Testing Scenario-Based Models*

Hillel Kugler[1], Michael J. Stern[2], and E. Jane Albert Hubbard[1]

[1] New York University, New York, NY, USA
kugler@cs.nyu.edu,
jane.hubbard@nyu.edu
[2] Yale University, New Haven, CT, USA
Michael.Stern@yale.edu

Abstract. The play-in/play-out approach suggests a new paradigm for system development using scenario-based requirements. It allows the user to develop a high level scenario-based model of the system and directly execute system behavior. The supporting tool, the Play-Engine has been used successfully in several projects and case-studies. As systems developed using this method grow in size and complexity, an important challenge is maintaining models that are well understood in terms of their behaviors and that satisfy the original intension of the system developers. Scenario-based methods are advantageous in early stages of system development since behaviors can be described in isolated fragments. A trade-off for this advantage, however, is that larger models comprising many separate scenarios can result in executable behavior that is difficult to understand and maintain. A methodology for facile testing of scenario-based requirements is needed. Here, we describe a methodology and supporting prototype implementation integrated into the Play-Engine for testing of scenario-based requirements. We have effectively applied the method for testing a complex model containing several hundred scenarios.

1 Introduction

Scenarios have been used in many approaches to describe system behavior [1,22,19], especially in the early stages of system design. This paper deals with a scenario-based approach that uses the language of live sequence charts (LSCs) [5] and the Play-Engine tool [11,12] and extends it to support the testing of scenario-based models.

One of the most widely used languages for specifying scenario-based requirements is that of message sequence charts (MSCs, adopted in 1996 by the ITU [24]), or its UML variant, sequence diagrams [23]. 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

* This research was supported in part by NIH grant R24 GM066969.

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 306–320, 2007.

live sequence charts (LSCs) [5]. Among other features, 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, that must be satisfied by at least one system run. Thus, existential charts 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 in which a certain behavior occurs. The distinction between mandatory (hot) and provisional (cold) behavior applies also to other LSC constructs, e.g., conditions and locations, thus creating a rich and expressive language. In addition to required behavior, LSCs can also explicitly express forbidden behavior ("anti-scenarios").

A methodology for specifying and validating requirements, termed the "play-in/play-out approach", and a supporting tool called the Play-Engine is described in [11,12]. Play-in is an intuitive way to capture requirements by demonstrating the behavior on a graphical representation of the system, while play-out is a method for executing LSC requirements, giving the feeling of working with an actual system. The expressive power of universal charts, based on the pattern "prechart implies main chart", forms the basis of an executable semantics for an LSC requirement model. In play-out mode, the Play-Engine monitors progress within charts and performs system events in the main charts for universal charts that have been activated, trying to complete all universal charts successfully. As the events are being executed, their effects are visualized via a Graphical User Interface (GUI), thus providing an animation of system behavior. The combination of an expressive specification language, a convenient method for capturing the requirements by playing them in on a GUI, and an executable framework that uses the same GUI for visualization, makes the Play-Engine a promising approach for modeling and system development, as shown by several projects [16, 15, 4, 9, 21].

In general, systems developed in any language are difficult to validate as they become large and complex, and a variety of verification and testing techniques have been developed to aid this process. For a number of reasons, the fragmented nature of scenario-based behavioral specifications makes large-scale modeling efforts that use languages and tools like LSCs and the Play-Engine particularly difficult to validate. First, unanticipated violations of the requirements may occur during play-out due to the interaction between several different LSCs. Furthermore, while the LSC language contains techniques for representing systems that contain significant levels of behavioral non-determinism, non-deterministic language structures increase the complexity of avoiding such LSC violations. Second, debugging LSC models is aided by the ability to visualize active LSCs during execution, showing graphically the progress along each active chart. However, visualization in the current Play-Engine implementation does not scale well for systems with hundreds of simultaneously active charts.

