Modeling and Verification of a Telecommunication Application using Live Sequence Charts and the Play-Engine Tool *

Pierre Combes[1], David Harel[2], and Hillel Kugler[3]

[1] France Telecom Research and Development, Paris, France
Pierre.Combes@francetelecom.com
[2] The Weizmann Institute of Science, Rehovot, Israel
dharel@weizmann.ac.il
[3] New York University, New York, NY, USA
kugler@cs.nyu.edu

Abstract. We apply the language of live sequence charts (LSCs) and the Play-Engine tool to a real-world complex telecommunication service. The service, called `Depannage`, allows a user to make a phone call and ask for help from a doctor, the fire brigade, a car maintenance service, etc. This kind of service is built on top of an embedded platform, using both new and existing service components. The complexity of such applications stems from their distributed architecture, the various time constraints they entail, and the fact the underlying systems are rapidly evolving, introducing new components, protocols and associated hardware constraints, all of which must be taken into account. We present the results of our work on the specification, animation and formal verification of the Depannage service, and draw some initial conclusions as to an appropriate methodology for using a scenario-based approach in the telecommunication domain. The complete specification of the Depannage application in LSCs and some animations showing simulation and verification results are made available as supplementary material. [1]

1 Introduction

The challenging complexity of telecommunication systems, together with a high demand for rapid deployment, encourages development of innovative techniques in order to design and deploy new applications in a quick and secure manner [2]. In the telecommunication domain, components play a crucial role. The majority of these components is embedded in a large and complex architecture which involves hard and soft real-time constraints and requirements. Moreover, non-functional requirements, in particular time dependent properties, also play an important role. A telecommunication application is always built from a set of

* This research was supported in part by the European Commission project OMEGA (IST-2001-33522) and by the Israel Science Foundation (grant No. 287/02-1).

[1] `http://cs.nyu.edu/~kugler/Depannage/`

embedded service components, and in the emerging architecture a challenge is providing a ubiquitous environment for telecommunication users. This means that the telecommunication applications should be provided in several contexts with a high level of quality of service, and always in a comprehensive way to the end-users. Nowadays, due to openness of the telecommunication architecture, a multiplicity of services and service features could be provided by several teams or companies, and must be dynamically added and updated. The consistent use of components and service features is becoming more critical in order to ensure that undesired behaviors do not occur [11]. The time seems ripe to go from ad-hoc techniques for component composition toward more integrated and formal ones. Such techniques should be based on the use of formal languages for design and verification. The languages and design models should be readable in order to facilitate the communication between telecommunication engineers and specialists in formal verification. A comprehensive animation tool is also very important in order to enhance the understanding of the model, and in order to show verification results to engineers and clients [4]. A proposed approach should enable quick and secure telecommunication service creation, answering questions like how to build an architecture based on a set of components (reused or/and shared by several services) in such a way that we can guarantee providing complete applications respecting quality of service and safety requirements.

2 Live sequence charts and the Play-Engine

Understanding system and software behavior by looking at various "stories" or scenarios seems a promising approach, and it has focused intensive research efforts in the last few years. One of the most widely used languages for specifying scenario-based requirements is that of message sequence charts (MSCs), adopted long ago by the ITU [15], or its UML variant, sequence diagrams [14]. Sequence charts (whether MSCs or their UML variant) possess a rather weak partial-order semantics that does not make it possible to capture many kinds of behavioral requirements of a system. To address this, while remaining within the general spirit of scenario-based visual formalisms, a broad extension of MSCs has been proposed, called live sequence charts (LSCs) [6]. Among other things, LSCs distinguish between behaviors that must happen in the system (universal) from those that may happen (existential). A universal chart contains a *prechart*, which specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body. Existential charts specify sample interactions between the system and its environment, and must be satisfied by at least one system run. They thus do not force the application to behave in a certain way in all cases, but rather state that there is at least one set of circumstances under which a certain behavior occurs. The distinction between mandatory (hot) and provisional (cold) applies also to other LSC constructs, e.g., conditions and locations, thus creating a rich and powerful language, which among many other things can express forbidden behavior ("anti-scenarios").

