Softw Syst Model
DOI 10.1007/s10270-007-0069-5

SPECIAL SECTION PAPER

Modeling and verification of a telecommunication application using live sequence charts and the Play-Engine tool

Pierre Combes · David Harel · Hillel Kugler

Received: 26 December 2005 / Revised: 9 June 2006 / Accepted: 10 July 2007

Abstract We apply the scenario-based approach to modeling, via the language of live sequence charts (LSCs) and the Play-Engine tool to a real-world complex telecommunication service, `Depannage`. It allows a user to call for help from a doctor, the fire brigade, a car maintenance service, etc. These kinds of services are built on top of an embedded platform, using both new and existing service components, and their complexity stems from their distributed architecture, the various time constraints they entail, and their rapidly evolving underlying systems. A well known problem in this class of telecommunication applications is that of feature interaction, whereby a new feature might cause problems in the execution of existing features. Our approach provides a methodology for high-level modeling of telecommunication applications that can help in detecting feature interaction at early development stages. We exhibit the results of applying the methodology to the specification, animation and formal verification of the Depannage service.

Communicated by Dr. Susanne Graf.

This research was supported by the European Commission project OMEGA (IST-2001-33522) "Correct Development of Real-Time Embedded Systems". During this project, the third author was at the Weizmann Institute of Science and later at New York University. A Preliminary version [11] has appeared in ATVA'05.

P. Combes
France Telecom Research and Development, Paris, France
e-mail: Pierre.Combes@francetelecom.com

D. Harel
The Weizmann Institute of Science, Rehovot, Israel
e-mail: dharel@weizmann.ac.il

H. Kugler (✉)
Microsoft Research, Cambridge, UK
e-mail: hkugler@microsoft.com

Keywords Live sequence charts (LSCs) · Requirements engineering · Verification · Telecommunication

1 Introduction

A scenario-based methodology for specifying and developing complex reactive systems, termed the *play-in/play-out approach*, and a supporting tool called the *Play-Engine*, are described in [24,25]. The method and tool are based on specifying and executing scenario-based requirements in the language of live sequence charts (LSCs) [12], a broad extension of classical message sequence charts [40,44]. LSCs distinguish between behaviors that must happen in the system (universal) from those that may happen (existential), and also allows to express forbidden behavior ("anti-scenarios") and other modalities of behavior. Play-in is an intuitive way to capture LSC specifications by demonstrating a desired behavior (up to a partial order induced equivalence) on a graphical representation of the system, while play-out is a method for directly executing LSC specifications, giving the feeling of working with an actual system.

The scenario-based method, via LSCs and the play-in/play-out approach allows one to specify and develop reactive systems in general, with potential application to many stages in classical system development. A relevant question concerns the specific application domains and development stages for which the approach and tool are especially well suited, and the issue of how well do the general principles work in practice. Our current work attempts (at least partially) to address these questions by applying the scenario-based approach to high level modeling and analysis of telecommunication applications. It has been carried out in the context of the EU project OMEGA [35] (correct development of real-time embedded systems) in which an important goal was

practical evaluation of verification tools by applying them to case studies provided by the industrial partners.

The challenging complexity of telecommunication systems, combined with a high demand for rapid deployment, encourages the development of innovative techniques in order to design and deploy new applications in a quick and secure manner [7]. A telecommunication application is usually built from a set of embedded service components. Nowadays, due to openness of the telecommunication architecture, multiple services and service features are often developed by several teams in parallel, in order to satisfy new customer requirements. Introducing new services always involves a risk of feature interaction, whereby new services might cause problems in the execution of existing ones. This is a critical problem in telecommunication, involving significant loss of time and money during testing and operation phases. This risk can be reduced by identifying the problems during the design and modeling phases.

We introduce an incremental methodology for high level modeling and analysis of telecommunication applications. Using LSCs with the play-in/play-out approach and the Play-Engine tool, we apply our methodology effectively to a concrete example of the nontrivial telecommunication service Depannage, provided by France Telecom. The complete specification of the Depannage application in LSCs and some animations showing simulation and verification results are available online.[1]

2 Modeling paradigm for telecommunication applications

2.1 An example: the Depannage service

The Depannage service allows a user to make a phone call and ask for help from a doctor, the fire brigade, a car maintenance service, etc. The service first authenticates the calling user, and then searches for the calling location. Once the calling location is found, the service searches a database for numbers of potential service providers corresponding to the Depannage society members in the vicinity of the caller. Once a set of numbers is found, the service tries to connect the caller to one of the potential numbers (where numbers are called sequentially or in parallel). In any case, the caller should be connected to a secretary of the Depannage society. In parallel, a second logic will make periodic location requests to the Depannage society members in order to record in the database their latest locations.

The Depannage service, like many other telecommunication applications, is built from a set of existing components deployed in several parts of the telecommunication infrastructure. For the Depannage application these components include the location-based architecture (platform and communication channels specific for location services), the core network platforms, and the terminals. These reused components are defined independently of any embedding application and offer specific service features (or enablers) such as authentication, location, call connection and session supervision.

2.2 Component-based design

In the telecommunication domain, components play a crucial role. The majority of these components is embedded in a large and complex architecture which involves real-time constraints and requirements. Moreover, non-functional requirements, in particular time dependent properties, also play an important role.

The Depannage service is typical to the telecommunication domain, insofar as the great majority of new applications are built and deployed based on the use of existing features or components. It is therefore crucial that any new component is specified and designed in a way that allows reusing the existing platform and its service components.

In our approach, a component specifies a formal contract for the services that it is supposed to provide to external clients (other components of the system) and for the services that it requires from other components. An example of specifying a black box view of the `SearchOnList` component, using a UML component diagram [40] is shown in Fig. 1. The provided interfaces are `CallControlService` and `SearchDataBase`, used in the ports `SearchApi` and `Data` respectively, while the required interface is `SearchOnListService` used in port `SearchService`. The `SearchOnList` component represents a modular part of a system, that encapsulates its contents and abstracts from the implementation details, (this is depicted in Fig. 1 by a box labeled `Abstraction`) and as a result is replaceable

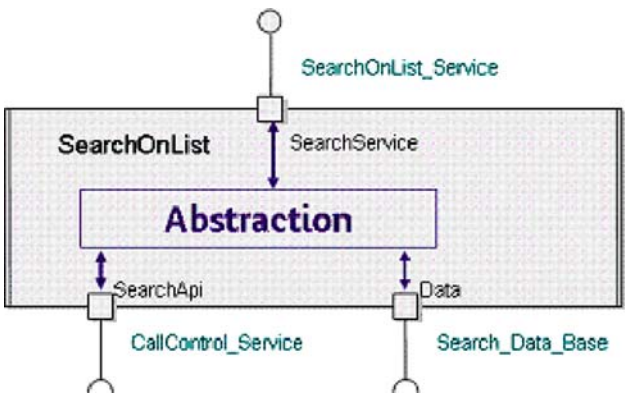


Fig. 1 The component diagram

[1] http://research.microsoft.com/~hkugler/Depannage/

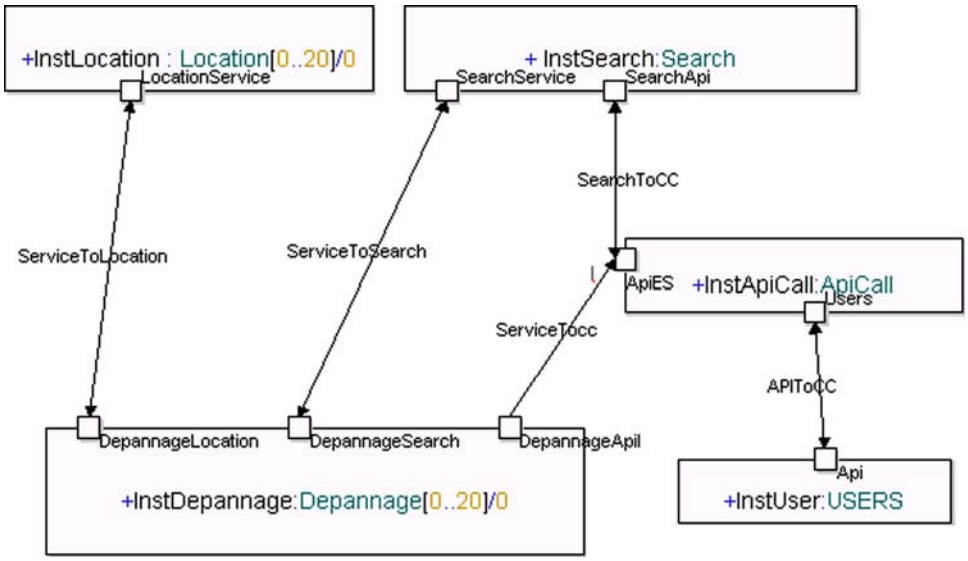


Fig. 2 The architecture of the Depannage application

by different component realizations that satisfy the interface requirements.

In our methodology, a telecommunication service is built by using a set of components—reusable software units specified by their interfaces. The specification of these interfaces is given by the signatures of the required and provided methods (procedures) and signals, and by the description of the dynamic behaviors. A component specification is defined in three views: (1) the structural view: ports, interfaces and signatures; (2) the dynamic behavior expressed in terms of interfaces and ports; and (3) the specification of extra-functional properties such as real-time properties.

In the first view, the structure is defined in terms of provided and required interfaces including signal or procedure call signatures. The provided interface consists of signals and procedure calls that can be sent by the environment, whereas the required interface consists of signals or procedure calls that the component requests to be handled by its environment. Interfaces may optionally be organized through ports. Ports correspond to different communication paths between the component and the external entities. A component may have the same interfaces (sets of signatures) associated with different ports, corresponding to different roles.

The objective of the second view is to specify an external description of the dynamic behavior of the component in terms of the given observables. Such a dynamic view is described by a set of causality relations between provided and required interfaces. For such a dynamic specification, we used LSCs. They describe the temporal order of events needed between ports in order to realize the provided functionalities. Using LSCs for interface description of components is a new approach and plays a central part in our methodology.

The third view is introduced by adding time constraints to the dynamic behavior specification. For this purpose we used the timing extension of LSCs [23], by adding timed assignments and conditions to new and existing LSCs. We can also describe other extra-functional requirements using LSCs, for example, the possibilities of message loss with given probability. Other extra-functional properties such as performance (throughput), availability and dependability are not in the focus of this paper. Details on using LSCs for interface description corresponding to the second and third views appear in Sect. 5.

Component diagrams [40] describe a composite structure by providing a view of the set of embedded components and their interconnection. Such a structural design supports hierarchical composition. The top-level of the composite structure corresponds to the complete system provided to the client, in our case the telecommunication service `Depannage`. A partial structural view of the composite associated with the Depannage application is shown in Fig. 2. The diagram shows the main components involved and the communication between these components using ports and connectors. For this composite structure, we define the three description views: structural, dynamic behavior and extra-functional properties, as explained above.