This work has been motivated by a project on modeling biological systems using tools for software and system design. Biological systems are in general very complex. Even the relatively small and well-defined subsystem we have focused on - VPC fate specification in the nematode C. *elegans* – involved constructing and testing a model of several hundred LSCs [17,15]. The size and complexity of this model (which is significantly larger than that of other Play-Engine models and also of most models used in other scenario-based approaches reported in the literature) required us to address several issues that can typically be ignored when dealing with "text-book" examples or small case studies, and to find solutions that are efficient and scale well as the system grows. In this paper we propose a testing-based method to address these challenges. The main strengths of our approach is the ability to apply it to large LSC models, a smooth integration into the Play-Engine tool and development process, and a user-friendly presentation of the test results.

2 LSCs and Play-Out Definitions

This section reviews the main ideas and definitions underlying the language of live sequence charts and the play-out execution method. For a detailed and systematic treatment the reader is referred to [5,11,12]. The main strength of live sequence charts 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 described 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.

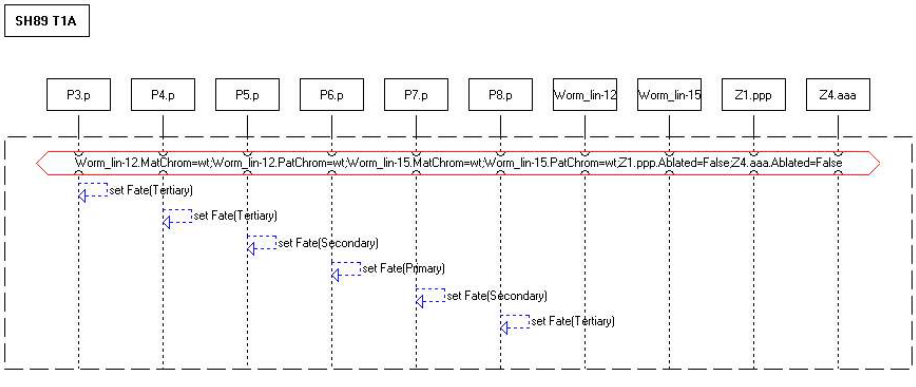

Fig. 1. Example of an existential chart

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. 1 is an existential chart as denoted by the dashed borderline. The scenario starts with a condition - an assertion that requires specific property values for some of the objects. The condition implies a synchronization over all participating objects. The scenario proceeds by objects sending (self) messages. For this chart to be satisfied all messages specified should occur but the ordering between them is not restricted since they appear on different object lines. This interpretation of existential charts is useful in early system design for demonstrating possible behavior, or in biological modeling for capturing experimental observations, but is too weak in terms of expressive power to define causality relations and provide executable semantics.

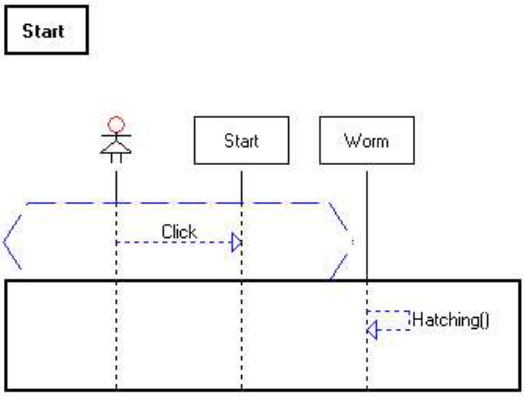

Fig. 2. Example of a simple universal chart

For this reason, 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. 2. Universal charts are denoted by a solid borderline. According to the LSC in Fig. 2, if the `Click` method is sent from the user to the Start object, as specified in the prechart (dashed hexagon), the `Hatching` method 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 `Click` method must be eventually followed by a `Hatching` method.

The LSC language is rich and supports many constructs; a flavor of the language is demonstrated using a more complex universal chart appearing in Fig. 3. The prechart does not include a single message as in Fig. 2 but rather describes a more complex behavior. The condition labeled `SYNC` in the prechart restricts the ordering of events such that the prechart is satisfied and the main chart

activated only if the `Hatching` method is followed by a `setFate(Primary)` message. The two `VPC` instances in Fig. 3 are symbolic instances, they represent the behavior of the VPC class, which can contain many concrete instances (in our biological model there are actually six VPCs). The behavior captured in Fig. 3 is that if the prechart holds, the relevant VPC will send the method `LS` to its right neighbor. An assignment appearing at the beginning of the main chart and binding conditions for the symbolic VPC instances (appearing in the ellipses) are used to capture this required behavior.

