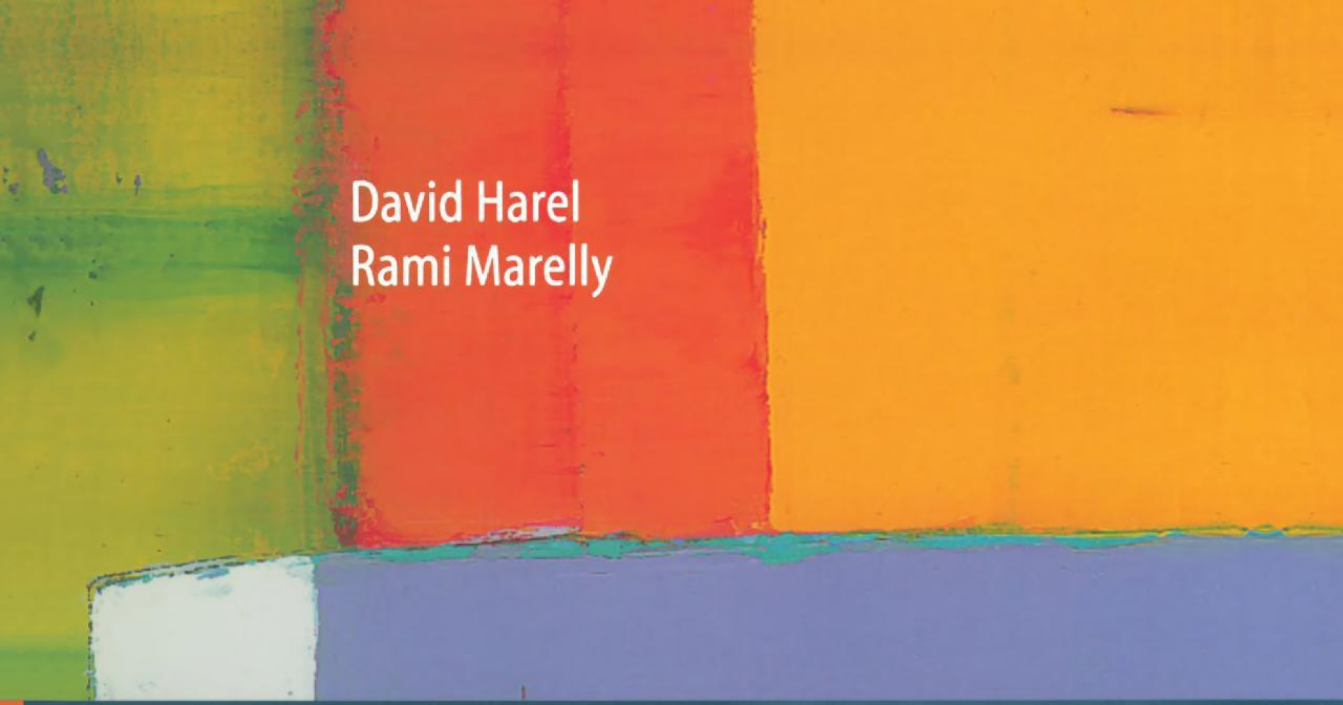

David Harel
Rami Marelly

Come, Let's Play

Scenario-Based Programming Using LSCs and the Play-Engine

Springer

David Harel Rami Marelly

Come, Let's Play:

Scenario-Based Programming Using LSCs and the Play-Engine

With 185 Figures and CD-ROM

Springer

David Harel
Rami Marelly

The Weizmann Institute of Science
Faculty of Mathematics and Computer Science
Rehovot 76100, Israel

Library of Congress Cataloging-in-Publication Data
Harel, David, 1950–
Come, let's play: scenario-based programming using LSCs and the play-engine/
David Harel, Rami Marelly.
p.cm.
ISBN 978-3-642-62416-2 ISBN 978-3-642-19029-2 (eBook)
DOI 10.1007/978-3-642-19029-2
1. Software engineering. 2. System design. 3. Object-oriented programming (Computer science) 4. Visual programming languages (Computer science) I. Marelly, Rami, 1967– II. Title
QA76.758.H365 2003
005.1–dc21 2003045546

ACM Computing Classification (1998): D.2, C.2.0, B.4.0, D.0, D.1.5, D.3, I.6.5, I.6.8

http://www.springer.de

Originally published by Springer-Verlag Berlin Heidelberg New York in 2003

Cover design: KünkelLopka, Heidelberg
Typesetting: Camera-ready by authors
Printed on acid-free paper 45/3142 GF– 543210

Contents

Part VI. Enhancing the Play-Engine

Part VII. Appendices

Part I

Prelude

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

D. Harel et al., *Come, Let's Play:*

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.

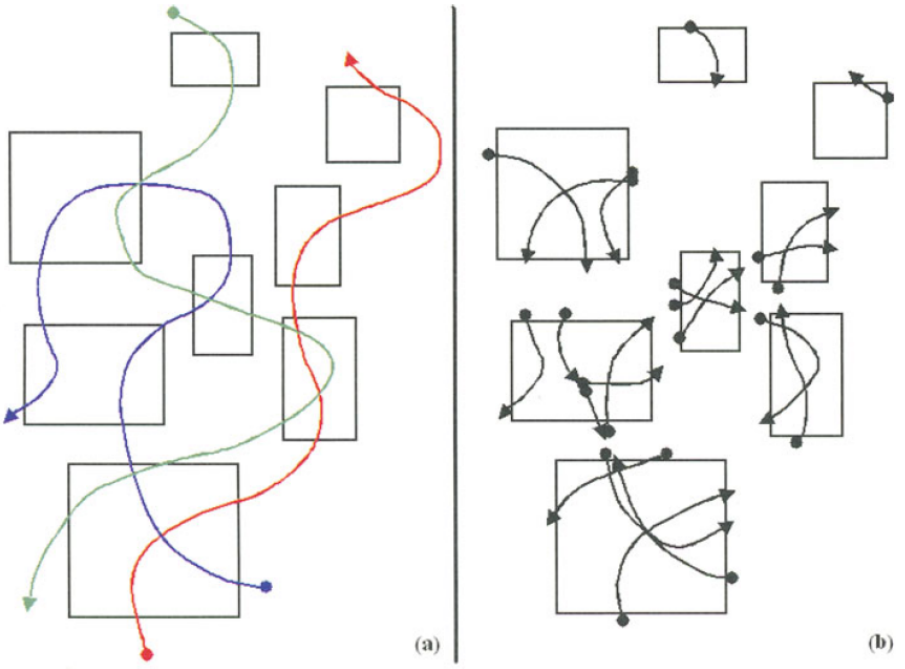

Fig. 1.1. Inter-object vs. intra-object behavior

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.

2. Setting the Stage

In this chapter we set the stage for the rest of the book, by describing some of the main ideas in systems and software engineering research that lead to the material developed later. We discuss visual formalisms and modeling languages, model execution and code generation, the connection between structure and behavior, and the difference between implementable behavior and behavioral requirements. We then go on to describe in somewhat more detail some of the basic concepts we shall be expanding upon, such as the inter-/intra-object dichotomy, MSCs vs. LSCs, the play-in and play-out techniques, and the way all these fit into our global view of the system development process.