3 Applying the Play-Engine to the Depannage service

3.1 Live sequence charts

This section reviews the main ideas and definitions underlying the language of LSCs and the play-in/play-out approach. For a detailed and systematic treatment the reader

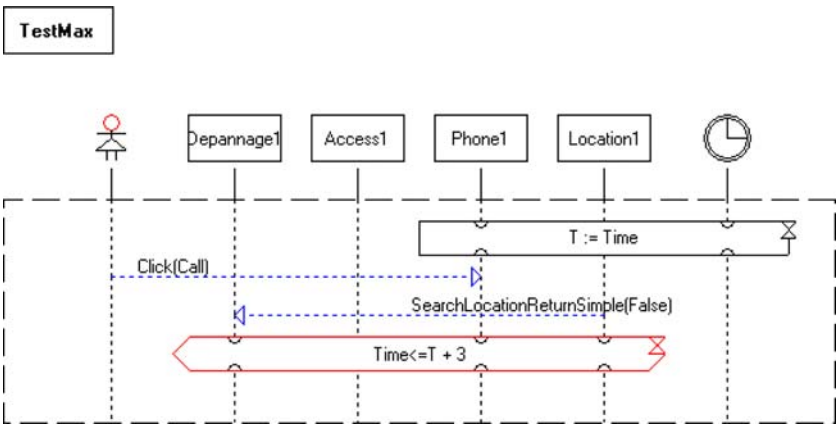


Fig. 3 Example of an existential chart

is referred to [12,24,25]. The main strength of LSCs over classical message sequence charts is the distinction between existential and universal charts. Common to both universal and existential charts is the notion of a scenario, in which several objects described by vertical lines communicate by exchanging messages specified using horizontal arrows. A scenario induces a partial order which is determined by the order along an instance line and by the fact that a message can be received only after it is sent.

Existential charts have a semantic interpretation close to that of a basic MSC in classical MSCs. LSC semantics requires that for an existential LSC there must be at least one run of the system satisfying the scenario, it does not require that this scenario hold for all runs. The chart appearing in Fig. 3 is an existential chart as denoted by the dashed borderline. The scenario starts with a timed assignment, followed by the user sending message `Click(Call)` to `Phone1` and then `Location1` sending message `SearchLocationReturnSimple(False)` to `Depannage1`. The timed condition holds if this behavior is performed within 3 time units causing the chart to be completed successfully. This interpretation of existential charts is useful in early system design for demonstrating possible behavior.

To extend the expressive power of classical MSCs, LSCs introduce the concept of a universal chart, which describes requirements that must hold for all runs, and is therefore constrained to specific circumstances specified by a scenario appearing in the prechart. An example of a simple universal chart appears in Fig. 4. Universal charts are denoted by a solid borderline. According to the LSC in Fig. 4, if the `Open` method is sent from the user to the `Cover`, as specified in the prechart (dashed hexagon), the self message `Sound(Silent)` appearing in the main chart must occur. The fact that this is a universal LSC means that this must hold for all system runs, i.e., for every run, each `Open` method must be eventually followed by a `Sound(Silent)` message.

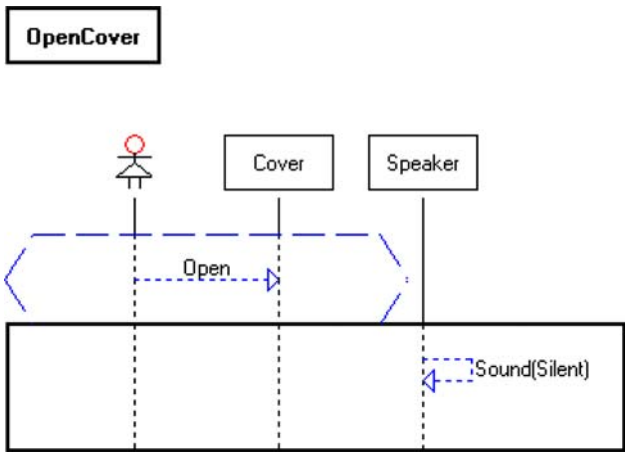


Fig. 4 Example of a universal chart

The core LSC language can be viewed as a graphical front-end to temporal logic, as is shown in [30]. The LSC language can also be used to specify more detailed requirements than the typical temporal logic properties, and for this purpose LSCs have been extended to support constructs that make it closer to a programming language, including variable assignments, loops, if–then–else, conditions, forbidden elements, parameterized messages, symbolic instances and discrete time [24].

We now present an outline of the formal definition of LSC semantics, capturing the requirements for a system to satisfy an LSC specification.

Formally, a **mode** of an LSC defines for each chart whether it is existential or universal.

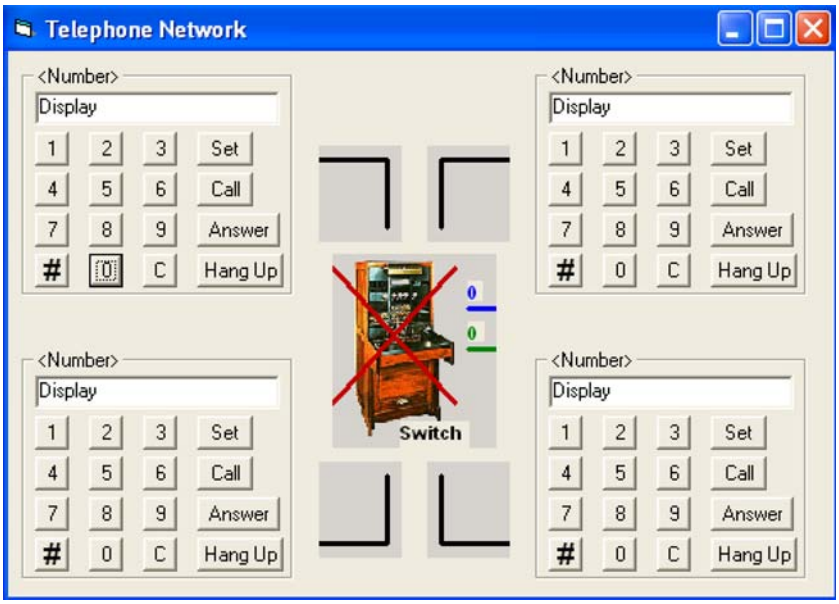
$$\text{mod} : m \rightarrow \{\text{existential, universal}\}$$

An *LSC specification* is a pair

$$\text{LS} = \langle M, \text{mod} \rangle,$$

where M is a set of charts, and *mod* is the mode of each chart.

Fig. 5 Phone GUI



The *language* of the chart m, denoted by $\mathcal{L}_m$, is defined as follows:

For an existential chart, $\text{mod}(m) = \text{existential}$, the language includes all traces for which the chart is satisfied at least once.

For a universal chart, $\text{mod}(m) = \text{universal}$, the language includes all traces for which each time the prechart is satisfied the behavior specified in the main chart follows.

Next we define for a given system S, which is compatible with an LSC specification LS (i.e., the system includes all objects, properties and messages referred to in the LSC specification) when the system satisfies the LSC specification.

Definition 1 A system S *satisfies* the LSC specification $\text{LS} = \langle M, \text{mod} \rangle$, written $S \models \text{LS}$, if:

1. $\forall m \in M, \ \text{mod}(m) = \text{universal} \Rightarrow \ \mathcal{L}_S \subseteq \mathcal{L}_m$
2. $\forall m \in M, \ \text{mod}(m) = \text{existential} \Rightarrow \ \mathcal{L}_S \cap \mathcal{L}_m \neq \emptyset$

Here $\mathcal{L}_S$ is the language consisting of all traces of system S.

3.2 Play-in

In play-in [25], requirements are captured by the user playing in scenarios using a (GUI) of the system under development or using an internal object diagram. The user "plays" the GUI by clicking buttons, and sending messages to objects. As this is being done, the supporting tool, the Play-Engine, records these actions in the form of a live sequence chart. Play-in can thus be viewed as an advanced user-friendly graphic editor for LSCs.

For the Depannage application we have reused an existing GUI of a phone application shown in Fig. 5, which is part of the Play-Engine samples and described in [24,33]. Most of the modeling has however been done using an internal object diagram, a variant of object model diagrams [40], as seen in Fig. 6. An internal object diagram is used to represent objects that are not part of the GUI. Typically, these internal (i.e.,

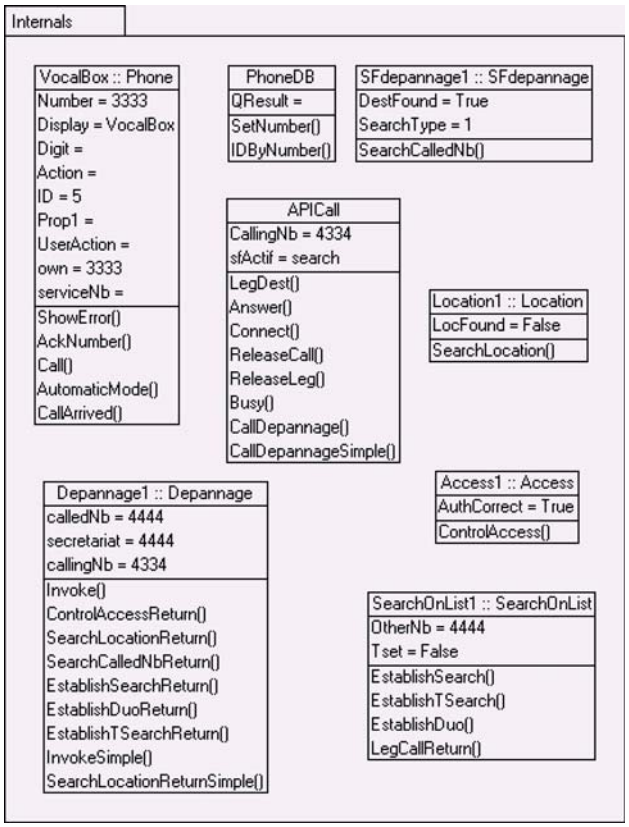
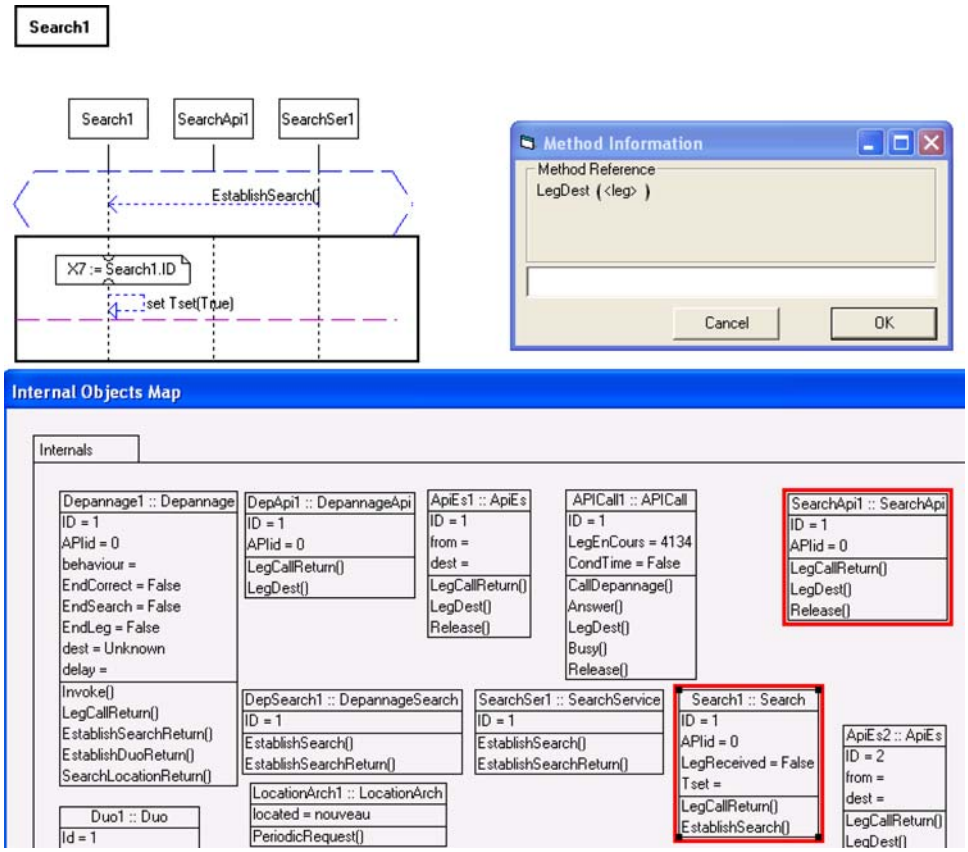


Fig. 6 Internal object diagram

Fig. 7 Play-in



GUI-less) objects represent more abstract objects that do not have a convenient or meaningful graphical representation. In internal object diagrams, each object is depicted by a box, showing its attributes and methods. The play-in and play-out processes are fully supported in the Play-Engine for internal objects.

