

# 1. Introduction

## 1.1 What Are We Talking About?

What kinds of systems are we interested in? Well, first and foremost, we have in mind computerized and computer embedded systems, mainly those that are reactive in nature. For these **reactive systems**, as they are called, the complexity we have to deal with does not stem from complex computations or complex data, but from intricate to-and-from interaction — between the system and its environment and between parts of the system itself.

Interestingly, reactivity is not an exclusive characteristic of man-made computerized systems. It occurs also in biological systems, which, despite being a lot smaller than us humans and our homemade artifacts, can also be a lot more complicated, and it also occurs in economic and social systems, which are a lot larger than a single human. Being able to fully understand and analyze these kinds of systems, and possibly to predict their future behavior, involves the same kind of thinking required for computerized reactive systems.

When people think about reactive systems, their thoughts fall very naturally into the realm of **scenarios of behavior**. You do not find too many people saying things like “Well, the controller of my ATM can be in waiting-for-user-input mode or in connecting-to-bank-computer mode or in delivering-money mode; in the first case, here are the possible inputs and the ATM’s reactions, ...; in the second case, here is what happens, ..., etc.”. Rather, you find them saying things like “If I insert my card, and then press this button and type in my PIN, then the following shows up on the display, and by pressing this other button my account balance will show”. In other words, it has always been a lot more natural to describe and discuss the reactive behavior of a system by the scenarios it enables rather than by the state-based reactivity of each of its components. This is particularly true of some of the early and late stages of the system development process — e.g., during requirements capture and analysis, and during testing and maintenance — and is in fact what underlies the early stage use case approach. On the other hand, it seems that in order to *implement* the system, as opposed to stating its required behavior or preparing test suites, **state-based modeling** is

needed, whereby we must specify for each component the complete array of possibilities for incoming events and changes and the component's reactions to them.

This is, in fact, an interesting and subtle duality. On the one hand, we have scenario-based behavioral descriptions, which cut across the boundaries of the components (or objects) of the system, in order to provide coherent and comprehensive descriptions of scenarios of behavior. A sort of **inter-object**, 'one story for all relevant objects' approach. On the other hand, we have state-based behavioral descriptions, which remain within the component, or object, and are based on providing a complete description of the reactivity of each one. A sort of **intra-object**, 'all pieces of stories for one object' approach. The former is more intuitive and natural for humans to grasp and is therefore fitting in the requirements and testing stages. The second approach, however, has always been the one needed for implementation; after all, implementing a system requires that each of the components or objects is supplied with its complete reactivity, so that it can actually run, or execute. You can't capture the entire desired behavior of a complex system by a bunch of scenarios. And even if you could, it wouldn't be at all clear how you could execute such a seemingly unrelated collection of behaviors in an orderly fashion. Figure 1.1 visualizes these two approaches.

This duality can also be explained in day-to-day terms. It is like the difference between describing the game of soccer by specifying the complete reactivity of each player, of the ball, of the goal's wooden posts, etc., vs. specifying the possible scenarios of play that the game supports. As another example, suppose we wanted to describe the 'behavior' of some company office. It would be a lot more natural to describe the inter-object scenarios, such as how an employee mails off 50 copies of a document (this could involve the employee, the secretary, the copy machine, the mail room, etc.), how the boss arranges a conference call with the project managers, or how information on vacation days and sick leave is organized and forwarded to the payroll office. Contrast this with the intra-object style, whereby we would have to provide complete information on the modes of operation and reactivity of the boss, the secretary, the employees, the copy machine, the mail room, etc.

We are not claiming that scenario-based behavior is technically superior in some global sense, only that it is a lot more *natural*. In fact, now is a good time to mention that mere isolated scenarios of behavior that the system can possibly give rise to are far from adequate. In order to get significant mileage out of scenario-based behavior, we need to be able to attach various modalities to the scenarios we are specifying. We would like to distinguish between scenarios that *may* occur and those that *must*, between those that occur spontaneously and those that need some trigger to cause them to occur.