2.1 Modeling and Code Generation

Over the years, the main approaches to high-level system modeling have been **structured-analysis and structured-design** (SA/SD), and **object-oriented analysis and design** (OOAD). The two are about a decade apart in initial conception and evolution. Over the years, both approaches have yielded **visual formalisms** for capturing the various parts of a system model, most notably its structure and behavior. A recent book, [120], nicely surveys and discusses some of these approaches.

SA/SD, which started in the late 1970s, is based on raising classic procedural programming concepts to the modeling level and using diagrams for modeling system structure. Structural models are based on **functional decomposition** and the flow of information, and are depicted using hierarchical dataflow diagrams. Many methodologists were instrumental in setting the ground for the SA/SD paradigm, by devising the functional decomposition and dataflow diagram framework, including DeMarco [31], and Constantine and Yourdon [25]. Parnas's work over the years was very influential too.

In the mid-1980s, several methodology teams enriched this basic SA/SD model by providing a way to add state-based behavior to these efforts, using state diagrams or the richer language of **statecharts** (see Harel [42]).

D. Harel et al., *Come, Let's Play:*

These teams were Ward and Mellor [117], Hatley and Pirbhai [54], and the Statemate team [48]. A state diagram or statechart is associated with each function or activity, describing its behavior. Several nontrivial issues had to be worked out to properly connect structure with behavior, enabling the modeler to construct a comprehensive and semantically rigorous model of the system; it is not enough to simply decide on a behavioral language and then associate each function or activity with a behavioral description.[1] The three teams struggled with this issue, and their decisions on how to link structure with behavior ended up being very similar. Careful behavioral modeling and its close linking with system structure are especially crucial for **reactive systems** [52, 93], of which real-time systems are a special case.

The first commercial tool to enable **model execution** and full **code generation** from high-level models was Statemate, built by I-Logix and released in 1987 [48, 60]. (Incidentally, the code generated need not necessarily result in software; it could be code in a hardware description language, leading to hardware.) A detailed summary of the SA/SD languages for structure and behavior, their relationships and the way they are embedded in the Statemate tool appears in [53].

Of course, modelers need not adopt state machines or statecharts to describe behavior. There are many other possible choices, and these can also be linked with the SA/SD functional decomposition. They include such visual formalisms as **Petri nets** [101] or **SDL diagrams** [110], more algebraic ones like **CSP** [59] or **CCS** [88], and ones that are closer in appearance to programming languages, like **Esterel** [14] and **Lustre** [41]. Clearly, if one does not want to use any such high-level formalisms, code in an appropriate conventional programming language could be written directly in order to specify the behavior of a function in an SA/SD decomposition.

The late 1980s saw the first proposals for object-oriented analysis and design (OOAD). Just like in the SA/SD approach, here too the basic idea in modeling system structure was to lift concepts up from the programming level — in this case object-oriented programming — to the modeling level and to use visual formalisms. Inspired by **entity-relationship** (**ER**) **diagrams** [21], several methodology teams recommended various forms of **class diagrams** and **object model diagrams** for modeling system structure [16, 26, 105, 111]. To model behavior, most object-oriented modeling approaches also adopted statecharts [42]. Each class is 'programmed' using a statechart, which then serves to describe the behavior of any instance object of that class; see, e.g., [105, 16, 44].

[1] This would be like saying that when you build a car all you need are the structural things — body, chassis, wheels, etc. — and an engine, and you then merely stick the engine under the hood and you are done.

In the OOAD world, the issue of connecting structure and behavior is subtler and a lot more complicated than in the SA/SD one. Classes represent dynamically changing collections of concrete objects. Behavioral modeling must thus address issues related to object creation and destruction, message delegation, relationship modification and maintenance, aggregation, inheritance, and so on. The links between behavior and structure must be defined in sufficient detail and with enough rigor to support the construction of tools that enable model execution and full code generation. See Fig. 2.1.

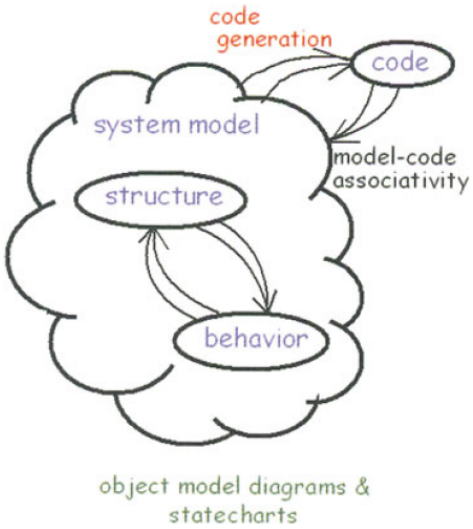

Fig. 2.1. Object-oriented system modeling with code generation

Obviously, if we have the ability to generate full code, we would eventually want that code to serve as the basis for the final implementation. In the OOAD world, a few tools have been able to do this. One is Rhapsody, also from I-Logix [60], which is based on the work of Harel and Gery in [44] on executable object modeling with statecharts. Another is ObjectTime, which is based on the ROOM method of Selic et al. [111], and is now part of the Rose RealTime tool from Rational [100]. There is no doubt that techniques for this kind of 'super-compilation' from high-level visual formalisms down to programming languages will improve in time. Providing higher levels of abstraction with automated downward transformations has always been the way to go, as long as the abstractions are ones with which the engineers who do the actual work are happy.

In 1997, the Object Management Group (OMG) adopted as a standard the **unified modeling language** (UML), put together by a large team led by Booch, Rumbaugh and Jacobson; see [115, 106]. The class/object diagrams, adapted from the Booch method [16] and the OMT (object modeling technique) method [105], and driven by statecharts for behavior [44], constitute that part of the UML that specifies unambiguous, executable (and therefore implementable) models. It has been termed XUML, for **executable UML**. The UML also has several means for specifying more elaborate aspects of system structure and architecture (for example, packages and components). Large amounts of further information on the UML can be found in OMG's website [115].

2.2 Requirements

So much for modeling systems in the SA/SD and OO worlds. However, the importance of executable models lies not only in their ability to help lead to a final implementation, but also in testing and debugging, the basis of which are the **requirements**. These constitute the constraints, desires, dreams and hopes we entertain concerning the behavior of the system under development. We want to make sure, both during development and when we feel development is over, that the system does, or will do, what we intend or hope for it to do.

Requirements can be formal (rigorously and precisely defined) or informal (written, say, in natural language or pseudocode). An interesting way to describe high-level behavioral requirements is the idea of **use cases**; see Jacobson [62]. A use case is an informal description of a collection of possible scenarios involving the system under discussion and its external actors. Use cases describe the observable reactions of a system to events triggered by its users. Usually, the description of a use case is divided into the main, most frequently used scenario, and exceptional scenarios that give rise to less central behaviors branching out from the main one (e.g., possible errors, cancelling an operation before completion, etc.). However, since use cases are high-level and informal by nature, they cannot serve as the basis for formal testing and verification. To support a more complete and rigorous development cycle, use cases must be translated into fully detailed requirements written in some formal language.