An example of a play-in session is shown in Fig. 7. The Play-Engine user is designating a method call `LegDest` between objects `Search1` and `SearchApi1` by clicking on these objects (highlighted in the internal object diagram), after the user completes specifying the `LegDest` parameter the method will be added to the LSC `Search1` appearing in the background at the location of the dashed line. The method `LegDest` is used for establishing a communication connection between the network and a relevant party, as is explained in more detail in Sect. 5.

3.3 Play-out

Play-out is a complementary idea to play-in, which makes it possible to execute the requirements directly. Play-out can be viewed as a simulation tool for LSCs, where the user plays the role of the environment while the Play-Engine plays the role of the system. This contrasts with a process in which a system S is constructed manually to satisfy the specification LS. In response to an external event performed by the user, the Play-Engine monitors all participating universal LSCs to determine if a prechart has reached its maximal locations, thus activating the main chart. A typical situation in play-out is one in which executing a message in one LSC activates a new LSC, creating a cascade of events. A sequence of events carried out by the Play-Engine as a response to an external event input by the user is called a *superstep*. Play-out assumes that the Play-Engine can complete a superstep before the next external event is performed by the user.

Play-out does not guarantee satisfying the LSC specification. Thus, the system requirements for satisfying an LSC specification as presented in Definition 1 do not necessarily hold. There are two broad instances in which specifications can fail to be satisfied: (1) there may be existential charts that are never satisfied; and (2) some universal charts may be violated. The play-out mechanism of [24] is rather naive when faced with nondeterminism, and makes essentially an arbitrary choice among the possible responses. This choice may later cause a violation of the requirements, whereas a different choice could have satisfied the requirements. Technically, the nondeterminism has several causes: (1) the partial order semantics among events in each chart; (2) the ability to separate scenarios in different charts without having to state explicitly how they should be composed; and (3) an explicit nondeterministic choice construct, which can be used in conditions. This nondeterminism, although very

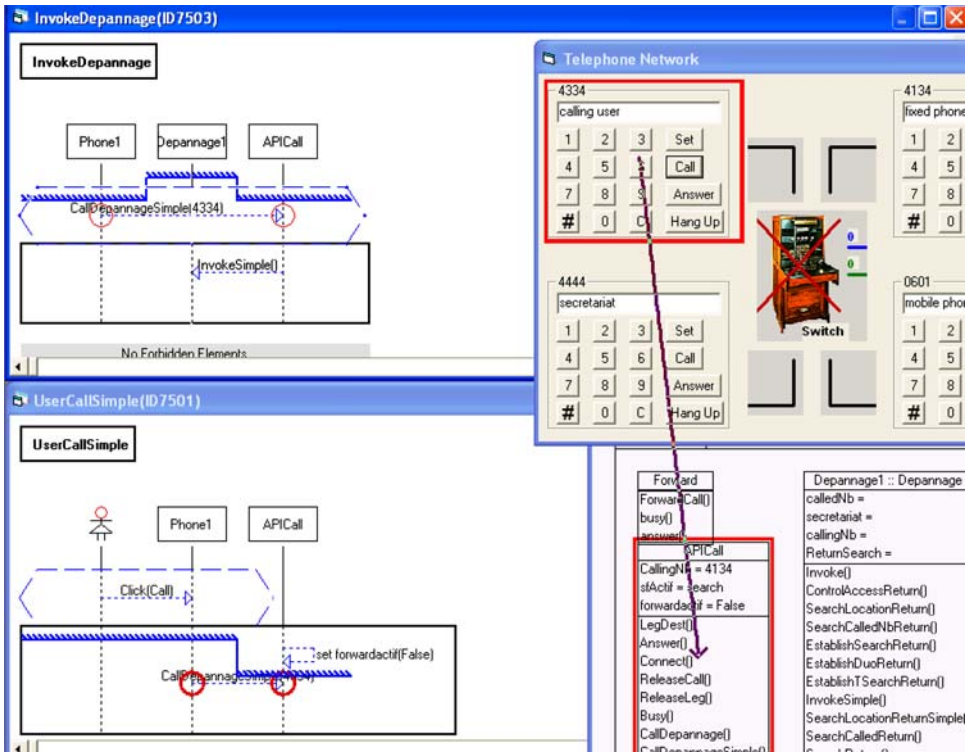


Fig. 8 Play-out

useful in early requirement stages, can cause undesired under-specification when one attempts to consider LSCs as the system's executable behavior using play-out. To address these problems [18] introduces a technique for executing LSCs, called *smart play-out*. Smart play-out uses model-checking to find a correct superstep if one exists. This is done by reducing the problem of finding a correct superstep (or satisfying an existential chart) to a model-checking problem [8], and then using the counter example provided by the model-checker as the correct superstep (or satisfying trace). Smart play-out is integrated in the Play-Engine tool and allows application of formal verification methods at early design stages in a user-friendly manner.

An example of a play-out session for the Depannage model is shown in Fig. 8. The LSC `UserCallSimple` at the lower left corner of Fig. 8 which is active, causes the method `CallDepannageSimple(4334)` appearing in its main chart to occur. This is reflected in the GUI and in the internal object diagram. Also as a result the LSC `InvokeDepannage` appearing in the upper left corner of Fig. 8 is activated, since the method `CallDepannageSimple(4334)` appears in its prechart.

4 Overall view of our design methodology

A classical problem in telecommunication is that of *feature interaction* [32]. The feature interaction problem occurs when the introduction of a new service (feature) causes the new system to allow violations of an existing service requirement. To illustrate the feature interaction problem let us consider an example from the Depannage application. A main advantage of the Depannage service is its ability to connect the user to a service provider in a timely and reliable manner—if at least one service provider is available within a predefined distance, it will be connected to the caller, otherwise the caller will be connected to the secretary of the Depannage society. For this reason, one of the requirements is not to connect a user to a vocal box of a mobile phone of a service provider. As we will show in detail later on, starting with a system that satisfies this requirement (no connection to a vocal box of a mobile phone), and adding a new feature supporting forwarding of calls, enables a scenario in which due to the interaction between features involving delicate timing issues, a user will be connected to the vocal box of a mobile phone. Here, we describe a methodology for detecting feature interactions.

Our methodology supports an incremental paradigm for specifying and developing telecommunication applications. First, we describe a high level LSC model of the service and component behavior, including the behavior of the communication between these components, using universal LSCs with relevant timed constraints. The requirements for the complete application are expressed as existential or universal LSCs. Then the consistency of this high level specification is validated, and testing is performed with respect to end-to-end

requirements. The analysis is performed initially by play-out, based on simulation and animation methods. In a second step, smart play-out is used in order to formally verify some of the requirements. We have to cope with the state explosion problem typical for model-checking techniques. This means that in many cases, we will have to work on a restricted part of the model.

Smart play-out is useful for the analysis of the feature interaction problem. In the telecommunication domain, feature interaction is defined as a problem occurring when several service features must coexist in the same architecture, and this may lead to incompatibilities. To verify the absence of feature interaction, we define a model of the underlying architecture, composed of the platforms, network and user components. For each service feature introduced in the architecture, we specify the components that correspond to the execution of this feature, and the feature requirement. An important point is that the feature requirements are not expressed on the component interfaces, but on the end-to-end requirements from the user point of view. It is the user requirements that involve the introduction of the feature components inside the embedded system. Next, we verify using smart play-out that the feature requirements are satisfied when each service feature is introduced in isolation on the architecture. A feature interaction problem occurs when a feature requirement is not satisfied when several service features are executed concurrently on the architectural model.

Our approach enables early experimentation with different components, and with different time constraints in the communication. These different time constraints correspond to different protocols. Feature interactions could also occur due to the timed constraints on the communication between components. Using LSCs with time constraints, we applied these techniques to an LSC model specifying the component behaviors, the communication between components and the end-to-end requirements. We have been able to detect an occurrence of feature interaction in the Depannage model, as shown in Sect. 7.

5 High level component and service specification using LSCs

The Depannage service is implemented as a layered application consisting of several components. Each layer or component is described by a set of scenarios; the connection between layers is defined such that the objects in each layer communicate only among themselves and with the objects in the adjacent layers. This architecture enables to break down the complexity of the system, abstracting a layer (or several adjacent layers) by replacing it with a simpler, more abstract specification. In this paper, we focus our presentation on the components `Search`, `API`, `Users` and the communication between these components. A detailed description of the entire model is available online at [10].

5.1 Search component

The `Search` component, as shown in Fig. 2, has two ports, `SearchService` for communicating with the application that uses it and `SearchApi` for communicating with platform components and indirectly with the users and the environment. An objective of the `Search` component is to initialize communication with several potential called parties, and eventually to set up a connection with one out of the set of potential service providers. The first called party that answers (and not too quickly, an answer that is too quick means that the communication is likely to be with the vocal box) is chosen as the final destination for the call.

The universal chart `Search1Exact`, appearing in Fig. 9, requires that whenever `SearchSer1` sends the `EstablishSearch` method to `Search1`, as specified in the prechart, the `Search1` port sets the value of `Tset` to `TRUE` and then sends the `LegDest(3)` method to `SearchApi1`. Intuitively, when the `Search` component is asked to establish a search, it will forward a request to the network `API` asking to make the communication between the network and the relevant party. In telecommunication a *leg* is the communication between the network and a party. A leg is also referred to as a half-call, but actually there can be more than two legs in a call, for example in a conference call. The message `LegDest(3)` appearing in Fig. 9 tries to establish the connection between the network and party 3, more generally the message `LegDest(x)` tries to connect the network and party x.