In [9, 10] a methodology for specifying and validating requirements, termed the "play-in/play-out approach", is described, as well as a supporting tool called the Play-Engine. According to this approach, requirements are captured by the user playing in scenarios using a graphical interface of the system to be developed or using an object model diagram. The user "plays" the GUI by clicking buttons, rotating knobs and sending messages (calling functions) to objects in an intuitive manner. By similarly playing the GUI, the user describes the desired reactions of the system and the conditions that may, must or may not hold. As this is being done, the supporting tool, the Play-Engine, constructs a formal version of the requirements in the form of LSCs. Note that it is not always necessary to spend much time designing a fancy graphical interface. In many cases, it is enough to use a standard object model diagram. The Play-Engine tool, supports class diagrams and allows to work with internal objects that are not reflected in the GUI.

Play-out is a complementary idea to play-in, which, rather surprisingly, makes it possible to execute the requirements directly. In play-out, the user simply plays the GUI application as he/she would have done when executing a system model, or the final system implementation, but limiting him/herself to "end-user" and external environment actions only. While doing this, 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.

Smart play-out [7] is a powerful technique for executing scenario-based requirements using verification methods. It can be used for driving the execution of the system, or for checking if a given existential chart can be satisfied without violating any of the universal charts. Smart play-out is integrated in the Play-Engine tool and allows developers to apply formal verification methods at early design stages in a user-friendly manner.

3 Components and System Architecture

3.1 The Telecommunication application

We apply LSCs and the Play-Engine to a telecommunication service called Depannage, provided by France Telecom. The Depannage service allows a user to make a phone call and ask for the help of a doctor, fire brigade, car maintenance, etc. The service invocation software first asks for authentication of the calling user, and then searches for the calling location. Once the calling location is found, the software searches in a data base for numbers of potential service providers corresponding to the Depannage society members in the vicinity of the caller. Once various numbers are found, the service tries to connect the caller to one of the potential called numbers (in a sequential or parallel way). In any case the caller should be connected to a secretary or to a vocal box. In parallel a second

logic will make periodic location requests to the Depannage society members in order to record their latest locations in the data base. The Depannage service is implemented as a layered application consisting of several components. Each layer or component is described by a group of scenarios; the connection between layers is very clean and precise. The objects in each layer communicate only among themselves and with the objects in the adjacent layers. This architecture enables applying methodological approaches to break down the complexity of the system.

3.2 Components and composites

A telecommunication system is based on a set of components — reusable software units specified by their interfaces. The specification of these interfaces should be given by the signatures of the required and provided methods and signals, and by the description of the dynamic behaviors. Components should be reusable, thus they should be specified independently of any embedding system.

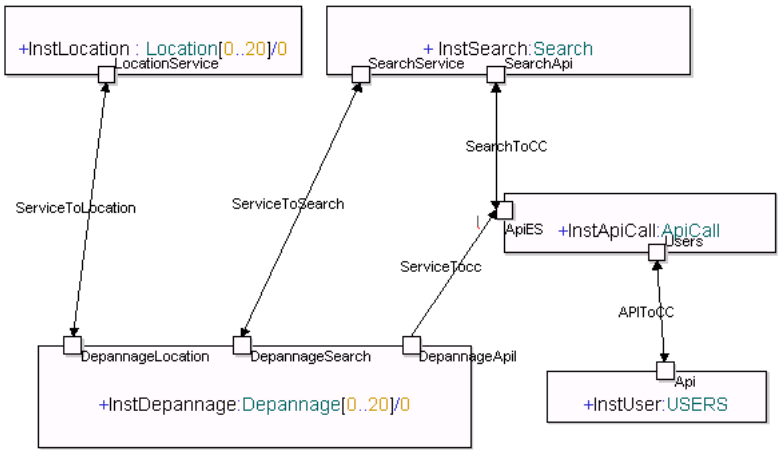

Fig. 1. The architecture of the Depannage application

A composite structure will be specified as a white box by the set of embedded components and the connections between these components [14]. Such structural design could use hierarchical composition. The top-level of the composite structure will correspond to the complete system provided to the client, in our case the telecommunication service Depannage.

Fig. 1 shows a partial view of the complete application (using UML composite structure diagram), the main components involved and the communication between these components using ports and connectors.