Ever since the early days of high-level programming, computer science researchers have grappled with requirements; namely, with how to best state what we want of a complex program or system. Notable efforts are those embodied in the classic Floyd/Hoare **inductive assertions** method, which uses

invariants, pre- and post-conditions and termination statements [12], and in the many variants of **temporal logic** [82]. These make it possible to express different kinds of requirements that are of interest in reactive systems. They include **safety constraints**, which state that bad things will not happen; for example, this program will never terminate with the wrong answer, or this elevator door will never open between floors. They also include **liveness constraints**, which state that good things must happen. For example, this program will eventually terminate, or this elevator will open its door on the desired floor within the allotted time limit.

A more recent way to specify requirements, which is popular in the realm of object-oriented systems, is to use **message sequence charts** (**MSCs**), which are used to specify scenarios as sequences of message interactions between object instances. This visual language was adopted as a standard long ago by the International Telecommunication Union (the ITU; formerly the CCITT) [123], and it also manifests itself in the UML as the language of **sequence diagrams** (see [115]). MSCs combine nicely with use cases, since they can specify the scenarios that instantiate the use cases. Sequence charts thus capture the desired interrelationships between the processes, tasks, components or object instances — and between them and the environment — in a way that is linear or quasilinear in time.[2] In other words, the modeler uses MSCs to formally visualize the actual scenarios that the more abstract and generic use cases were intended to denote.

Objects in MSCs are represented by vertical lines, and messages between these instances are represented by horizontal (or sometimes down-slanted) arrows. Conditional guards, showing up as elongated hexagons, specify statements that are to be true when reached. The overall effect of such a chart is to specify a scenario of behavior, consisting of messages flowing between objects and things having to be true along the way.

Figure 2.2 shows a simple example of an MSC for the quick-dial feature of a cellular telephone. The sequence of messages it depicts consists of the following: the user clicks the $*$ key, and then clicks a digit on the *Keyboard*, followed by the *Send Key*, which sends a *Sent* indication to the internal *Chip*. The *Chip*, in turn, sends the digit to the *Memory* to retrieve the telephone number associated with the clicked digit, and then sends out the number to the external *Environment* to carry out a call. A signal is then received from the environment, guarded by a condition asserting that it is not a busy signal.

[2] Tasks, processes and components are mentioned here too, since although the book is couched in the terminology of object-orientation, many of the ideas apply also to other ways of structuring systems.

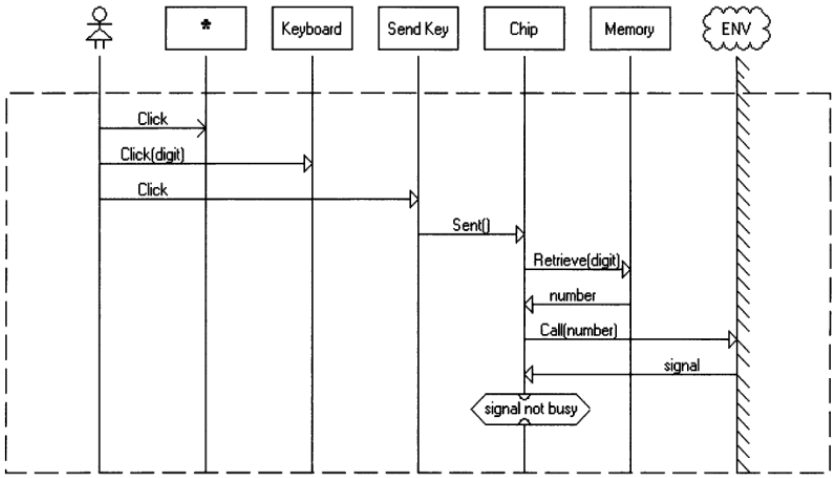

Fig. 2.2. A message sequence chart (MSC)

2.3 Inter-Object vs. Intra-Object Behavior

The style of behavior captured by sequence charts is inter-object, to be contrasted with the intra-object style of statecharts. Whereas a sequence chart captures what goes on in a scenario of behavior that takes place between and amongst the objects, a statechart captures the full behavioral specification for one of those objects (or tasks or processes). Statecharts thus provide details of an object's behavior under all possible conditions and in all the possible 'stories' described previously in the inter-object sequence charts.

Two points must now be made regarding sequence charts. The first is one of exposition: by and large, the subtle difference in the roles of sequence-based languages for behavior and component-based ones is not made clear in the literature. Again and again, one comes across articles and books (many of them related to UML) in which the very same phrases are used to introduce sequence diagrams and statecharts. At one point such a publication might say that "sequence diagrams can be used to specify behavior", and later it might say that "statecharts can be used to specify behavior". Sadly, the reader is told nothing about the fundamental difference in nature and usage between the two — that one is a medium for conveying requirements, i.e., the inter-object behavior required of a model, and the other is part of the executable model itself. This obscurity is one of the reasons many naive readers come away confused by the multitude of diagram types in the full UML standard and the lack of clear recommendations about what it means to specify the behavior of a system in a way that can be implemented and executed.

The second point is more substantial. As a requirements language, the many variants of MSCs, including the ITU standard [124] and the sequence diagrams adopted in the UML [115], as well as versions enriched with timing constraints and co-regions, and the **high-level MSCs** that make it possible to combine charts using the power of regular expressions, have very limited expressive power. Their semantics is intended to support the specification of possible scenarios of system behavior, and is therefore usually given by a set of simple constraints on the partial order of possible events in a system execution: along a vertical object line higher events precede lower ones, and the sending of a message precedes its receipt.[3] Virtually nothing can be said in such diagrams about what the system will actually do when run. They can state what might *possibly* occur, not what *must* occur. In the chart of Fig. 2.2, for example, there is nothing to indicate whether some parts of the scenario are mandatory. For example, can the *Memory* 'decide' not to send back a number in response to the request from the *Chip*? Does the guarding condition stating that the signal is not busy really *have* to be true? What happens if it is not? If one wants to be puristic, then, under most definitions of the semantics of message sequence charts, an empty system -- one that doesn't do anything in response to anything — satisfies such a chart. Hence, just sitting back and doing nothing will make your requirements happy. (Usually, however, there is a minimal, often implicit, requirement that each one of the specified sequence charts should have at least one run of the system that winds its way correctly through it.)

MSCs can be used to specify expected scenarios of behavior in the requirements stage, and can be used as test scenarios that will be later checked against the executing behavior of the final system. However, they are not enough if we want to specify the actual behavior of a reactive system in a scenario-based fashion. We would like to be able to say what may happen and what must happen, and also what is not allowed to happen. The latter gives rise to what we call **anti-scenarios**, in the sense that if they occur something is very wrong: either something in the specification is not as we wanted, or else the implementation does not correctly satisfy the specification. We would like to be able to specify **multiple scenarios** that combine with each other, or even with themselves, in subtle ways. We want to be able to specify **generic scenarios**, i.e., ones that stand for many specific scenarios, in that they can be instantiated by different objects of the same class. We want variables and means for specifying real time, and so on.

[3] There can also be **synchronous** messages, for which the two events are simultaneous.

leads to the final software or hardware, and will consist of the complete behavior coded for each object. In contrast, it is common to assume that the left-hand side, the set of requirements, is not implementable or executable. A collection of scenarios cannot be considered an implementable model of the system: How would such a system operate? What would it do under general dynamic circumstances? How would we decide what scenarios would be relevant when some event suddenly occurs out of the blue? How should we deal with the mandatory, the possible and the forbidden, during execution? And how would we know what subsequent behaviors these and other modalities of behavior might entail?