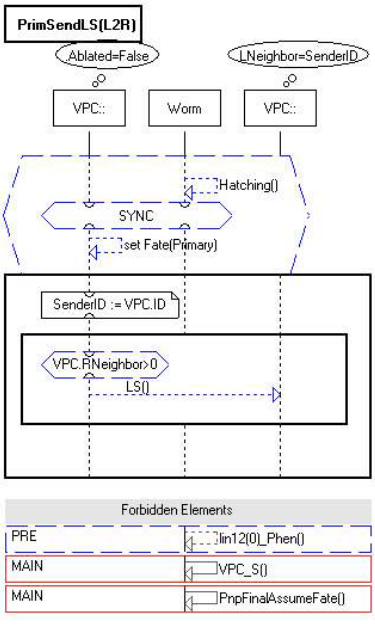

Fig. 3. Example of a more complex universal chart

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.

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

An **LSC specification** is a pair

$$LS = \langle M, mod \rangle,$$

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

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

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

For a universal chart, $mod(m) = universal$, 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 $LS = \langle M, mod \rangle$, written $S \models LS$, if:*

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

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

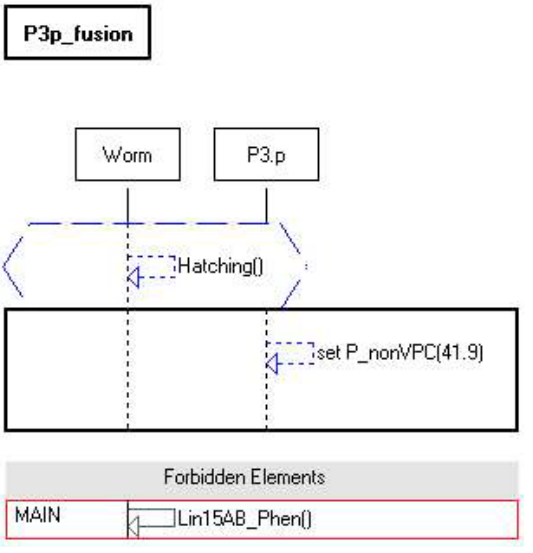

Fig. 4. Activated LSC

Play-out is a method that attempts to execute an LSC specification LS directly. This contrasts with a process in which a system S is constructed manually to satisfy the specification LS. The idea behind play-out is actually simple, yet leads to a surprisingly useful execution method. 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. In our example, if the user has clicked on the `Start` button, the prechart of the LSC in Fig. 2 reaches its maximal locations and, as a result, activates the main chart. The Play-Engine, in response, will execute the method `Hatching` appearing in the main chart, thus fulfilling the requirement of this