4 Overall view of a design methodology based on verification

A classical problem in telecommunication is that of "feature interaction" [11]. Telecommunication infrastructure and applications are in a continuous evolution, new services and service features are developed and deployed in the network along with existing ones. They are developed by several teams in parallel, in order to satisfy new customer requirements. The feature interaction problem occurs when the introduction of a new service (feature) causes the new system to violate an existing service requirement. This is a critical problem in telecommunication — involving significant loss of time and money during testing and operation phases. It can be properly solved only by identifying the problems during the design and modeling phases.

To address these issues we present a methodology that supports an incremental paradigm for specifying and developing telecommunication applications. First, we describe a high level specification of the service and component behavior, including the behavior of the communication between these components. This description includes timed constraints. 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 simulation and animation methods. In a second step, smart play-out is used in order to formally verify some of the requirements.

5 High level specification

The wish to specify components in a reusable way requires that the component specification should be done independently of any embedding architecture. Such specification should correspond in a universal LSC to an abstract view of the component, describing how the component will react to events coming from its provided ports and how (and when) this component will act on its required ports (execution flows).

For the system — i.e., the complete application — the specification should be enhanced by universal LSCs describing the communications between these components. Such LSCs could include time constraints and delays on the communication. The end-to-end requirements are expressed by existential LSCs and will be validated during the simulation/animation of the model.

In this paper, we will focus our presentation on the `Search` component, the `Users` component and the communication between these components. A detailed description of the entire model is available online at [5].

5.1 Search component

This component has two ports, `SearchService` for communicating with the application that will use it and `SearchApi` in order to communicate with platform components and indirectly with the users and the environment.

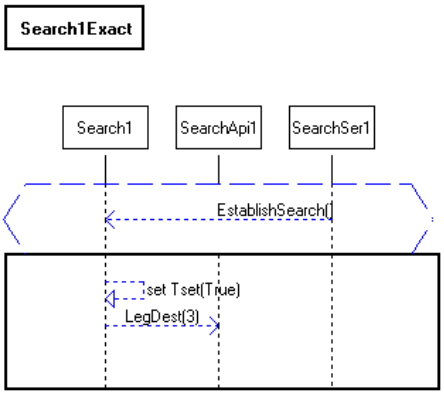

Fig. 2. First LSC for Search Component - Concrete

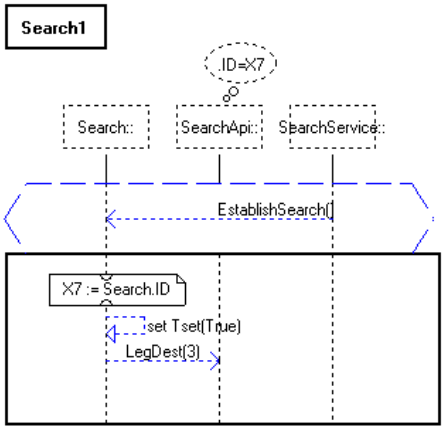

Fig. 3. First LSC for Search Component - Symbolic

The universal chart `Search1Exact`, appearing in Fig. 2, 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`.

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 [12] as shown in the chart `Search1` in Fig. 3. 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

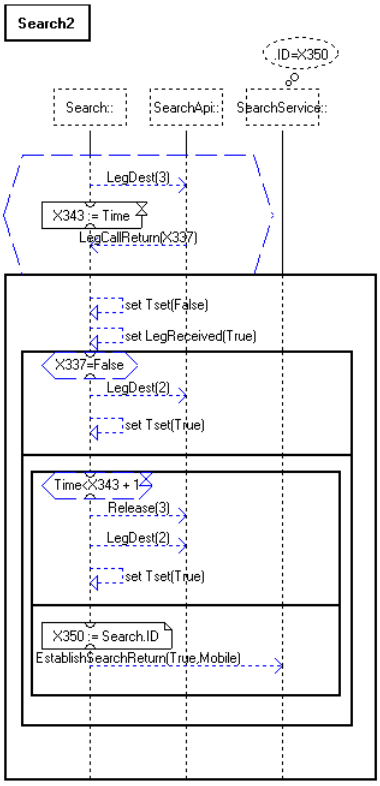

Fig. 4. Second LSC for Search Component

`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 will be sent to it.

The universal chart `Search2`, appearing in Fig. 4 specifies a behavioral requirement that is relevant when the `SearchApi` gets information on the `LegCallReturn` and forwards it to the `Search` port. The prechart of Fig. 4 contains a scenario and not a single message as in Fig. 3. The chart will be 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.