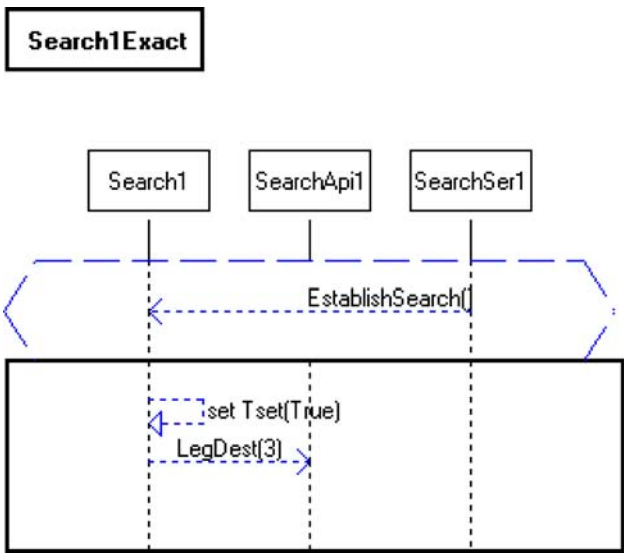


Fig. 9 First LSC for Search component—concrete

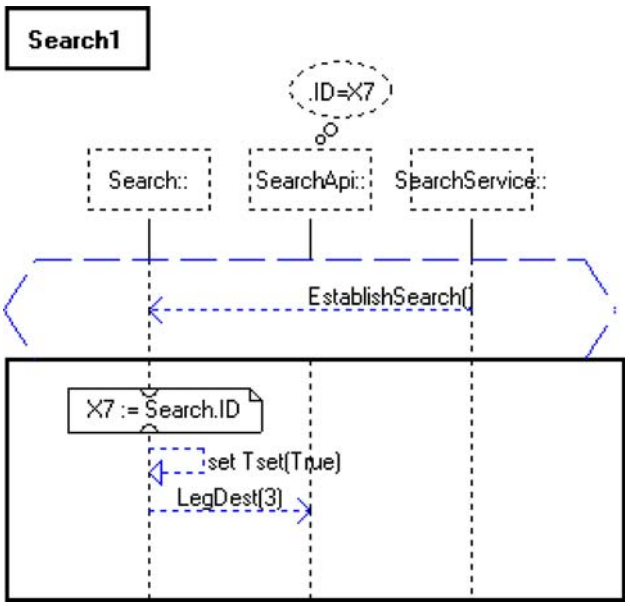


Fig. 10 First LSC for Search component—symbolic

In order to specify this requirement in a generic way, so it will hold for all other instantiations of the classes `SearchService`, `Search` and `SearchApi`, we use symbolic instances [33] as shown in the chart `Search1` in Fig. 10. Whenever an instance of class `SearchService` sends the `EstablishSearch` method to an instance of class `Search`, the `Search` instance sets the value of `Tset` to `TRUE` and then sends the `LegDest(3)` method to a `searchApi` instance which has an ID that is identical to the ID of the `Search` instance. This is done by storing the `Search` ID using an assignment to variable `X7`, and in the ellipse above the `SearchApi` instance specifying the binding condition `.ID = x7`, meaning that an instance of class `SearchApi` with ID equal to the value stored in `X7` will be bound to this chart, and then later the `LegDest(3)` method is sent to it.

The universal chart `Search2`, of Fig. 11 specifies a behavioral requirement that is relevant when the `SearchApi` gets information on the `LegCallReturn` and forwards it to the `Search` port, as is specified in the prechart, which contains a scenario and not a single message as in Fig. 10. The chart is activated if an instance of class `Search` sends the `LegDest(3)` method to a `searchApi` instance, and this `searchApi` instance sends the `LegCallReturn` message back to the `Search` instance. The `LegCallReturn` message is a confirmation from the network `API` about setting up the connection of a leg.

Another LSC feature introduced in Fig. 11 is the if–then–else construct used to specify conditional behavior. In the main chart, if the parameter of `LegCallReturn` is `FALSE` (the parameter is stored in variable `X337`) meaning that the

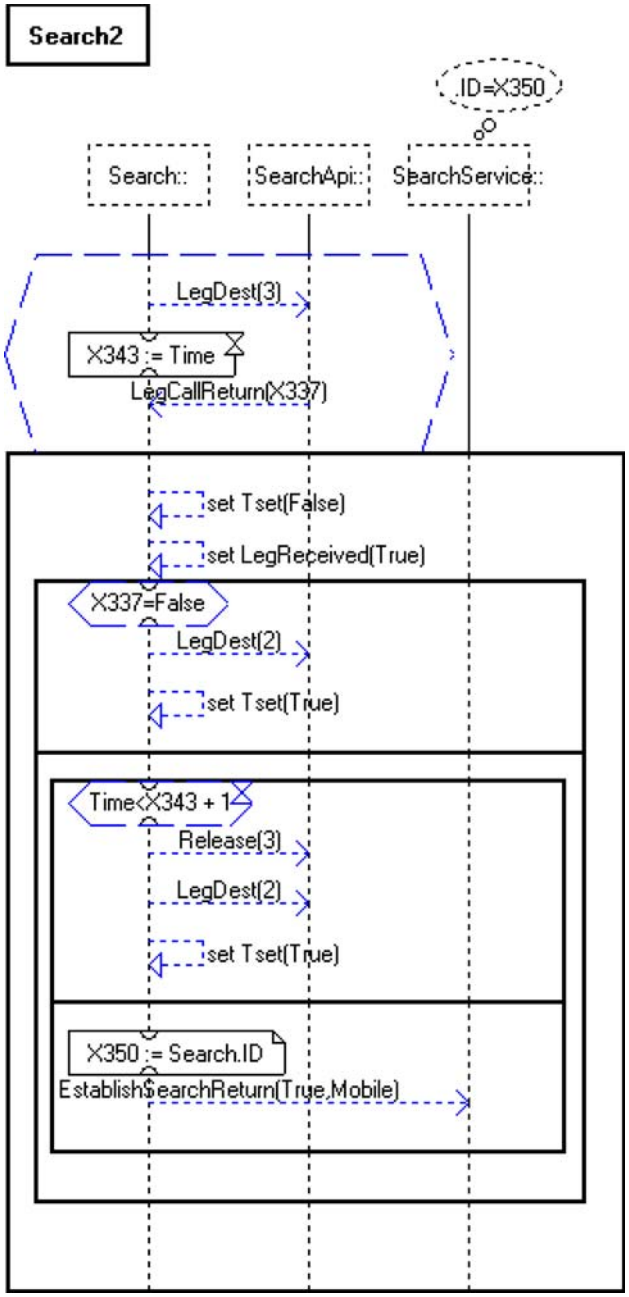


Fig. 11 Second LSC for Search component

communication for setting up a leg with party 3 has failed, then `Search` sends `LegDest(2)` to the `SearchApi` instance, trying to establish a communication path with party 2 instead, and sets a new timer by setting the value of `Tset` to `TRUE`. Otherwise, the other part of the subchart is taken, which involves a nested if–then–else construct. Here we branch according to the time that has elapsed since the `LegDest(3)` message was sent. If this time delay is smaller than one time unit, `Search` sends `Release(3)`, `LegDest(2)` and sets the value of `Tset` to `TRUE`. This corresponds to a situation in the system where a quick answer

by the mobile phone means that the caller would be connected to a vocal box, a situation which should be avoided in the Depannage service. In this case, the intention is to avoid the connection setup with party 3 and to establish a leg with party 2. If the time that has elapsed since the `LegDest(3)` message was sent is greater than or equal to one time unit the message `EstablishSearchReturn(TRUE, Mobile)` is sent to the appropriate `SearchService` instance, which means to continue the process of connecting to the called mobile phone.

5.2 Component platform: API

In our model there are predefined phones for secretary, fixed and mobile users. We defined only one phone of each category in the model, but the model can be extended to several phones of each category to capture a more realistic setting. The phones may return `busy` or `answer` when they are called. The LSC `CalledLeg` in Fig. 12 specifies that when an instance of class `ApiEs` sends a `LegDest` message to an instance of class `APICall`, this instance sends the message `callArrived` to the corresponding phone, according to the ID of the parameter in the `LegDest` message. This message can later on be used to trigger a notification of the called party by ringing the called phone.

The charts `APIanswer` and `APIbusy` of Fig. 13 and Fig. 14 respectively return the associated state of the destination phone by sending the message `LegCallReturn(True)` if there is an answer and the message `LegCallReturn(False)` if the phone is busy. For validation purposes, we introduce in the chart `APIanswer` of Fig. 13 a potential delay in the communication. This delay of more than two time units holds only if the boolean variable `CondTime` is `TRUE`. The value of the `CondTime` variable can be set before play-out and can also be modified during the play-out session. In reality, this delay corresponds to potential delays in the network or to the invocation of another service component. An alternative modeling option is to use nondeterminism to determine the length of the time delay, we used the option of determining the delay according to the variable `CondTime` for convenient testing.