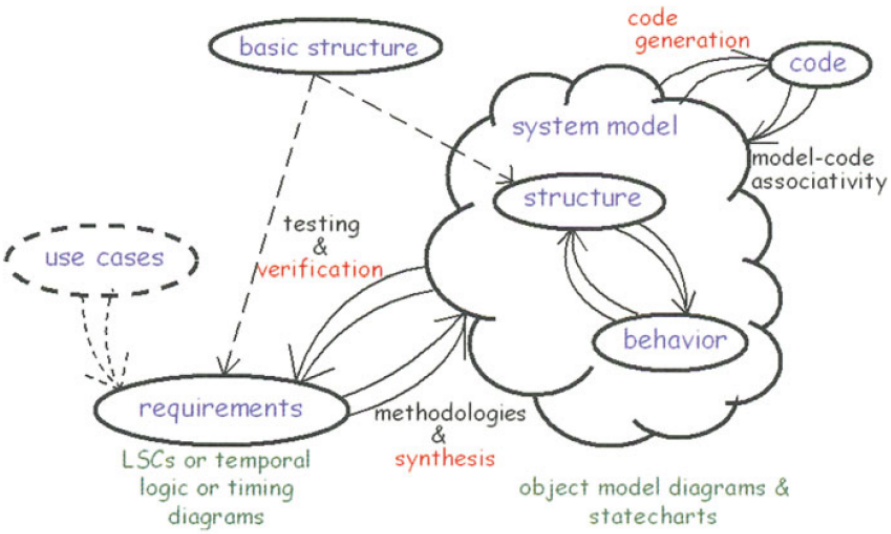

Fig. 2.4. Conventional system development

One of the main messages of this book is that this assumption is no longer valid. Scenario-based behavior need not be limited to requirements that will be specified before the real executable system is built and will then be used merely to test that system. Scenario-based behavior, we claim, can actually be executed. Furthermore, we predict that in many cases such behavior will become the implemented system itself. This will be illustrated and discussed in detail as the book progresses.

For now, let us discuss the relations and transitions between the different parts of the conventional setup of system development, as shown in Fig. 2.4. The arrow between the use cases and the requirements is dashed for a reason: it does not represent a 'hard' computerized process. Going from use cases to

formal requirements is a 'soft' methodological process performed manually by system designers and engineers. It is considered an art or a craft and requires a good understanding of the target formal requirements language and a large amount of creativity.

The arrow going from the system model to the requirements depicts testing and verifying the model against the requirements. Here is a nice way to do testing using an automated tool.[6] Assume the user has specified the requirements as a set of sequence diagrams, perhaps instantiating previously prepared use cases. For simplicity, let us say that this results in a diagram called A. Later, when the executable intra-object system model has been specified, the user can execute it and ask that during execution the system should automatically construct an animated sequence diagram, call it B, on the fly. This diagram will show the dynamics of object interaction as it actually happens during execution. When this execution is completed, the tool can be asked to compare diagrams A and B, and to highlight any inconsistencies, such as contradictions in the partial order of events, or events appearing in one diagram but not in the other. In this way, the tool helps debug the behavior of the system against the requirements.

A recently developed tool, called TestConductor, which is integrated into Rhapsody [60], enables a richer kind of testing using a subset of LSCs. The test scenarios can describe scenarios of interaction between the environment and the system under development. The tool then runs the tests, and simulates the behavior of the environment by monitoring the test scenarios and sending messages to the system on behalf of the environment, when required. The tool determines the results of such a test by comparing the sequence diagrams produced by the system with those that describe the tests using visual comparison, as described above.

Note that even these powerful ways to check the behavior of a system model against our expectations are limited to those executions that we actually carry out. They thus suffer from the same drawbacks as classic testing and debugging. Since a system can have an infinite number of runs, some will always go unchecked, and it could be those that violate the requirements (in our case, by being inconsistent with diagram A). As Dijkstra famously put it years ago, "testing and debugging cannot be used to demonstrate the absence of errors, only their presence".

One remedy is to use true verification. This is not what CASE-tool people in the 1980s often called "validation and verification", which amounted to little more than checking the consistency of the model's syntax. What we have in mind is a mathematically rigorous and precise proof that the model

[6] Rhapsody supports this technique.

satisfies the requirements, and we want this to be done automatically by a computerized verifier. Since we would like to use highly expressive languages like LSCs (or the analogous temporal logics [82] or timing diagrams [108]) for requirements, this means far more than just executing the system model and making sure that the sequence diagrams you get from the run are consistent with those you prepared in advance. It means making sure, for example, that the things an LSC says are not allowed to happen (the anti-scenarios) will indeed never happen, and the things it says must happen (or must happen within certain time constraints) will indeed happen. These are facts that, in general, no amount of execution can fully verify.

Although general verification is a non-computable algorithmic problem, and for finite-state systems it is computationally intractable, the idea of rigorously verifying programs and systems — hardware and software — has come a long way since the pioneering work on inductive assertions in the late 1960s and the later work on temporal logic and model checking. These days we can safely say that true verification can be carried out in many, many cases, even in the slippery and complex realm of reactive real-time systems.

So much for the arrow denoting checking the model against the requirements. In the opposite direction, the transition from the requirements to a model is also a long-studied issue. Many system development methodologies provide guidelines, heuristics, and sometimes carefully worked-out step-by-step processes for this. However, as good and useful as these processes are, they are 'soft' methodological recommendations on how to proceed, not rigorous and automated methods. Here too, there is a 'hard', computerized way to go: Instead of guiding system developers in informal ways to build models according to their dreams and hopes, the idea is to automatically synthesize an implementation model directly from those dreams and hopes, if they are indeed implementable. (For the sake of the discussion, we assume that the structure — for example, the division into objects or components and their relationships — has already been determined.) This is a whole lot harder than generating code from a system model, which is really but a high-level kind of compilation. The duality between the inter-object scenario-based style (requirements) and the intra-object state-based style (modeling) in saying what a system does over time renders the synthesis of an implementable model from the requirements a truly formidable task. It is not too hard to do this for the weak MSCs, which can't say much about what we really want the system to do. It is a lot more difficult for far more realistic requirements languages, such as LSCs or temporal logic.

How can we synthesize a good first approximation of the statecharts from the LSCs? Several researchers have addressed such issues in the past, resulting in work on certain kinds of synthesis from temporal logic [98] and timing

diagrams [108]. In [46], there is a first-cut attempt at algorithms for synthesizing state machines and statecharts from simple LSCs. The technique therein involves first determining whether the requirements are consistent (i.e., whether there exists any system model satisfying them), then proving that being consistent and having a model (being implementable) are equivalent notions, and then using the proof of consistency to synthesize an actual model. The process just outlined yields unacceptably large models in the worst case, so that the problem cannot yet be said to have been solved satisfactorily. We do believe, however, that synthesis will eventually end up like verification — hard in principle but not beyond a practical and useful solution in practice. This is the reason for the solid arrow in Fig. 2.4.