Another LSC feature introduced in Fig. 4 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`) then `Search` sends `LegDest(2)` to the `SearchApi` instance and sets 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 is less than 1 time unit

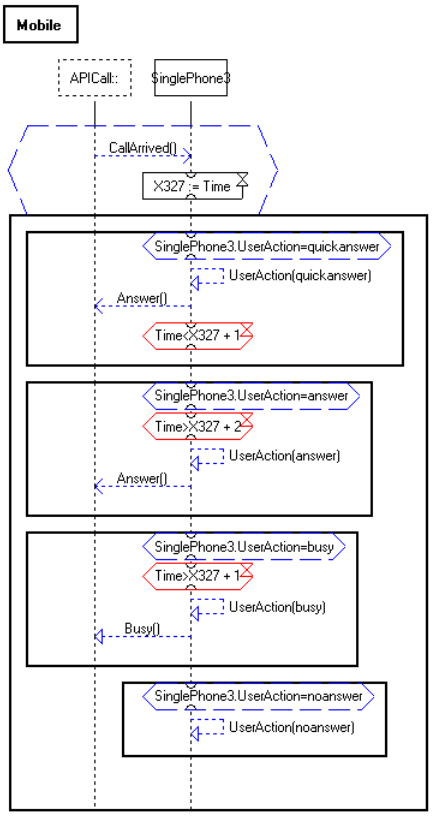

Fig. 5. The Mobile Phone

`Search` sends `LegDest(2)` and sets the value of `Tset` to `TRUE` as before. This corresponds to a situation in the system where a very quick answer by the mobile phone means that we will be connected to its vocal box, a situation which should be avoided in the Depannage service. If the time that has elapsed since the `LegDest(3)` message was sent is greater than or equal to 1 time unit the message `EstablishSearchReturn(TRUE, Mobile)` is sent to the appropriate `SearchService` instance, corresponding to continuing the process of connecting to the mobile phone.

5.2 The users

We model only a simple view of the user behavior, focusing for a fixed phone on three possible states, corresponding to user actions : `busy`, `answer` with a delay, or `noanswer`. The specification of a mobile phone, shown in Fig. 5 introduces an additional state `quickanswer`. In reality, if a mobile phone is reachable but in a disconnected state, 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.