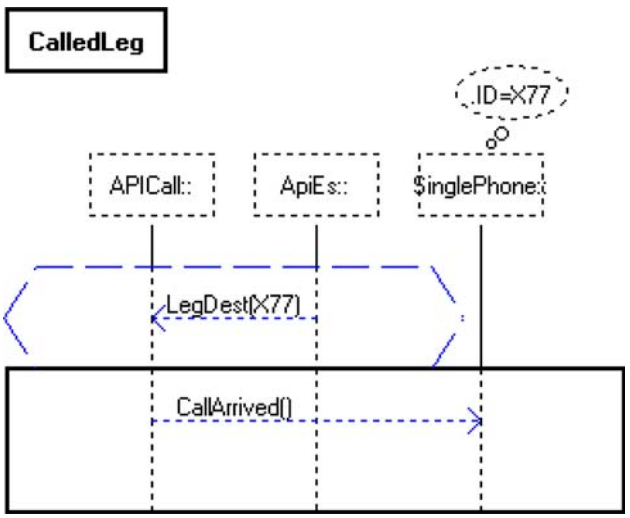


Fig. 12 APICall to the users

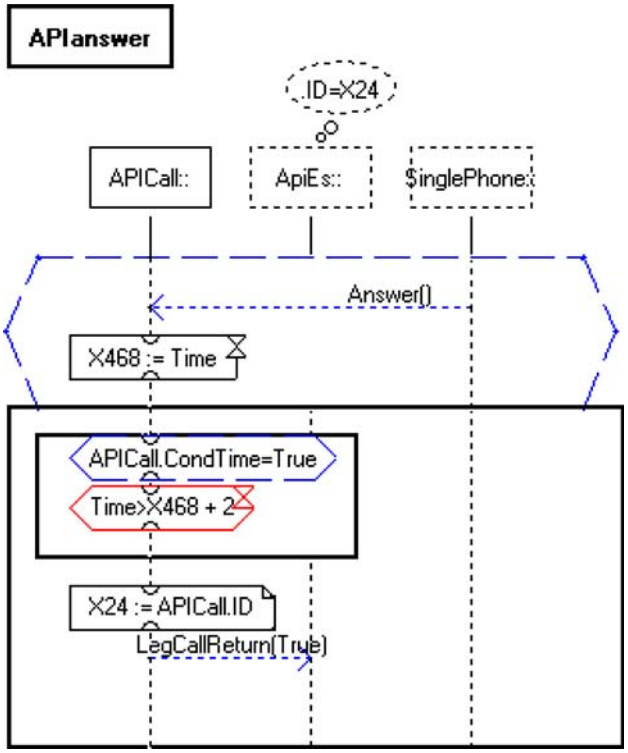


Fig. 13 APICall answer

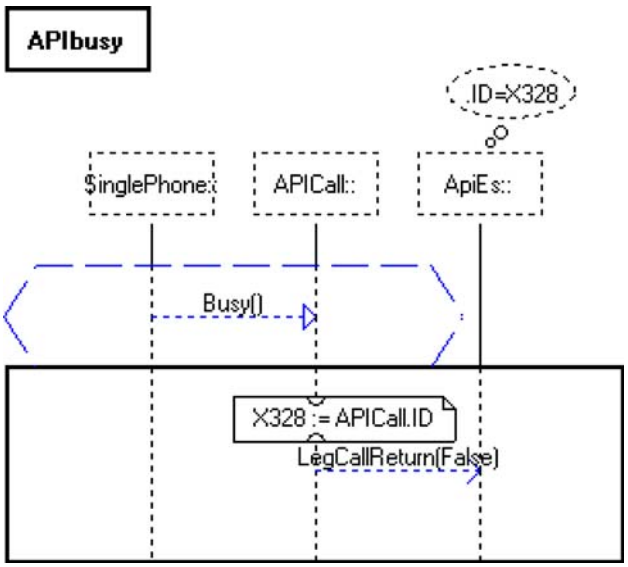


Fig. 14 APICall busy

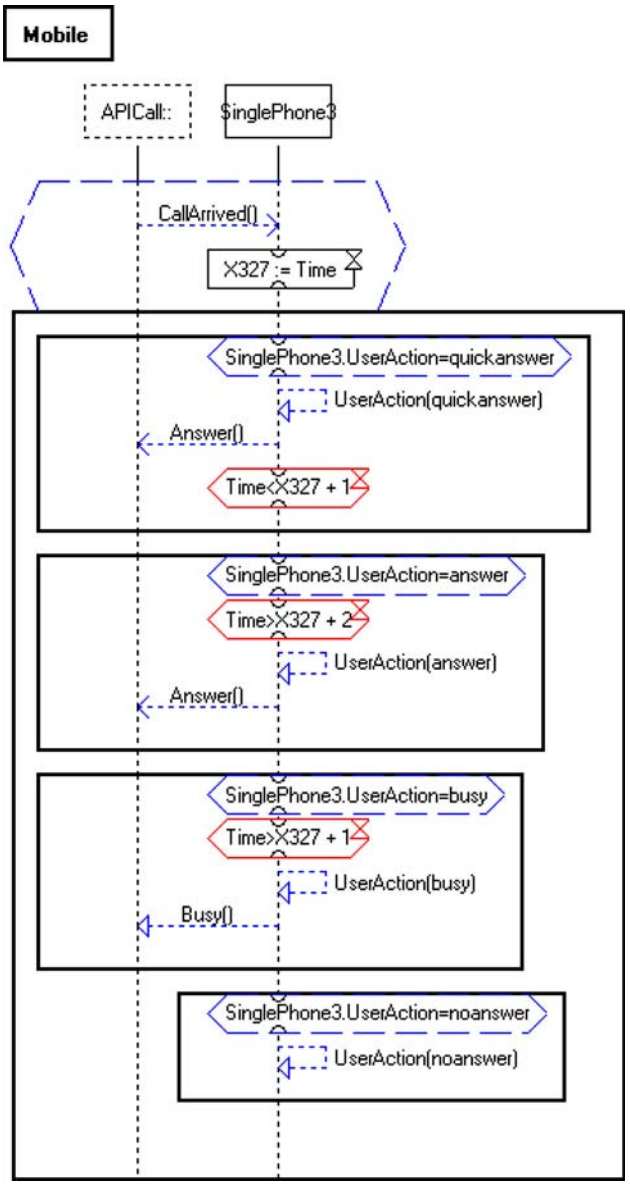


Fig. 15 The mobile phone

5.3 The users

We model a simple view of the service provider behavior, focusing for a mobile phone on four possible states, corresponding to user actions : `quickanswer`, `answer` (with a delay), `busy`, or `noanswer`. The specification of a mobile phone, shown in Fig. 15, describes the behavior of a mobile phone as a called party. When a `callArrived` message arrives to the mobile phone, this party may either answer by connecting to the vocal box of the mobile phone (`quickanswer`), make a normal answer after the ringing tone (`answer`), send a busy message (`Busy`), or do nothing (`noanswer`). The case that there is no answer is detected by another component in the network after the expiration of an appropriate timeout.

In general, if a mobile phone is reachable but disconnected, the communication will quickly be connected to the vocal box of the phone. This behavior should be taken into account carefully while designing the service. Some service logics should not connect the calling party to a vocal box. In the Depannage service we want to be connected to a person which is available or to a secretary or in the worst case to the vocal box of the Depannage company, but not to the vocal box of the mobile phone of one of the Depannage service providers.

5.4 The communication view

Developing a new telecommunication application is performed by taking existing components (each such component is already specified by a set of LSCs), and connecting them together. In our methodology this assembly of components is done by specifying universal LSCs defining the connection between components. Following the component diagram, these LSCs specify the communications between components. Such LSCs for connector behaviors may be simple or complex, depending on time constraints and delays, on the parallelism of thread execution, and on the fact that in the system architecture, a component port could be connected to several other component ports (for example the port `ApiES` of the component `ApiCall` in Fig. 2).

To specify the communication between two components following a a component diagram, we construct two LSCs for each event, allowing specific time constraints depending on the events. Consider the connection between the components `Depannage` and `Search`. We have to express that the event `EstablishSearch` required by the component `Depannage` and provided by the component `Search` should go through the port `DepannageSearch` of the component `Depannage` and the port `SearchService` of the component `Search`. This is described in the charts `DepToSearch1` and `DepToSearch2` in Fig. 16a, b. Similar LSCs are also specified for the return event `Establish SearchReturn` in Fig. 17a, b.

The connection between the components `Depannage` and `Location` is described in Figs. 18, 19. In these LSCs we also specify a communication delay. The LSC `DepToLoc1` of Fig. 18 specifies that invoking the method `SearchLocation` will take between one and two time units. The method `SearchLocation` is an asynchronous method, designated by the open arrow, in contrast to the closed arrows for synchronous methods. This time constraint is specified by storing the time in variable `×452` immediately after sending `SearchLocation` and adding the two hot conditions requiring that `Time > ×452 + 1` and `Time ≤ ×452 + 2`. A similar requirement that the method

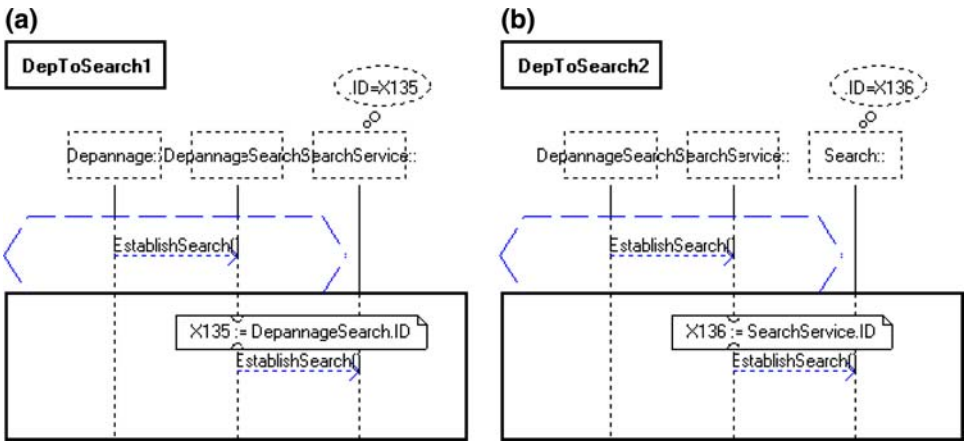


Fig. 16 Connectors between components Depannage and search

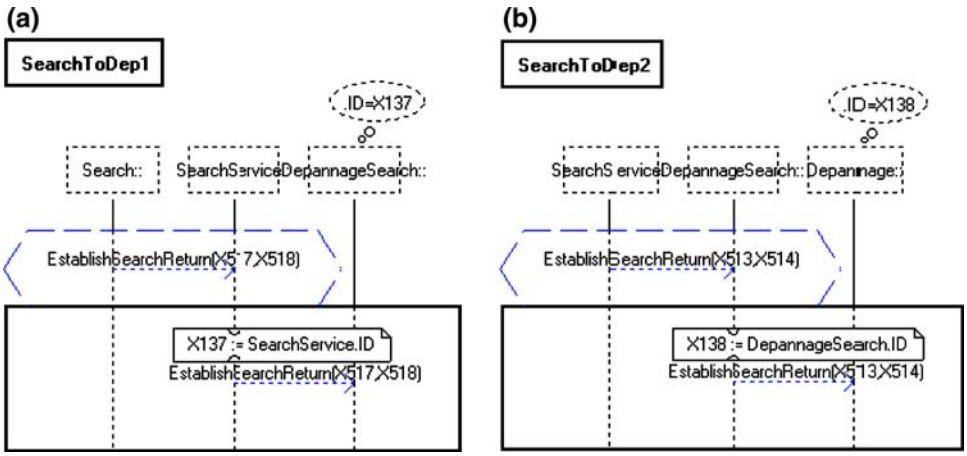


Fig. 17 Connectors between components search and Depannage

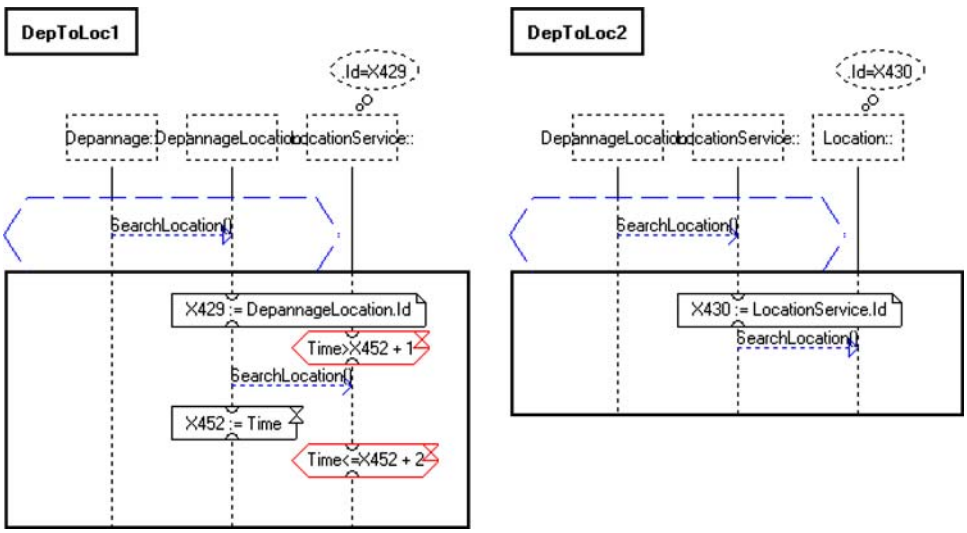


Fig. 18 Connectors between components Depannage and Location

`SearchLocationReturn` will take between one and two time units is specified in Fig. 19.

