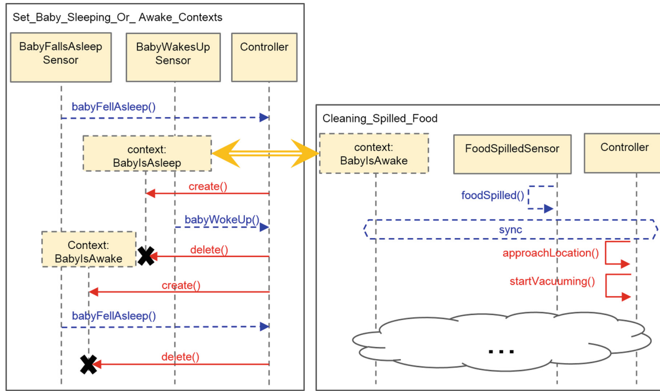

Fig. 1. Creating some context objects for a home-assistant robot (left) and subjecting a noisy home-cleaning scenario to one such context (right). (Color figure online)

Context objects. It is 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 in these lifelines). 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, e.g., terminate the affected scenario immediately; activate scenarios to handle the new situation (including completing in-flight activities 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 concisely specify both the required behavior and the contexts in which the specified reactivity applies.

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 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, context, structure and dependencies. This can be done by adding indexing, queries and filters, sorted lists, trees and rectangle-containment views, dependency graphs, 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 (r.g., day vs. night and then specific hour) or location (e.g., city, street, building, room). As orthogonal hierarchies intersect, they can still be navigated and understood using the above idioms. Intuitive visual representations, such as 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.

Summary. Once scenarios are both *subjected to* and *organized by* contexts, several potential advantages emerge as compared with implementing contexts as in-line conditions (clearly, empirical quantification and assessment remain as future work): (a) When a scenario is subjected to context objects such as `No_One_Is_Asleep` or `DayTime`, it is clear that it is applicable when the context conditions do hold. With statements like `if NoOneIsAsleep` or `if TimeOfDay >= 22 and TimeOfDay <= 07`, even rich classical search commands cannot readily inform of both the properties checked, their desired values, and the actions that are taken or skipped. (b) With contexts, one can readily check against the requirements which scenarios are applicable in which contexts without examining implementation code. (c) When in-line conditions are replaced with contexts, each scenario can be better understood as doing just one or very few tasks.

6 Research on New Language Idioms

We are pursuing 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.b-prog.org/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 `www.b-prog.org/morse17s`). 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.

Another relevant question is whether a scenario should have access to context objects in which it is invoked, with all their details, causes, and other dependencies.

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.

References

1. Abowd, G.D., Dey, A.K., Brown, P.J., Davies, N., Smith, M., Steggles, P.: Towards a better understanding of context and context-awareness. In: Gellersen, H.-W. (ed.) HUC 1999. LNCS, vol. 1707, pp. 304–307. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48157-5_29
2. Apel, S., Kästner, C.: An overview of feature-oriented software development. J. Object Technol. **8**(5), 49–84 (2009)
3. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: context-oriented programming with Java. Inf. Media Technol. **6**(2), 399–419 (2011)
4. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. J. Form. Methods Syst. Des. **19**, 45–80 (2001)
5. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
6. Harel, D.: Can programming be liberated, period? IEEE Comput. **41**(1), 28–37 (2008)
7. Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., Weiss, G.: On composing and proving the correctness of reactive behavior. In: EMSOFT (2013)
8. Harel, D., Katz, G., Lampert, R., Marron, A., Weiss, G.: On the succinctness of idioms for concurrent programming. In: CONCUR, pp. 85–99 (2015)
9. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-19029-2
10. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM **55**, 90–100 (2012)
11. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. J. Object Technol. **7**(3), 125–151 (2008)
12. Keays, R., Rakotonirainy, A.: Context-oriented programming. In: MobiDE 2003 (2003)