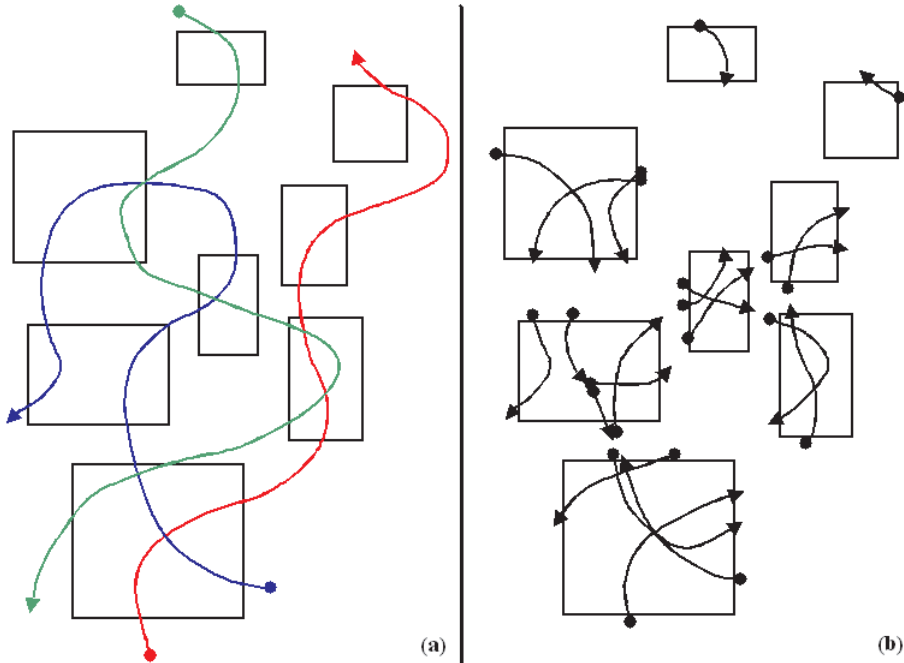


Fig. 1.1. Inter-object vs. intra-object behavior

We would like to be able to specify multiple scenarios that combine with each other, or even with themselves, in subtle sequential and/or concurrent ways. We want generic scenarios that can be instantiated by different objects of the same class, we want to be able to use variables to store and retrieve values, and we want means for specifying time. Significantly, we would also like to be able to specify **anti-scenarios**, i.e., ones that are forbidden, in the sense that if they occur there is something very wrong: either something in the specification is not as we wanted, or else the implementation does not correctly satisfy the specification.

Obviously, it would also be very nice if we could actually ‘see’ scenario-based behavior in operation, before (or instead of?) spending lots of time, energy and money on intra-object state-based modeling that leads to the implementation. In other words, we could do with an approach to inter-object behavior that is expressive, natural and executable.

This is what the book is about.

## 1.2 What Are We Trying to Do?

We propose a powerful setup, within which one can conveniently capture scenario-based behavior, and then execute it and simulate the system under development exactly as if it were specified in the conventional state-based fashion. Our work involves a language, two techniques with detailed underlying algorithms, and a tool. The entire approach is made possible by the language of **live sequence charts**, or **LSCs**, which is extended here in a number of ways, resulting in a highly expressive medium for scenario-based behavior. The first of our two techniques involves a user-friendly and natural way to **play in** scenario-based behavior directly from the system's GUI (or some abstract version thereof, such as an object-model diagram), during which LSCs are generated automatically. The second technique, which we consider to be the technical highlight of our work, makes it possible to **play out** the behavior, that is, to execute the system as constrained by the grand sum of the scenario-based information. These ideas are supported in full by our tool — the **Play-Engine**.

There are essentially two ways to view this book. The first — the more conservative one — is to view it as offering improvements to the various stages of accepted life-cycles for system development: a more convenient way to capture behavioral requirements, the ability to express more powerful scenario-based behavior, a fully worked-out formalization of use cases, a means for executing use cases and their instantiations, tools for the dynamic testing of requirements prior to building the actual system model or implementation, a highly expressive medium for preparing test suites, and a means for testing systems by dynamic and run-time comparison of two dual-view executables.

The second way to view our work is less conservative. It calls for considering the possibility of an alternative way of programming the behavior of a reactive system, which is totally scenario-based and inter-object in nature. Basic to this is the idea that LSCs can actually constitute the implementation of a system, with the play-out algorithms and the Play-Engine being a sort of 'universal reactive mechanism' that executes the LSCs as if they constituted a conventional implementation. If one adopts this view, behavioral specification of a reactive system would not have to involve any intra-object modeling (e.g., in languages like statecharts) or code.

This of course is a more outlandish idea, and still requires that a number of things be assessed and worked out in more detail for it to actually be feasible in large-scale systems. Mainly, it requires that a large amount of experience and modeling wisdom be accumulated around this new way of specifying executable behavior. Still, we see no reason why this ambitious possibility should not be considered as it is now. Scenario-based behavior is

what people use when they think about their systems, and our work shows that it is possible to capture a rich spectrum of such behavior conveniently, and to execute it directly, resulting in a runnable artifact that is as powerful as an intra-object model. From the point of view of the user, executing such behavior looks no different from executing any system model. Moreover, it is hard to underestimate the advantages of having the behavior structured according to the way the engineers invent and design it and the users comprehend it (for example, in the testing, maintenance and modifications stages, in sharing the specification process with less technically oriented people, etc.).

In any case, the book concentrates on describing and illustrating the ideas and technicalities themselves, and not on trying to convince the reader of this or that usage thereof. How, in what role, and to what extent these ideas will indeed become useful are things that remain to be seen.

### 1.3 What's in the Book?

Besides this brief introductory chapter, Part I of the book, the Prelude, contains a chapter providing the background and context for the rest of the book, followed by a high-level overview of the entire approach, from which the reader can get a pretty good idea of what we are doing.

Part II, Foundations, describes the underlying basics of the object model, the LSCs language and the Play-Engine tool.

Parts III, IV and V treat in more detail the constructs of the enriched language of LSCs, and the way they are played in and played out. Almost every chapter in these three parts contains a section named “And a Bit More Formally ...”, which provides the syntax and operational semantics for the constructs described in the chapter. As we progress from chapter to chapter, we use a blue/black type convention to highlight the additions to, and modifications of, this formal description. (Appendix A contains the fully accumulated syntax and semantics.)

Part VI describes extensions and enhancements, with chapters on the innards of the Play-Engine tool, particularly the play-out algorithms, on the GUI editor we have built to support the construction of application GUIs, on the smart play-out module, which uses formal verification techniques to drive parts of the execution, and on future research and development directions.

Part VII contains several technical appendices, one of which is the full formal definition of the enriched LSCs language.



<http://www.springer.com/978-3-540-00787-6>

Come, Let's Play  
Scenario-Based Programming Using LSCs and the  
Play-Engine

Harel, D.; Marelly, R.

2003, XVIII, 382 p., Hardcover

ISBN: 978-3-540-00787-6