In some of the cases, describing the connection between components using LSCs is quite straightforward, as shown in the examples above, thus such LSCs could in principle be derived automatically by the tool using appropriate annotations on the component diagram.

6 Simulation using play-out

Play-out allows debugging requirements at an early stage and detecting problems in the design. We can designate LSCs that will be monitored during play-out. For monitoring purposes we can also use anti-scenarios, behavioral requirements that are forbidden in the system. Consider the chart

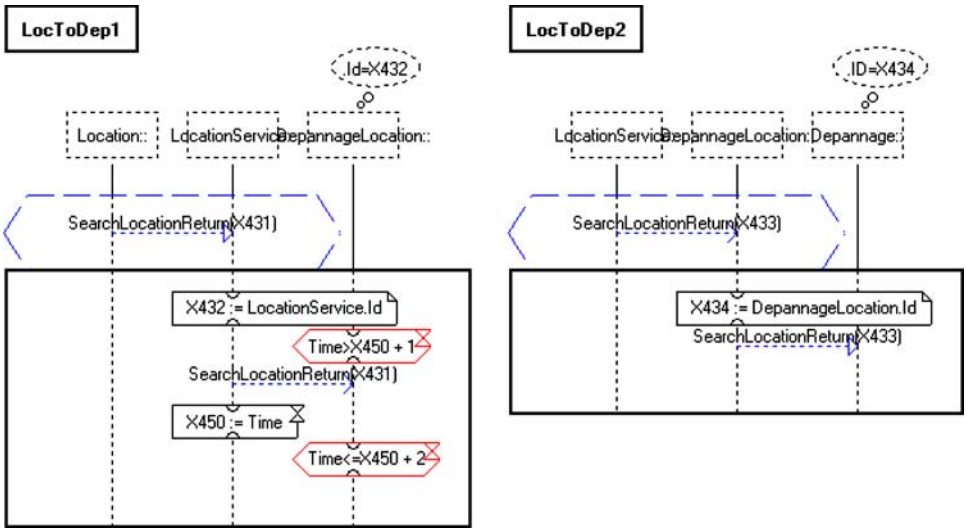


Fig. 19 Connectors between components Location and Depannage

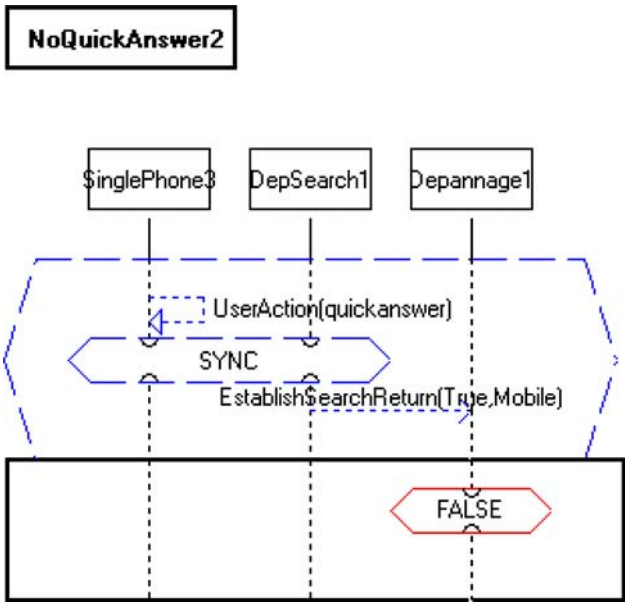


Fig. 20 A forbidden scenario—no connection to the vocal box of a mobile phone

`NoQuickAnswer2` described in Fig. 20. It specifies that whenever `SinglePhone3` makes a quick answer by sending the self message `UserAction(quickAnswer)` and after that `DepSearch1` sends the message `EstablishSearchReturn(True, Mobile)` to `Depannage1`, then the condition `FALSE` specified in the main chart must hold—which can never occur—implying that this sequence of messages specified in the prechart corresponding to a connection to the vocal box of a mobile phone must never occur.

In play-out mode, if this chart participates in the execution, the prechart will be traced and if it is completed the user will get a message that the system has aborted due to the violation of a hot condition, as shown in Fig. 21. In this case the violation was caused by a time delay in the `APICall` which is triggered by setting the property `CondTime` of this object to `TRUE`. In general, once a violation is detected it indicates a problem in the specification or the design of the service and needs to be looked into carefully to identify and fix the cause of the violation.

7 Verification using smart play-out

Smart play-out [18,19] uses verification methods, mainly model-checking, to execute and analyze LSCs. There are two main modes in which smart play-out can work. In the first mode smart play-out functions as an enhanced play-out mechanism, helping the execution to avoid deadlocks and violations. In the second mode, smart play-out is given an existential chart and asked if it can be satisfied without violating any of the universal charts. If it manages to satisfy the existential chart the satisfying run is played out, providing full information on the execution and reflecting the behavior in the GUI.

In the Depannage application we mainly used existential charts for specifying scenarios that should not occur, and then asked smart play-out if they can be satisfied. If the existential chart was satisfied, this means we have discovered an error in our specification model, and the execution can provide insights on what went wrong.

Consider the existential chart shown in Fig. 22. It describes a scenario that implies a user (on `Phone1`) being connected to the vocal box of a mobile phone (`Phone3`), an undesired behavior since then the user does not get a personal response to his request as is desired for the Depannage service. Smart play-out proves given the universal charts in the model that this scenario cannot be satisfied.

Fig. 21 Violation of a forbidden scenario during play-out

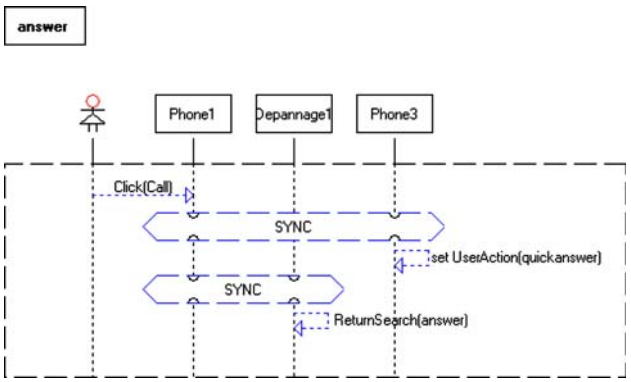


Fig. 22 An existential chart implying connection to the vocal box of a mobile phone

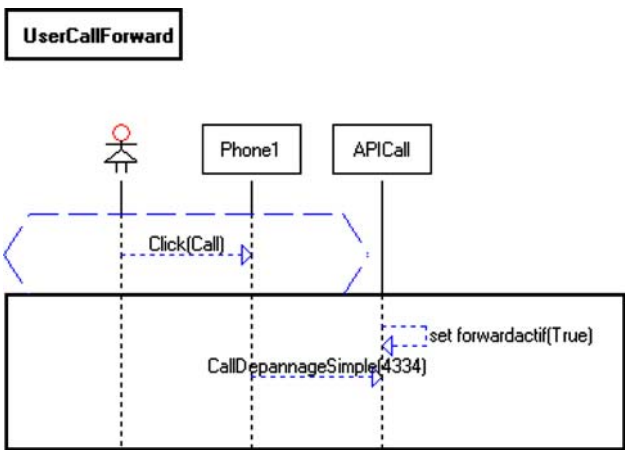


Fig. 23 A new feature of forwarding calls

We then added a new feature to our telecommunication model, forwarding calls, shown in Fig. 23, applied smart play-out, and it found a way to satisfy the chart of Fig. 22. The interaction of the new feature of the forwarding calls allowed an erroneous situation in which a user is connected to a vocal box. A short animation of this behavior is shown in [10]. In a similar manner we have verified also timed properties of the application, an example is shown in Fig. 24.

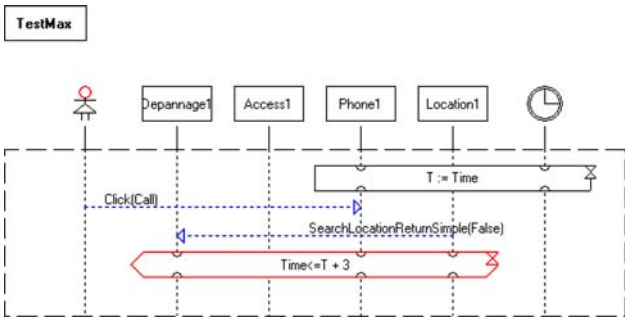


Fig. 24 Timing requirements

The version of smart play-out used in this work was restricted in terms of the LSC language features supported. Thus to use it some restrictions had to be made on the model: no symbolic-instances, and only one parameter for each signal. We have also abstracted and simplified the model to avoid the well known state-explosion problem. While play-out, which makes a 'local' choice during the selection of the next event for execution managed to execute efficiently a Depannage model containing approximately 100 LSCs, some of them using symbolic instances, in smart play-out we managed to deal with a model containing approximately 30 LSCs, without symbolic instances.

8 Additional Play-Engine features

We describe some of the Play-Engine features that are important in facilitating the scenario-based approach.

8.1 Setting the relevant LSCs: the execution configuration

The execution configuration allows the user to select the LSCs that are considered by the Play-Engine while executing the model in (smart) play-out mode. An example of the execution configuration dialog is shown in Fig. 25. The dialog is composed of three sections: the first allows the user to select the participating universal LSCs, the second to select

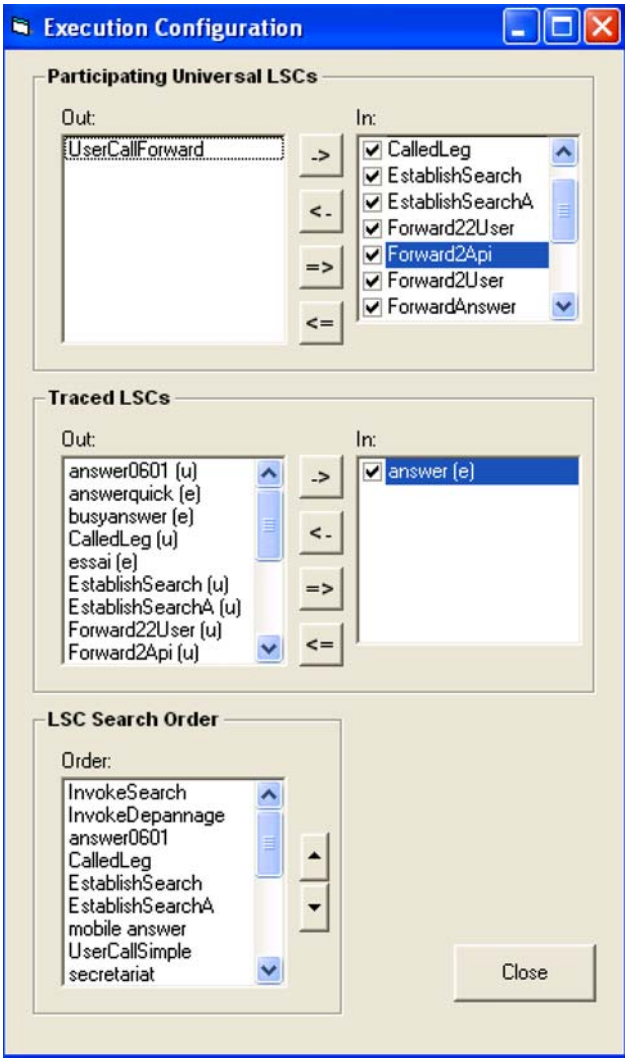


Fig. 25 The execution configuration

traced LSCs — either universal or existential charts, and the third designates the LSC search order. Charts selected as traced LSCs are monitored for their progress, but do not affect the execution itself. Thus, traced universal charts that are not in the set of participating LSCs will be monitored, and any activation or violation of them will be detected. However, a chart that is only traced will not cause any events in its main chart to occur, even if becoming active. The LSC search order contained in the third section is used by the play-out execution algorithm while searching for the next event to be executed. For the participating universal LSCs and traced LSCs sections, each chart that is included can be either checked or unchecked, specifying whether to show the chart when running in a mode where charts are open or to hide the chart. This allows the user to focus on the charts that are most interesting in a certain context.