5.3 The communication view

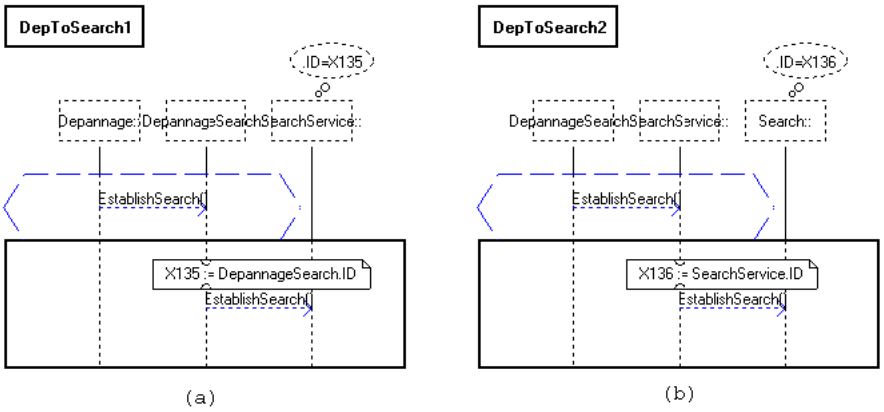

Fig. 6. Connectors between components Depannage and Search

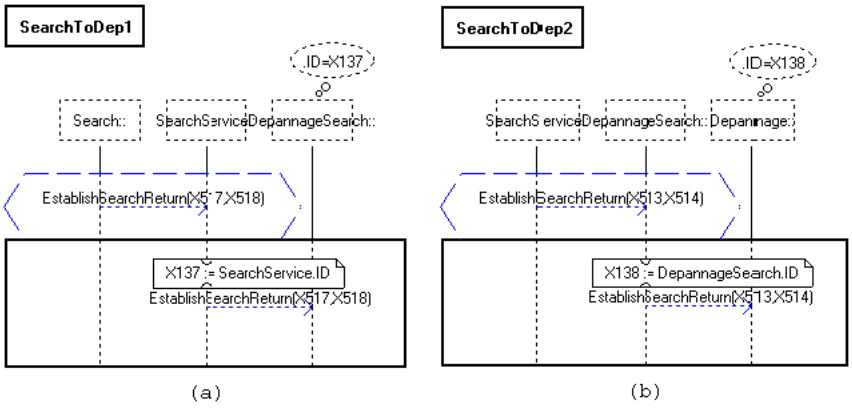

Fig. 7. Connectors between components Search and Depannage

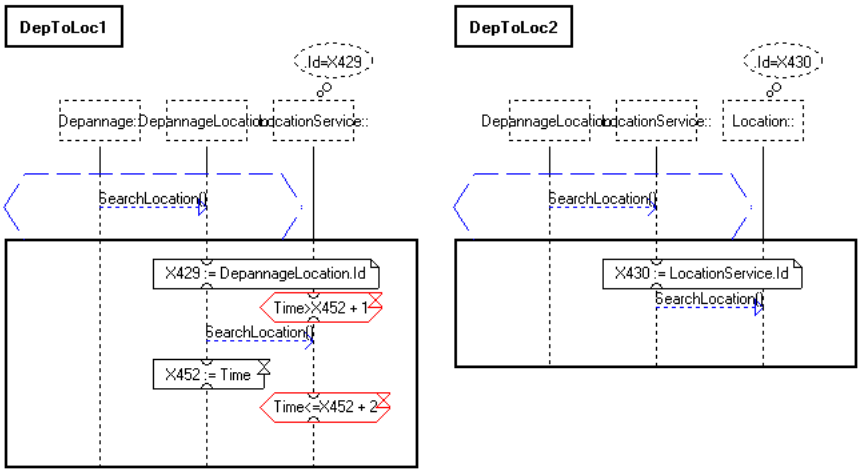

Fig. 8. Connectors between components Depannage and Location

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 also done by specifying universal LSCs defining the connection between components. Following the architecture diagram, these LSCs will 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 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. 1).

To specify the communication between two components following an architectural diagram, we have to construct two LSCs for each event. 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. 6(a),(b). Similar LSCs are also specified for the return event `EstablishSearchReturn` in Fig. 7(a),(b).

The connection between the components `Depannage` and `Location` is described in Figs. 8, 9. In these LSCs we also introduce time delay on the communication. The LSC `DepToLoc1` of Fig. 8 specifies that the method `SearchLocation` will take between 1 and 2 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 `x452` immediately after sending `SearchLocation` and adding the two

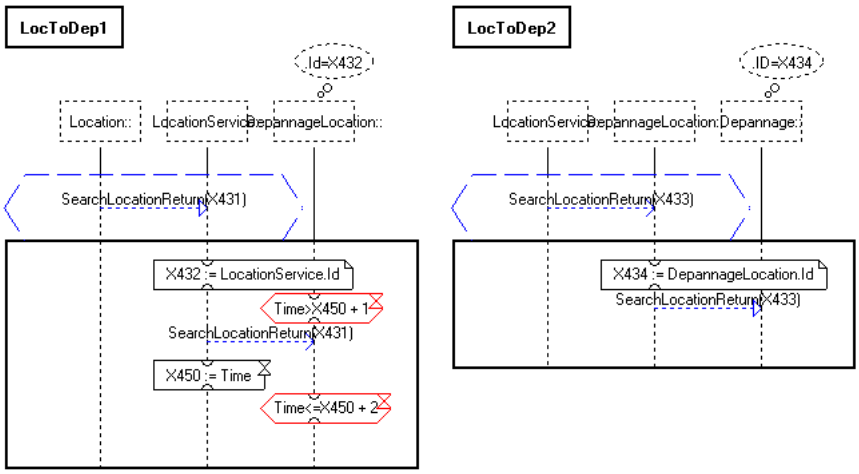

Fig. 9. Connectors between components Location and Depannage

hot conditions requiring `Time > x452 + 1` and `Time <= x452 + 2`. A similar requirement that the method `SearchLocationReturn` will take between 1 and 2 time units is specified in Fig. 9.