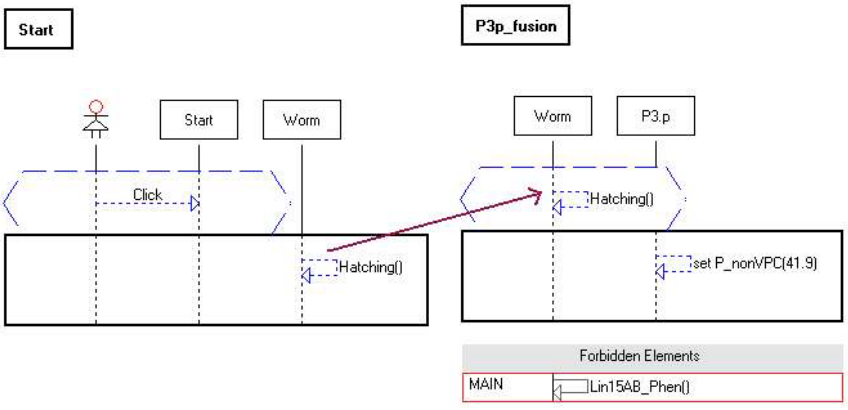

Fig. 5. Cascade of LSC activation

universal chart. Executing the method `Hatching` as specified in the main chart of Fig. 2, in turn will activate the LSC of Fig. 4, since the `Hatching` method is the only event appearing in the prechart of Fig. 4. The Play-Engine will then execute the message `setP_nonVPC(41.9)`. This is a typical situation in play-out, where executing a message in one LSC activates a new LSC, creating a cascade of events, as illustrated for our example in Fig. 5. 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.

While play-out is an effective method for executing LSCs, it 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. Therefore, a mechanism is needed to test whether specifications have, in fact, been satisfied. 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 latter may be due to an unforeseen interaction between several different universal LSCs. A blatant example of universal LSC violation is when two charts are actually contradictory. One such simple example consists of two LSCs, both being activated by the same message `a` appearing in the prechart, and containing the two messages `b` and `c` in the main chart. One chart requires that the ordering between the events is `b,c` while the other chart requires that the ordering is `c,b`. A less blatant violation can occur if play-out is unable to execute the requirements correctly. The play-out mechanism of [11] 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 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 the challenge of arbitrary "dead ends" occurring in instances of non-determinism, [8] introduces a technique for executing LSCs, called *smart play-out*. It takes a significant step towards removing the sources of nondeterminism during execution, proceeding in a way that eliminates some of the dead-end executions that lead to violations. Smart play-out uses model-checking to execute and analyze LSCs. Smart play-out, like more ambitious synthesis methods, does not currently scale to handle large models, especially when using the full-range of LSC constructs, including symbolic instances [20] and time [11]. For this reason we have focused our work on using play-out and developing testing methods to detect those cases in which play-out does not execute the LSC requirements correctly. After the problems are detected and fixed, testing is used to increase the confidence in the correctness of play-out for the new model.

3 Execution Configurations

The testing of system behavior often requires the analysis of specific variations of distinct behavior fragments. These variations can be represented in related sets of "execution configurations". The execution configuration allows the user to specify the LSCs that should be considered by the Play-Engine while executing a model in play-out mode. An example of the execution configuration dialog is shown in Fig. 6. The dialog is composed of three sections: the first allows the user to select the participating universal LSCs, the second to select 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. 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.

We have extended the Play-Engine to support handling and saving multiple execution configurations for the same model, allowing the user to select among them an active execution configuration. This allows the simultaneous maintenance of several variants of an LSC model, differing in the modeling of certain aspects of the system, values of various parameters, or corresponding to different levels of abstraction. The testing of these alternative models can be helpful in developing the system, in testing different possible implementation alternatives, and in fine tuning parameters.

Our extension allows the user to add, edit and delete execution configurations. There is always exactly one active execution configuration, specified by the user, which is used in play-out mode to determine the LSCs participating in the

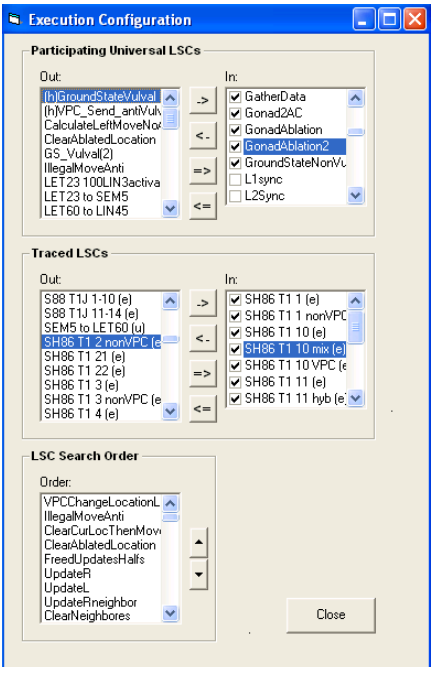

Fig. 6. The execution configuration dialog

execution, the traced LSCs and the search order. For certain applications, the ability to change the execution configuration while in the middle of play-out may be an interesting and useful feature. Currently, however, changing the active execution configuration is not allowed while in the middle of a play-out session. To set a new active execution configuration, play-out must first be stopped, the new active execution configuration set, and only then can play-out be reactivated.