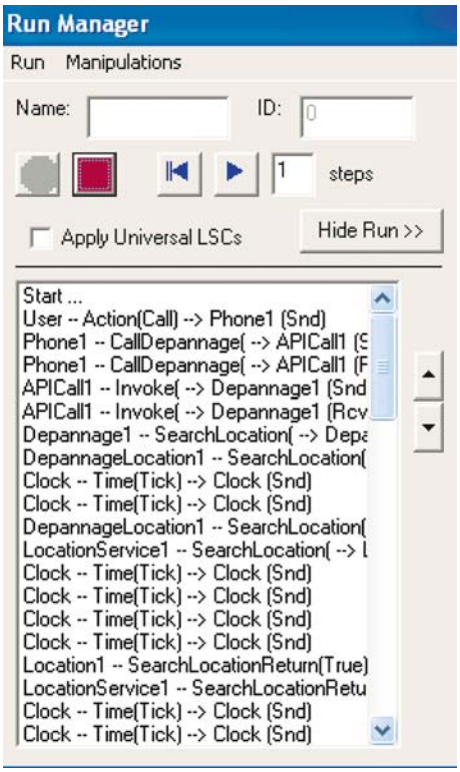


Fig. 26 Replaying a recorded run

8.2 Recording and loading runs: the run manager

The run manager allows recording an execution of the system during play-out and storing it. This trace can then be replayed later, by running it, either step by step or a given number of step each time. This allows the user to look more carefully at certain interesting behavior, or to display it to other people involved in the project. The run manager also allows removing all system events from a recorded run, and so remaining only with external events. These can be later fed by the run manager, and play-out will execute the system events in response to those external stimuli. The information is stored by the run manager in XML format defined in [24], and consists of sending or receiving of messages. An example of a recorded run from the Depannage model is shown in Fig. 26.

8.3 Connection with UML tools

This work was done as part of the OMEGA project [35], which focused on the development of verification methods for UML models constructed using industrial case tools. To connect the Play-Engine and LSC modeling to UML development, a Play-Engine model an be automatically transformed into a skeleton UML model, including all classes, methods, attributes and types. This is part of the tool support

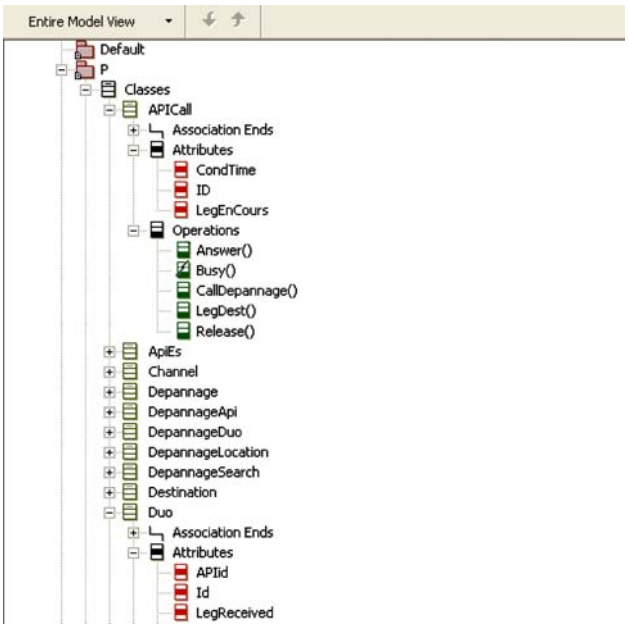


Fig. 27 Depannage model in Rhapsody

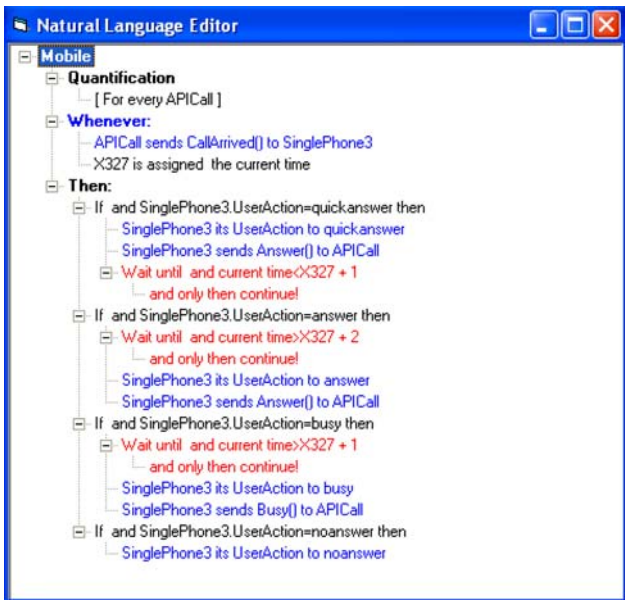


Fig. 28 Natural language translation

for synthesizing statechart based models from LSC specifications [20]. A fully automatic correct by construction synthesis algorithm from LSCs for large designs is still very hard to achieve, but even synthesizing a skeleton UML model, that can be taken and further developed manually is useful, an example of UML model for the Rhapsody tool generated from the Depannage model is shown in Fig. 27. The other direction, importing a UML Rhapsody model into the Play-Engine is also supported under the restriction that the actual code and statecharts are not imported, only the basic skeleton UML model.

8.4 Translation to natural language and report generation

The Play-Engine supports a translation from a given LSC to a natural language description (currently in English) of the behavior. An example of the translation for the LSC `Mobile` of Fig. 15 appears in Fig. 28. In addition, a comprehensive report on a model, in the form of a word document can be generated. Examples of such reports generated for the Depannage application are available at [10].

9 Related work

Scenario-based specification is very helpful in early stages of development [3], and is used widely by engineers. A considerable amount of experience has been gained from it being integrated into the MSC ITU standard [34] and the UML [40]. The latest versions of the UML recognized the importance of scenario-based requirements, and UML sequence diagrams have been significantly enhanced in expressive capabilities. Modal UML Sequence Diagrams (MUSD) [22] extend UML sequence diagrams based on the universal/existential modal semantics of LSCs. In [38] triggered message sequence charts are introduced, enhancing the expressive power of classical MSCs and supporting a notion of refinement. A survey on various scenario-based languages and methodologies is presented in [4,26].

Related research on LSCs includes application to hardware systems [5,6], where LSCs are translated to temporal logic formula and used in a verification environment. The relationship of LSCs and temporal logic is studied in [13,30]. In [37] LSCs are used in a UML verification environment for the Rhapsody tool. A framework for applying symmetry and data-type reductions for LSC verification of UML systems (with an a priori unbounded state space) is developed in [14]. A symbolic execution environment for LSCs based on constraint logic programming is described in [41]. In [21], we report on the methodological experience gained by using LSCs and the Play-Engine in several industrial case studies. (We briefly mention the Depannage application too.)

Deriving executable models from scenario-based specification via state-based models or statechart synthesis is a topic that focused much research in recent years, some representative research results are [17,27–29,31,42,43]. The consistency of scenario-based requirements and a notion of implied scenarios is developed in [39].

Other work uses model-checking for detecting feature interaction, based on formal models in SDL, LOTOS, LUSTRE, temporal logic, etc., [1,32,36], obtaining interesting results for classical telephony service features. Our

method can be applied beyond the domain of classical telephony and can handle timed properties.

Performance requirements—the number of requests that a system can manage—are very important in telecommunication applications but are not considered in this work. Simulation techniques based on queuing theory can be used for such performance evaluation. These techniques are, in many tools, based on the description of dynamic behavior as execution flows between components and machines. Thus, LSCs seem to be a suitable language for integrating performance evaluation and formal verification [9].

10 Conclusion and future directions

We have described an incremental methodology for the high level modeling and analysis of telecommunication applications, and its application to the nontrivial telecommunication service Depannage. This section contains additional information regarding the benefits and difficulties in applying the Play-Engine tool and our methodology, and mentions future directions for research and tool development.

In general, the language of LSCs proved well suited for our application domain. The main strengths are the visual representation (which is also one of the strengths of classical MSCs, on which LSCs are based), the enhanced expressive power (especially the ability to express required behavior and causality using universal charts), and the fact that the language is formal yet understandable by engineers and seems natural for expressing high level behavioral requirements. The fact that the same language is used in our methodology both for defining an executable model and for specifying the requirements turned out to be convenient in the work on the Depannage application.

The LSC language, originally defined in [12] and extended in [23,24,33] is a rich and complex language supporting many advanced features—symbolic instances and messages, timed requirements, forbidden elements and probabilistic choice. One of the questions that may arise is whether these extensions do not make the language too difficult and cumbersome for practical usage by telecommunication engineers. Our experience indicates that this is not the case, the advanced features turned out to be essential for the Depannage application—in particular the timed requirements and symbolic instances—and they are also well integrated into the language. Using LSCs does require a learning curve, despite the intuitive method for capturing requirements supported by play-in. A natural way to proceed while learning the language is to start from the more basic language features, using them for building simplified LSC models which can then be executed using play-out. Then, gradually, one can add more advanced behavior, e.g., by changing concrete instances into symbolic ones and by adding timing requirements.

Constructing an advanced graphical interface and utilizing the full power of play-in has not been a high priority in the work described here. Rather, we focused our efforts on constructing a model and requirements as efficiently and as quickly as possible, taking into account the current limitations of the Play-Engine, which, after all, is still a research prototype-level tool. For this purpose we reused an existing GUI of a phone system that is provided as one of the Play-Engine samples, and did much of the play-in using internal object diagrams. This last point turned out to be a good practical decision, allowing us to proceed quickly to the specification and analysis phase, without spending too much time on constructing the GUI.