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

6 Simulation using play-out

Play-out allows a convenient way to debug requirements at an early stage and to detect problems in the design. For this purpose we can use anti-scenarios, behavioral requirements that are forbidden in the system. Consider the chart `NoQuickAnswer2` in Fig. 10. It specifies that whenever `SinglePhone3` makes a quick answer by sending the self message `UserAction(quickAnswer)` and after that `DepSearch1` sends the message `EstablishSearch Return(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. 11. 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

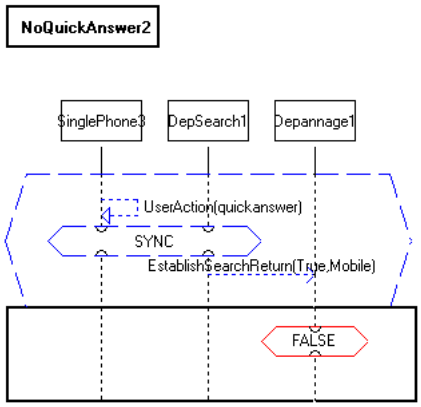

Fig. 10. A forbidden scenario - No connection to the vocal box of a mobile phone

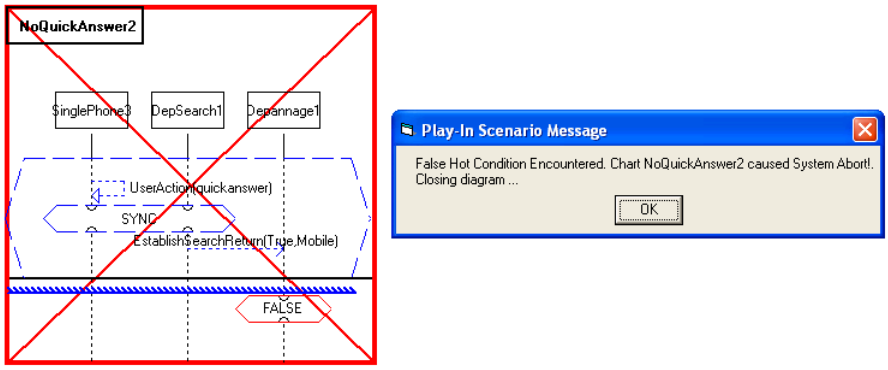

Fig. 11. Violation of a forbidden scenario during play-out

is detected it indicates a problem in the specification or the design of the service and should be looked into carefully to identify and fix the cause of the violation.

7 Verification using smart play-out

Smart play-out [7] uses verification methods, mainly model-checking, to execute and analyze LSCs. There are various modes in which smart play-out can work. In one of the modes smart play-out functions as an enhanced play-out mechanism, helping the execution to avoid deadlocks and violations. Thus, in this mode smart play-out utilizes verification techniques to run programs, rather than to verify them. In another 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.

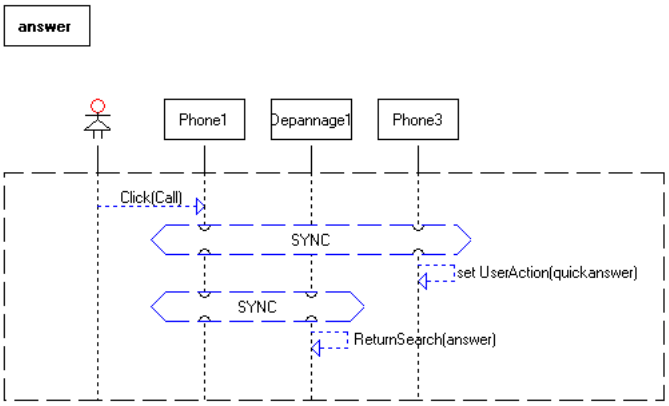

Fig. 12. An existential chart implying connection to the vocal box of a mobile phone

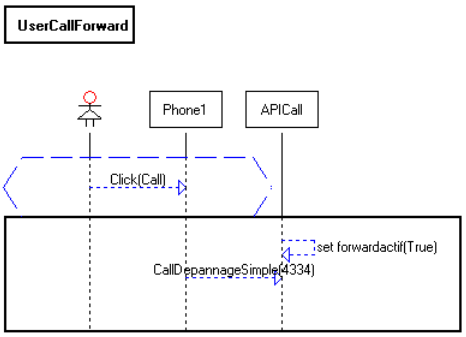

Fig. 13. A new feature of forwarding calls

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. A cleaner way would have been to specify these scenarios as anti-scenarios, as shown in Fig. 10. An enhancement to smart play-out is currently being developed to support this work-flow.