2.6 The Play-In/Play-Out Approach

To complete a full rigorous system development cycle we need to bridge the gap between use cases and the more formal languages used to describe the different scenarios. How should the more expressive requirements themselves be specified? One cannot hope to have a general technique for synthesizing LSCs or temporal logic from the use cases automatically, since use cases are informal and high level. This leaves us with having to construct the LSCs manually. Now, LSCs constitute a formal (albeit, visual) language, and constructing them requires the skill of working in an abstract environment, and detailed knowledge of the syntax and semantics of the language. In a world in which we would like as much automation as possible we would like to make this process more convenient and natural, and accessible to a wider spectrum of people.

This problem was addressed towards the end of [43], and a higher-level approach to the problem of specifying scenario-based behavior, termed **play-in scenarios**, was proposed and briefly sketched. The methodology, supported by a tool called the **Play-Engine** was presented in more detail by the present authors in [49]. The main idea of the play-in process is to raise the level of abstraction in requirements engineering, and to work with a look-alike version of the system under development. This enables people who are unfamiliar with LSCs, or who do not want to work with such formal languages directly, to specify the behavioral requirements of systems using a high-level, intuitive and user-friendly mechanism. These could include domain experts, application engineers, requirements engineers, and even potential end-users.

What 'play-in' means is that the system's developer (we will often call him/her a **user** — not to be confused with the eventual end-users of the system under development, which are sometimes called **actors** in the litera-

ture) first builds the GUI of the system, with no behavior built into it, with only the basic methods supported by each GUI object. This is given to the Play-Engine. In systems for which there is a meaning to the layout of hidden objects (e.g., a board of an electrical system), the user may build the graphical representation of these objects as well. In fact, for GUI-less systems, or for sets of internal objects, we simply use the object model diagram as a GUI. In any case, the user then 'plays' the incoming events on the GUI, by clicking buttons, rotating knobs and sending messages (calling functions) to hidden objects, in an intuitive drag & drop manner. (With an object model diagram as the interface, the user clicks the objects and/or the methods and the parameters.) By similarly playing the GUI, often using right-clicks, the user then describes the desired reactions of the system and the conditions that may or must hold. As this is being done, the Play-Engine does essentially two things continuously: it instructs the GUI to show its current status using the graphical features built into it, and it constructs the corresponding LSCs automatically. The engine queries the application GUI (that was built by the user) for its structure and methods, and interacts with it, thus manipulating the information entered by the user and building and exhibiting the appropriate formal version of the behavior. So much for play-in.

After playing in (a part of) the behavior, the natural thing to do is to make sure that it reflects what the user intended to say. Instead of doing this the conventional way, by building an intra-object model, or prototype implementation, and using model execution to test it, we would like to test the inter-object behavior directly. Accordingly, we extend the power of our GUI-intensive play methodology, to make it possible not only to specify and capture the required behavior but to test and validate it as well. And here is where our complementary **play-out** mechanism enters.

In play-out, which was first described in [49], the user simply plays the GUI application as he/she would have done when executing a system model, or the final system, limiting him-/herself to end-user and external environment actions. As this is going on, the Play-Engine keeps track of the actions and causes other actions and events to occur as dictated by the universal charts in the specification. Here too, the engine interacts with the GUI application and uses it to reflect the system state at any given moment. This process of the user operating the GUI application and the Play-Engine causing it to react according to the specification has the effect of working with an executable model, but with no intra-object model having to be built or synthesized.

Figure 2.5 shows an enhanced development cycle, which includes the play-in/play-out methodology inserted in the appropriate place.

We should emphasize that the behavior played out need not be merely the scenarios that were played in. The user is not just tracing previously

3. An Example-Driven Overview

In this chapter, we overview the main ideas and principles of our work. The purpose of the overview is to give a broad, though very high-level, view of the LSC language, the play-in methodology for specifying inter-object scenario-based behavior, and the play-out mechanism for executing such behavior. We will touch upon many issues, but will not dwell on the details of the language constructs, nor the methodology, nor the tool. The overview is presented as a guided walk-through, using a simple example of a reactive system.

3.1 The Sample System

Consider a bakery, in which different kinds of bread, cakes and cookies are baked in three ovens. Suppose Ms. B., the owner of the bakery, wants to automate the bakery by adding a bakery panel that will control and monitor the three ovens. According to the **play-in** approach, the first thing to do is to ask our user, Ms. B, to describe the desired panel. In this preliminary phase, a very high-level description, focusing on the panel's **graphical user interface** (GUI), is sufficient. The panel, coded using some rapid development language (or a special-purpose tool, as we discuss later) is shown in Fig. 3.1. The panel

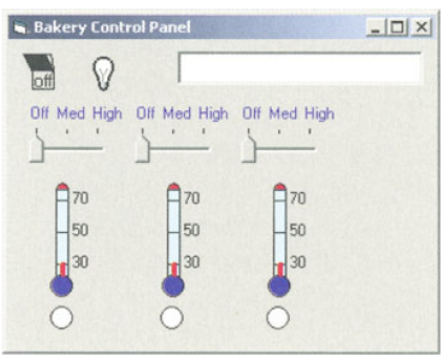

Fig. 3.1. A central panel controlling the bakery's ovens

has a main switch and a main light in its top-left corner. On the top right, there is a console display, which is used to show textual messages. The rest of the GUI contains three 3-state switches, three thermometers and three warning lights. Each set of switch, thermometer and light is used to control and monitor a different oven.

Note that this bakery panel is nothing but a graphical user interface. No behavior is programmed into it, and all it can do is interact with the Play-Engine tool in a rather trivial way. All the behavioral requirements of this panel will be defined as we go along. As we progress with the example, we may add more graphical elements to the panel and define their behavior as well.

3.2 Playing In

Having the GUI application at hand, Ms. B. is ready to specify the required behavior of the bakery panel. She wants to add a new LSC and give it a name. Figure 3.2 shows the Play-Engine with the empty LSC just added. The top blue dashed hexagon is the LSC's prechart and the bottom solid rectangle is its main chart. The prechart should contain a scenario, which, if satisfied, forces the satisfaction of the scenario given in the main chart. The relation between the prechart and the chart body can be viewed as an action-reaction; if and when the scenario in the prechart occurs, the system is obligated to satisfy the scenario in the main chart.

The first thing our user would like to specify is what happens when the bakery panel is turned on. Since this is done using a switch, the action of clicking the switch is put in the prechart, and the appropriate system reactions are put in the chart body. In our case, we want the system, as a response, to turn on the light and to change the display's color to green.

The process of specifying this behavior is very simple. First, the user clicks the switch on the GUI, thus changing its state[1] from *Off* to *On*. When the Play-Engine is notified of this event, it adds the appropriate message in the (initially empty) prechart of the LSC from the *user* instance to the *main switch* instance. See Fig. 3.3.

The user then moves the cursor (a dashed purple line) into the chart body and right-clicks the light on the GUI. The engine knows the properties of the light (in this case, there is just one) and pops up a menu, from which the user chooses the `State` property and sets it to *On*. Figure 3.4 shows the popup menu that is opened after the light is right-clicked, and the dialog that opens

[1] We use the word 'state' to describe a property of the switch. This should not to be confused with the term 'state' from finite state machines and statecharts.