4 The Testing Environment

We have created a testing tool that allows batch runs. Using the tool, the user designates a test plan that includes sets of initial configurations (also called jumpstarts) paired with iteration numbers, as shown in Fig. 7. The user specifies when to end an iteration by designating a specific universal LSC; upon the successful completion of that LSC, the iteration ends and a new one begins. This termination LSC, therefore, must be one that will be activated and successfully completed in every run, thus ensuring that the iteration will end and that the test plan execution can advance.

For each pair of initial configuration and iteration, the user can specify as part of the test plan which execution configuration to use. If no execution configuration is selected for a certain pair of initial configuration and iteration, the last active execution configuration is used. The ability to specify an execution

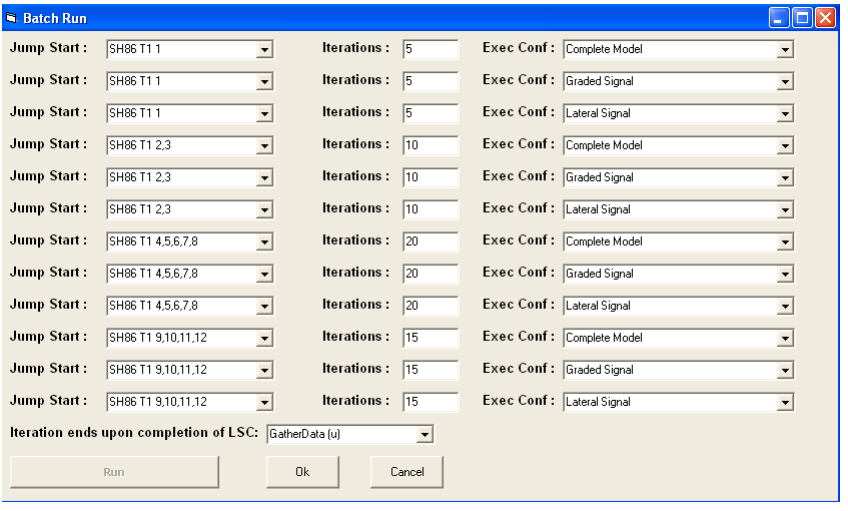

Fig. 7. Batch Mode Dialog

configuration permits testing several variants of a model as part of the same test plan, allowing convenient comparison of the test results.

We have added a number of additional features that are convenient for testing. One feature allows the user to develop a scenario-based model that does not depend on the search order or execution policy. This is effected by selecting a mode in which the LSC order is randomly permuted rather than fixed by the active execution configuration. This feature can help uncover problems that were not observable while running the model under a specific execution order.

Another set of features permits the user to allow running an unspecified number of iterations in order to satisfy a specific set of existential charts and an initial configuration. Under these conditions, the tool will run iterations until all of these existential charts are satisfied. Since it is often unknown a priori whether the model is indeed capable of satisfying all charts, an additional feature was implemented to avoid getting stuck in one initial configuration: the tool can be made to alternate between initial configurations that still have not satisfied all of their target charts, until all tests and charts are satisfied or until the tool is stopped by the user. The user can also set a maximum bound on the runtime or number of iterations after which the execution will be stopped, even if all of the existential charts have not been satisfied. These features are particularly necessary for the nondeterministic / probabilistic aspects of a model, where satisfying the relevant existential charts from a given initial configuration typically requires running multiple iterations.

Multiple execution configurations are also helpful for models that have large sets of existential charts that need to be satisfied, but only smaller subsets of these charts that need to be monitored per test. The user can create

corresponding execution configurations that vary in the list of traced charts and use them in the test plan. Thus the computational price of monitoring many charts that are not relevant is reduced.