Consider the existential chart shown in Fig. 12. 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.

We then added a new feature to our telecommunication model, forwarding calls, shown in Fig. 13, applied smart play-out, and it found a way to satisfy

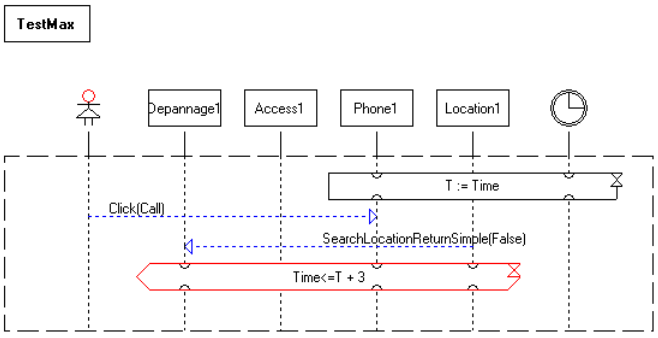

Fig. 14. Timing Requirements

the chart of Fig. 12. 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 [5].

The current version of smart play-out is still restricted in terms of the language features it supports. Thus to use it some restrictions should be made on the model: no symbolic-instances, and only one parameter for each signal. We are currently working on lifting these restrictions. We have also abstracted and simplified the model to avoid the well known state-explosion problem. In a similar manner we have verified also timed properties of the application, as specified in Fig. 14. The entire model and the reduced versions are all available in [5].

8 Related work and Future directions

Scenario-based specification is very helpful in early stages of development [1], and is used widely by engineers. A considerable amount of experience has been gained from it being integrated into the MSC ITU standard [13] and the UML [14]. The latest versions of the UML recognized the importance of scenario-based requirements, and UML 2.0 sequence diagrams have been significantly enhanced in expressive capabilities, inspired in part by the LSCs of [6]. In [8], 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.)

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 [3].

References

1. R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.
2. R. Castanet, A. Cavalli, P. Combes, P. Laurencot, M. MacKaya, A. Mederreg, W. Monin, and F. Zaidi. A multi-service and multi-protocol validation platform-experimentation results. In *TestCom*, volume 2978 of *Lect. Notes in Comp. Sci.*, pages 17–32. Springer-Verlag, 2004.
3. P. Combes, F. Dubois, W. Monin, and D. Vincent. Looking for better integration of design and performance engineering. In R. Reed, editor, *SDL Forum*, volume 2708 of *Lect. Notes in Comp. Sci.*, pages 1–17. Springer-Verlag, 2003.
4. P. Combes, F. Dubois, and B. Renard. An Open Animation Tool: Application to Telecommunications Systems. *Computer Networks*, 40(5):599–620, December 2002.
5. P. Combes, D. Harel, and H. Kugler. Supplementary material on the depannage application. http://cs.nyu.edu/∼kugler/Depannage/.
6. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. Preliminary version appeared in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99).
7. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Proc. 4^{th} Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, volume 2517 of *Lect. Notes in Comp. Sci.*, pages 378–398, 2002. Also available as Tech. Report MCS02-08, The Weizmann Institute of Science.
8. D. Harel, H. Kugler, and G. Weiss. Some Methodological Observations Resulting from Experience Using LSCs and the Play-In/Play-Out Approach. In *Proc. Scenarios: Models, Algorithms and Tools*, volume 3466 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2005.
9. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
10. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play In/Play-Out Approach. *Software and System Modeling (SoSyM)*, 2(2):82–107, 2003.
11. L. Logrippo and D. Amyot. *Feature Interactions in Telecommunications and Software Systems VII*. IOS Press, 2003.
12. R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, pages 83–100, Seattle, WA, 2002.
13. ITU-TS Recommendation Z.120 (11/99): MSC 2000. ITU-TS, Geneva, 1999.
14. UML. Documentation of the unified modeling language (UML). Available from the Object Management Group (OMG), http://www.omg.org.
15. Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.