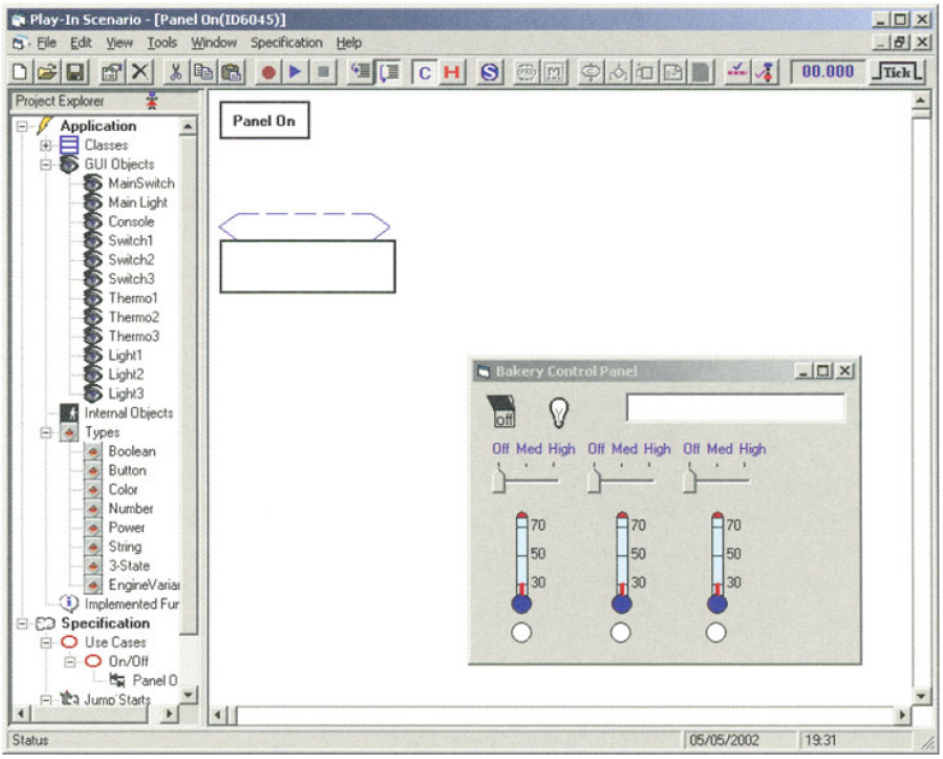

Fig. 3.2. An empty universal LSC in the Play-Engine

after the `State` property is chosen. A similar process is then carried out for the *background* property of the display. After each of these actions, the engine adds a self-message in the LSC from the instance representing the selected object, showing the change in the property. The Play-Engine also sends a message to the GUI application, telling it to change the object's property in the GUI itself so that it reflects the correct value after the actions have been taken. Thus, when this stage is finished, the GUI shows the switch on, the light on, and the display colored green. Figure 3.5 shows the resulting LSC and the status of the GUI panel.

Suppose now that the user wishes to specify what happens when the switch is turned off. In this case we want the light to turn off and the display to change its color to white and erase any displayed characters. The user may, of course, play in another scenario for this, but these two scenarios will be very similar, and they are better represented in a single LSC. This can be done using symbolic messages. We play a scenario as before, with the switch being clicked as part of the prechart, and the system's reactions being played in as the chart's body. However, this time we do it with the *symbolic* flag on.

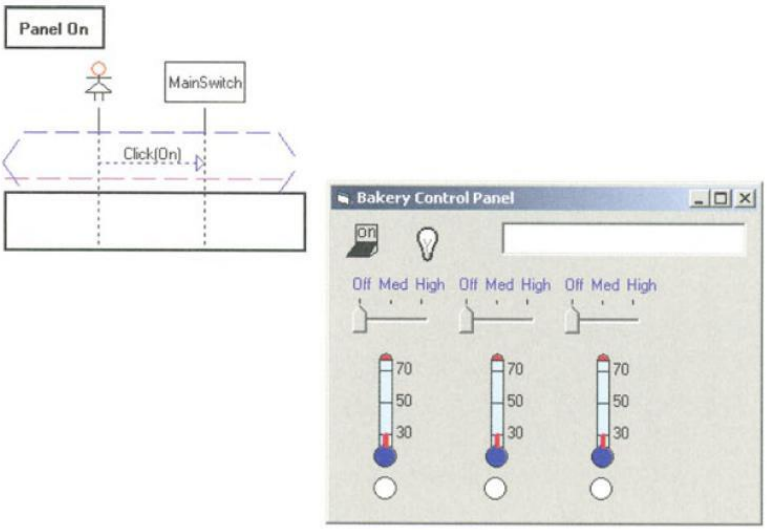

Fig. 3.3. The results of clicking the main switch to *On*

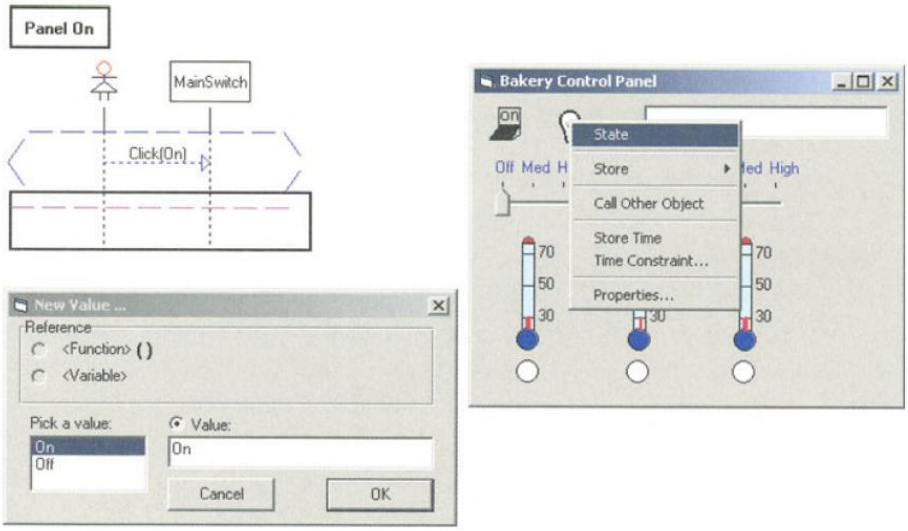

Fig. 3.4. Changing the light state to *On*

When in symbolic mode, the values shown in the labels of messages are the names of variables (or functions), rather than actual values. So the user will now not say that the light should turn on or off as a result of the prechart, but that it should take on the same state as the switch did in the prechart. The Play-Engine provides a number of ways of doing this. A variable can be selected from a table of predefined variables or, as shown in Fig. 3.6, we