We think that building a GUI is very worthwhile, especially for larger systems, since it allows additional stakeholders to participate in the design activity and it enables better feedback in early stages of the project. For this purpose we plan to improve the support for constructing GUIs in future Play-Engine versions, possibly by allowing the user to assemble and reuse existing GUIs and controls, as we have done in the Depannage model. Connection to existing commercial tools for GUI building [2,15,16] can also significantly improve the usability of the Play-Engine. In addition to play-in, the Play-Engine allows its user direct editing of LSCs through a graphic editor; it seems that for future versions it would be helpful to enhance these editing capabilities. An advantage of play-in is that it provides a high level of abstraction and does not force the user to directly construct LSCs. Another advantage we have observed is that playing in a scenario by actually demonstrating it on the GUI, enforces the dynamics of this in the designer's memory, which is helpful while in the requirement elucidation phase.

Play-out was very effective when building and analyzing the Depannage system. The ability to execute partial models, consisting initially of a small set of LSCs representing basic features, and gradually extending it by more advanced features—behavior fragments specified in new LSCs, allows incremental modeling and analysis. Often the LSCs were modified and extended according to the feedback obtained using play-out. For a real-world project in which the Play-Engine would be deployed in the telecommunication domain, play-out will play an important role in the communication between the development team members (telecommunication engineers and specialists in formal verification), managers and customers. Our main usage of play-out was for exploring the Depannage service design, as a preliminary stage before applying verification methods using smart play-out. An ambitious paradigm for system development put forward in [24] suggests using play-out as the system's final implementation. Investigating this possibility in the

telecommunication domain was beyond the scope of our work and remains as future work.

Using smart play-out we have managed to formally verify various properties for a simplified version of the Depannage model. The main restriction in terms of the LSC language was due to the fact that symbolic instances were not supported by smart play-out, thus a simplified model with concrete instances had to be prepared manually. As is the case for other automatic verification methods, scalability is a major problem, thus developing improved algorithms and methodologies for handling large designs efficiently is among the main future directions.

Acknowledgments Many thanks to all OMEGA partners for helpful feedback during the project. We would also like to thank the anonymous reviewers for constructive comments on the content and presentation of the paper.

References

1. Aggoun, I., Combes, P.: Observers in the SCE and the SEE to Detect and Resolve Services Interactions. IOS Press, Amsterdam (1997)
2. Altia Design & Altia FacePlate, web page http://www.altia.com
3. Alur, R., Holzmann, G., Peled, D.: An analyzer for message sequence charts. Softw. Concepts Tools **17**(2), 70–77 (1996)
4. Amyot, D., Eberlein, A.: An evaluation of scenario notations and construction approaches for telecommunication systems development. Telecommun. Syst. J. **24**(1), 61–94 (2003)
5. Bunker, A., Gopalakrishnan, G.: Verifying a VCI bus interface model using an LSC-based specification. In: Ehrig, H., Kramer, B.J., Ertas, A. (eds.) Proceedings of the 6th Biennial World Conference on Integrated Design and Process Technology, pp. 1–12 (2002)
6. Bunker, A., Slind, K.: Property generation for live sequence charts. Technical report, University of Utah (2003)
7. Castanet, R., Cavalli, A., Combes, P., Laurencot, P., MacKaya, M., Mederreg, A., Monin, W., Zaidi, F.: A multi-service and multi-protocol validation platform-experimentation results. In: TestCom, Lect. Notes in Comp. Sci., vol. 2978, pp. 17–32. Springer, Heidelberg (2004)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
9. Combes, P., Dubois, F., Monin, W., Vincent, D.: Looking for better integration of design and performance engineering. In: Reed, R. (ed.) SDL Forum, Lect. Notes in Comp. Sci., vol. 2708, pp. 1–17. Springer, Heidelberg (2003)
10. Combes, P., Harel, D., Kugler, H.: Supplementary material on the depannage application. http://research.microsoft.com/~hkugler/Depannage/
11. Combes, P., Harel, D., Kugler, H.: Modeling and Verification of a Telecommunication Application using Live Sequence Charts and the Play-Engine Tool. In: Proceedings of 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA '05), Lect. Notes in Comp. Sci., vol. 3707, pp. 414–428. Springer, Heidelberg (2005)
12. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods Syst. Des. **19**(1), 45–80 (2001). In: Preliminary version appeared in Proceedings of 3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)
13. Damm, W., Toben, T., Westphal, B.: On the expressive power of live sequence charts. In: Program Analysis and Compilation, Lect. Notes in Comp. Sci., vol. 4444, pp. 225–246. Springer, Heidelberg (2006)
14. Damm, W., Westphal, B.: Live and let die: LSC-based verification of UML-models. Sci. Comput. Program. **55**(1–3), 117–159 (2005)
15. e-SIM , web page http://www.e-sim.com/home
16. Macromedia Flash, web page http://www.adobe.com/products/flash
17. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Int. J. Found. Comput. Sci. (IJFCS) **13**(1), 5–51 (2002). (Also, Proceedings of 5th International Conference on Implementation and Application of Automata (CIAA 2000), July 2000, Lecture Notes in Computer Science. Springer, Heidelberg, 2000)
18. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: Proceedings of 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon, Lect. Notes in Comp. Sci., vol. 2517, pp. 378–398 (2002). Also available as Tech. Report MCS02-08, The Weizmann Institute of Science
19. Harel, D., Kugler, H., Pnueli, A.: Smart play-out extended: time and forbidden elements. In: International Conference on Quality Software (QSIC04), pp. 2–10. IEEE Press, New York (2004)
20. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: generating statechart models from scenarios-based requirements. In: Formal Methods in Software and System Modeling, Lect. Notes in Comp. Sci., vol. 3393, pp. 309–324. Springer, Heidelberg (2005)
21. Harel, D., Kugler, H., Weiss, G.: Some methodological observations resulting from experience using LSCs and the play-in/play-out approach. In: Proceedings of scenarios: models, algorithms and tools, Lect. Notes in Comp. Sci., vol. 3466, pp. 26–42. Springer, Heidelberg (2005)
22. Harel, D., Maoz, S.: Assert and negate revisited: modal semantics for UML sequence diagrams. Softw. Syst. Model. (SoSyM) (2007) (to appear). In: Early version in 5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06)
23. Harel, D., Marelly, R.: Playing with time: on the specification and execution of time-enriched LSCs. In: Proceedings of 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02). Fort Worth, Texas (2002)
24. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, (2003)
25. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play in/play-out approach. Softw. Syst. Model. (SoSyM) **2**(2), 82–107 (2003)
26. Harel, D., Thiagarajan, P.: Message sequence charts. In: UML for Real: Design of Embedded Real-time Systems (2003)
27. Koskimies, K., Makinen, E.: Automatic synthesis of state machines from trace diagrams. Softwa. Practice Exp. **24**(7), 643–658 (1994)
28. Koskimies, K., Mannisto, T., Systa, T., Tuomi, J.: SCED: a tool for dynamic modeling of object systems. Technical Report A-1996-4, University of Tampere (1996)
29. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to statecharts. In: Proceedings of International Workshop on Distributed and Parallel Embedded Systems (DIPES'98), pp. 61–71. Kluwer, Dordrecht (1999)
30. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal logic for scenario-based specifications. In: Halbwachs, N., Zuck, L.D. (eds.) Proceedings 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), Lect. Notes in Comp. Sci., vol. 3440, pp. 445–460. Springer, Heidelberg (2005)

31. Leue, S., Mehrmann, L., Rezai, M.: Synthesizing ROOM models from message sequence chart specifications. Technical Report 98-06, University of Waterloo (1998)
32. Logrippo, L., Amyot, D.: Feature Interactions in Telecommunications and Software Systems VII. IOS Press, Amsterdam (2003)
33. Marelly, R., Harel, D., Kugler, H.: Multiple instances and symbolic variables in executable sequence charts. In: Proceedings of 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02), pp. 83–100. Seattle, WA (2002)
34. ITU-TS Recommendation Z.120 (11/99): MSC 2000. ITU-TS, Geneva (1999)
35. OMEGA—Correct development of real-time embedded systems. http://www-omega.imag.fr/
36. Reiff-Marganiec, S.: Feature Interactions in Telecommunications and Software Systems VIII. IOS Press, Amsterdam (2005)
37. Schinz, I., Toben, T., Westphal, B.: The Rhapsody UML Verification Environment. In: 2nd International Conference on Software Engineering and Formal Methods. IEEE Computer Society Press, New York (2004)
38. Sengupta, B., Cleaveland, R.: Triggered message sequence charts. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 167–176. ACM Press, New York (2002)
39. Uchitel, S., Kramer, J., Magee, J.: Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. ACM Trans. Softw. Eng. Methods **13**(1), 37–85 (2004)
40. UML: Documentation of the unified modeling language (UML) (2007). Available from the Object Management Group (OMG), http://www.omg.org
41. Wang, T., Roychoudhury, A., Yap, R., Choudhary, S.: Symbolic execution of behavioral requirements. In: Jayaraman, B. (ed.) 6th International Symposium on Practical Aspects of Declarative Languages, (PADL 2004), Lect. Notes in Comp. Sci., vol. 3057, pp. 178–192. Springer, Heidelberg (2004)
42. Whittle, J., Saboo, J., Kwan, R.: From scenarios to code: an air traffic control case study. In: 25th International Conference on Software Engineering (ICSE 2003), pp. 490–495. IEEE Computer Society, New York (2003)
43. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: 22nd International Conference on Software Engineering (ICSE 2000), pp. 314–323. ACM Press, New York (2000)
44. Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva (1996)

Author's Biography

Pierre Combes is working in the research center of France Telecom for several years. He is responsible for the France Telecom research program on Dependability and Performance of software systems for telecommunication and he is a senior expert in this domain. He has been working in several European and national projects, in particular he is responsible for the French project PerSiForm on software performance.

David Harel has been at the Weizmann Institute of Science in Israel since 1980. He was Department Head from 1989 to 1995, and was Dean of the Faculty of Mathematics and Computer Science between 1998 and 2004. He was also co-founder of I-Logix, Inc. He received his PhD from MIT in 1978, and has spent time at IBM Yorktown Heights, and sabbaticals at Carnegie-Mellon University, Cornell University and the University of Edinburgh. In the past he worked mainly in theoretical computer science (logic, computability, automata, database theory), and now he works in software and systems engineering, modeling biological systems, and the synthesis and communication of smell. He is the inventor of statecharts and co-inventor of live sequence charts, and co-designed Statemate, Rhapsody and the Play-Engine. Among his awards are the ACM Karlstrom Outstanding Educator Award (1992), the Israel Prize (2004), the ACM SIGSOFT Outstanding Research Award (2006), and three honorary degrees. He is a Fellow of the ACM and of the IEEE.

Hillel Kugler is a scientist at Microsoft Research Cambridge. Prior to that he was a postdoctoral researcher at New York University. He received a Ph.D. in Computer Science in 2004 from the Weizmann Institute of Science in the area of visual languages and formal verification. His research interests include scenario-based requirements, statecharts, object-oriented analysis and design, formal verification and synthesis, and modeling and analysis of biological systems.