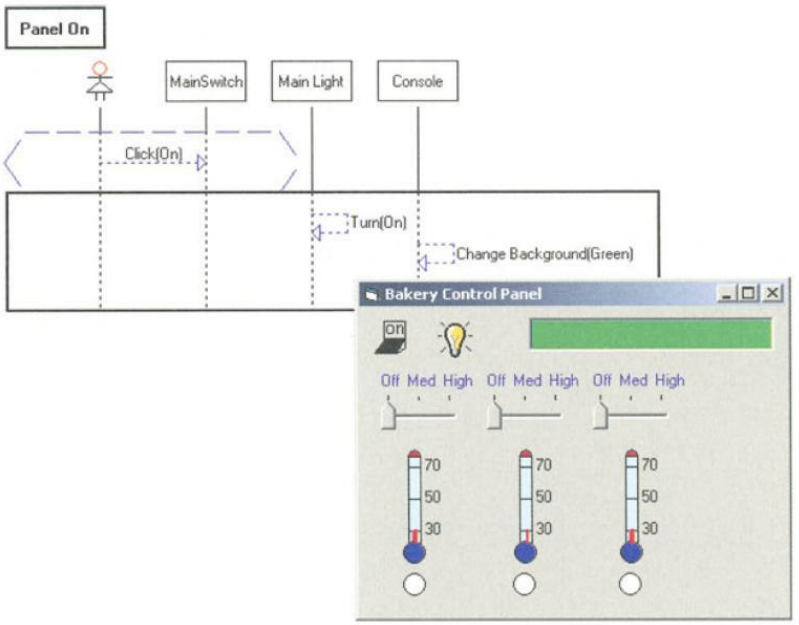

Fig. 3.5. LSC: Turning on the panel

can indicate that the value should be the same as in some message in the LSC. Here X_s is a variable. For the second option, the user simply clicks the

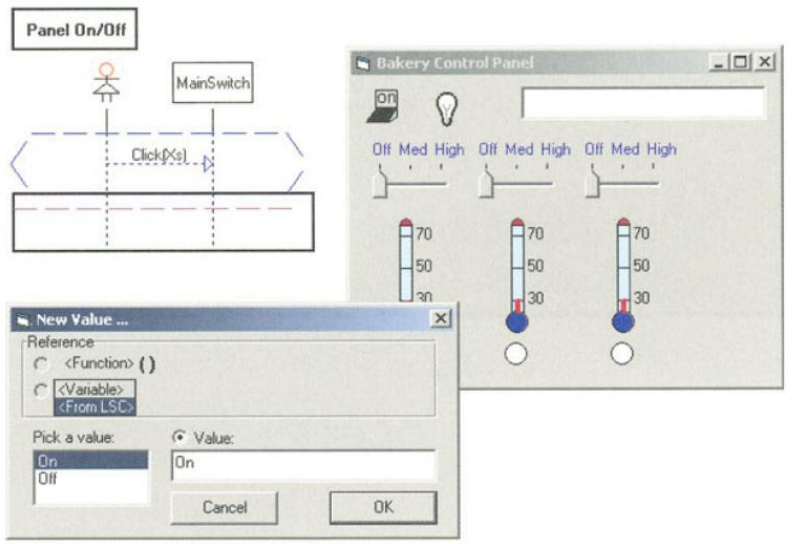

Fig. 3.6. Symbolic mode: the light takes on the same state as the switch

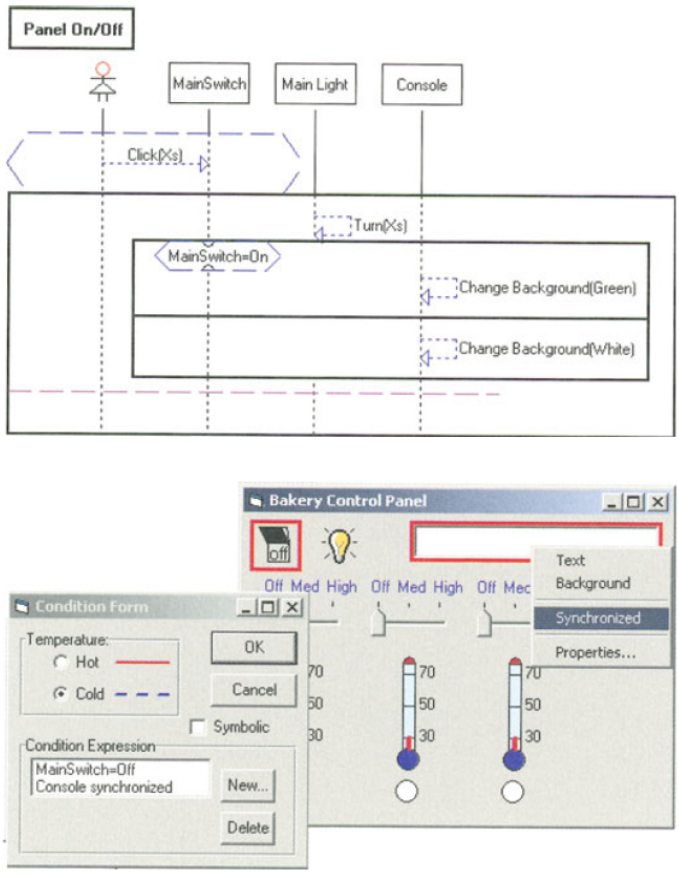

Fig. 3.9. Specifying a stand-alone condition guard

A condition hexagon will be stretched along the LSC, reaching all the instances to which it refers. To distinguish such instances from those that do not participate in the condition's definition or are not to be synchronized with it, the engine draws small semicircular connectors at the intersection points of the condition with the participating instance line. Figure 3.10 shows the final LSC and the way conditions are rendered.

One aspect of the LSC language that contributes to its flexibility is the fact that behaviors can be given in separate charts, and these can each describe a fragment of a more complex behavior. When these fragments become relevant during execution is a consequence of their precharts, and thus no explicit order is imposed on the charts (in contrast to the mechanism for combining charts in high-level MSCs, for example). So, suppose that our user has just decided that when the switch is turned on, the three warning lights should flicker (by changing colors from green to red and back) three times, termi-

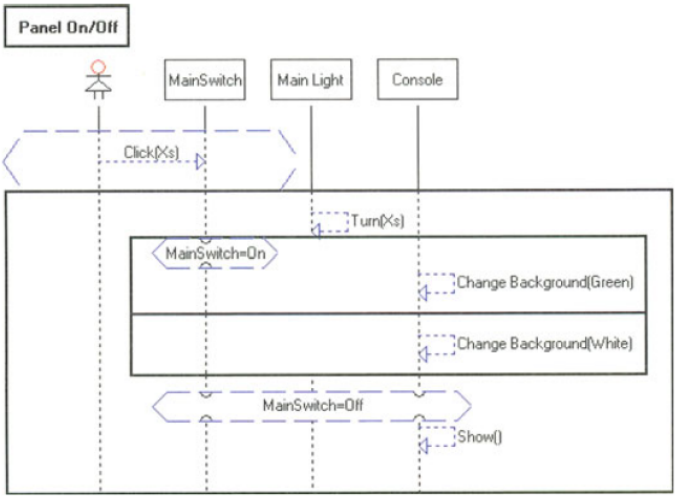

Fig. 3.10. A symbolic LSC for turning the panel on or off

nating in green. One way to capture that is to go back to the LSC shown before and add this 'piece' of behavior in its correct place (i.e., in the 'then' part of the if-then-else construct). Alternatively, we can create another LSC, which is activated only when the switch is turned *On*.

To do this, the user clicks the switch on the GUI to *On* while the cursor is in the prechart, in the same way as described earlier. We now want to specify the three-fold flickering itself. Of course, we could simply play in the six color changes for every light. It is better, however, to use the loop construct of LSCs. As with the if-then-else construct, a loop is inserted by clicking a button on the toolbar, which causes a wizard to open. Figure 3.11 shows the wizard and the LSC during the process of specifying a loop.

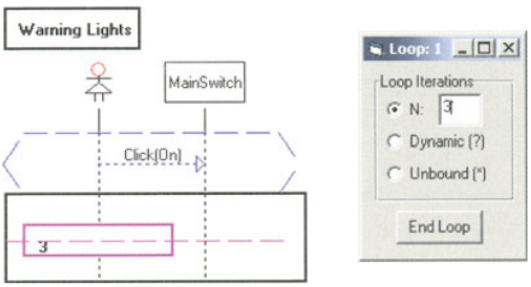

Fig. 3.11. Specifying loops in LSCs

There are three types of loops in LSCs. This one is a **fixed** loop; it is annotated by a number or a variable name, and is performed a fixed number of times.

After selecting the desired type of loop using the loop wizard, the user continues playing in the required behavior in the same way as before. The loop is ended by clicking the `End Loop` button in the loop wizard. Figure 3.12 shows the resulting LSC. Note the special use of a cold condition in

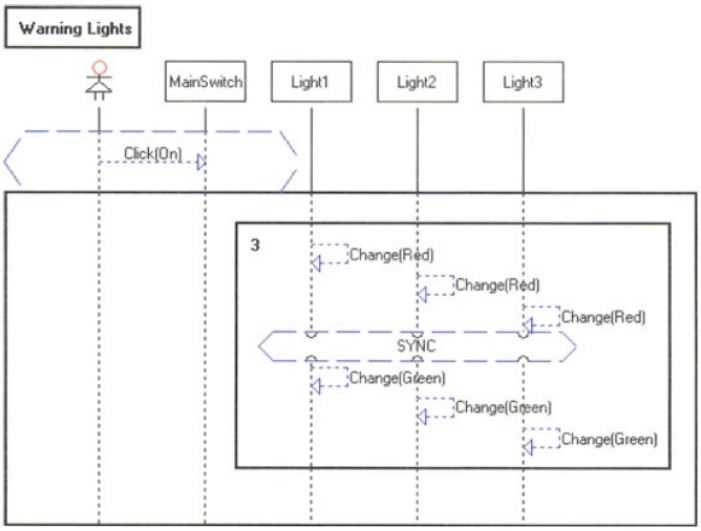

Fig. 3.12. An LSC with a fixed loop

the middle of the loop. A condition with the reserved word *SYNC* is always evaluated to *true*. (Actually, the reserved word *TRUE* could have been used instead, but SYNC reflects better the underlying intuition.) Placing such a condition where it is, and synchronizing the three lights with it has the effect of synchronizing the lights and forbidding a light to change its color to *green* before the others have changed their color to *red*.

Thinking a bit more about the bakery panel, Ms. B. now decides that she would like to be able to probe the thermometers for their exact temperature. To do that, some additional graphical objects should be added to the panel. Figure 3.13 shows the modified panel. Two selectors and one *Probe* button have been added. One selector is used to select a thermometer and the other to select the units for displaying the temperature (i.e., Celsius or Fahrenheit).

We now specify the following requirement:

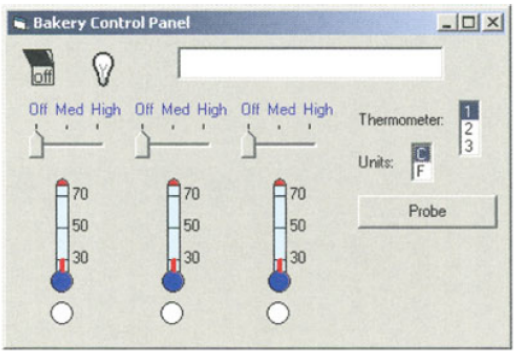

Fig. 3.13. The modified bakery panel

When the Probe button is clicked, the console should display the temperature of the thermometer selected using the thermometer selector. The temperature should be displayed in Celsius or Fahrenheit degrees, according to the units in the units selector.

The prechart is quite simple, and contains a single message denoting the event of the user clicking the *Probe* button. In the chart body, we first want to store the temperature of the selected thermometer. We do this using three if-then constructs. The temperature is stored in a variable using an assignment construct. Assignments are internal to a chart, and can be used to save values of the properties of objects, or of functions applied to variables holding such values. The assigned-to variable stores the value for later use in the LSC. The expression on the right-hand side contains either a reference to a property of some object (this is the typical usage) or a function applied to some predefined variables. It is important to note that the assignment's variable is local to the containing chart and can be used in that chart only, as opposed to the system's state variables, which may be used in several charts. Figure 3.14 shows how an object property can be stored, by right-clicking the desired object, choosing `Store` and then the desired property name. It also shows the resulting assignment, drawn as a rectangle folded in its top-right corner. Since after storing a value we might like to refer to it later, and for this a meaningful name is helpful, the Play-Engine lets the user name the assigned variable; here we use *Tc* (since the original temperature is given in Celsius). Figure 3.15 shows an LSC in which the variable *Tc* stores the required temperature.

Next, the console should display *Tc* according to the selected units. If Celsius is selected, then *Tc* should be displayed as is. If Fahrenheit is selected, the temperature should be converted. This is an example of the need to use data manipulation algorithms and functions that are applied to specified variables. Such functions cannot (and should not) be described using LSC-

the environment reacts with the system in a way similar to the interaction with the end-user.

Figure 3.17 shows an LSC that describes the above requirement for thermometer ♯1. When the temperature of thermometer ♯1 is changed (by the

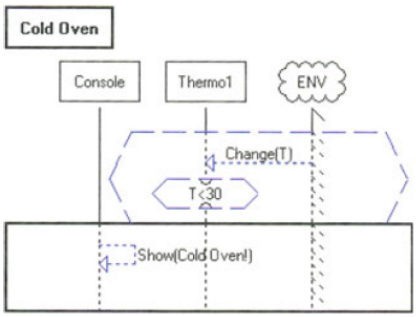

Fig. 3.17. Interacting with the external environment

environment) to some value T, and T is less than 30 degrees Celsius, then the console displays the required warning message.

We could have created two more charts for the other two thermometers, yet there is a more elegant way to do this. Many systems feature multiple objects that are instances of the same class. This is one of the central maxims of the object-oriented paradigm. For example, a communication system contains many phones, a railroad control system may have not only many trains and terminals but also many distributed controllers, etc. We would like to be able to specify behavioral requirements in a general way, on the level of classes and their parameterized instances, not necessarily restricting them to concrete objects. In our example, since the three thermometers are actually three instances of the same *thermometer* class, we would like to take the LSC shown in Fig. 3.17 and generalize it so that it deals with all the thermometers in the system. We can do this using an extension we have defined for LSCs, involving **classes** and **symbolic instances**. A symbolic instance is associated with a class rather than with an object, and may stand for any object that is an instance of the class. Figure 3.18 shows the generalized version of the LSC of Fig. 3.17. In our example, a *CTherm* class was created and all the thermometers were defined as instances thereof. In Fig. 3.18, the instance representing thermometer ♯1 was turned into a symbolic instance, and thus now represents any object that is an instance of class *CTherm*.

Symbolic instances constitute a rather complex topic, raising several interesting and non-trivial issues that we deal with later in the book.