5 Test Recording Methods

While developing an effective testing methodology and tool, it is important to store sufficient information that will allow users to examine relevant test results carefully. Adequate recorded information, combined with user-friendly ways to display it, can provide for effective debugging of large scenario-based models. We have developed three features that help gather, record and display test information for effective understanding of simulation results: (1) a text log of important events that occur during a run; (2) an excel worksheet for recording key properties of objects during a run; and (3) a full run trace.

First, there are several types of events that are stored in a text log file for each of the runs. In the default mode, these include a list of all existential charts that have been traced to completion and universal charts that have been violated. In a more detailed mode, we additionally record when universal charts become activated (the prechart has completed successfully), and when universal charts are completed successfully. In addition, general information on the test plan, execution configurations used, and execution time is stored at the beginning and end of this log file.

Second, links to an excel worksheet have been enabled to store additional information from a simulation. The Play-Engine supports basic excel functions, which can be referred to in new or existing LSCs at specific time points to record relevant information, e.g., values of properties for participating objects. The user can determine exactly which information will be recorded, and at which relevant time points during a run. Typically, information for each iteration will be written at a distinct location in the excel sheet, for example in a new row. These results can then easily be examined in the excel file, and scripting options for excel and connection to existing tools may be used to allow manipulation and analysis of the collected data.

Finally, a full trace of the runs in batch mode can be recorded and saved. These traces can then be replayed after the test execution has ended to examine interesting results. A recorded run can be executed without the relevant LSCs participating, since the recorded run stores all event occurrences. All of the events are displayed on the graphical user interface (GUI) as in normal play-out mode, but the execution of recorded runs is much faster, allowing users to examine interesting traces more efficiently.

6 Applications and Testing Methodology

We have effectively applied our method to a complex biological model [16] containing several hundred scenarios, and are currently evaluating the method on a telecommunication application designed by France Telecom [4]. In this section we

describe the basic steps performed in testing a model and outline some methodological guidelines derived from our initial experience. Some of the testing tool features have been designed with a goal of automating specific steps.

The batch run mode format for testing differs in certain fundamental ways from the standard manual play-out mode, thus necessitating some system reconfiguration prior to testing. Some of these adjustments are enabled by features of the Play-Engine tool itself, while others require altering some of the LSCs. The major difference between these two modes of play-out stems from the nature of the interactions between the system and its environment. The Play-Engine supports specification of reactive systems, providing an explicit distinction between the system and its environment. Environment objects are either an external user, a more abstract "environment", any object explicitly designated as external, or the global clock. In a manual play-out session, the user performs the events initiated by these environment objects, while the play-out execution engine performs all system events in response. By contrast, testing is performed in batch run mode with no user interaction, so it requires working with a closed system, where the environment is modeled and considered part of the system. To enable this closed-environment testing, the user creates interface system objects which appear in the LSCs instead of the environment objects, and all external objects are changed to system objects. In testing mode, time progresses automatically by the play-out mechanism according to the play-out execution semantics [10]. For testing behaviors that in manual play-out mode require user interaction due to external events, the interaction must be modeled explicitly using LSCs, by either modifying the existing LSCs or constructing new ones. Various initial system configurations can be pre-set in jumpstarts, obviating the need for the user to manually play these in. Jumpstarts can also serve as shortcuts in manual play-out mode as well.

Once the reconfigurations associated with system-environment interactions are in place, the user next prepares a test plan containing pairs of initial configurations and iteration numbers as described above. If there are several variants of models or large sets of traced LSCs, then appropriate execution configurations are created and used for the test plan. The user runs the test plan, a task that can typically run for several hours until all iterations are completed. Due to the size and complexity of current scenario-based models, the testing effort can also be distributed between several machines, each running a Play-Engine version and performing tests independently.

Next, the test results are examined by the user. Chart violations reported in the log file are worthwhile examining first, since these violations usually indicate problems in the model that must be fixed before investing much time in analyzing the other detailed results or viewing detailed simulations. Additional potential violations for universal charts can occur if the main chart was not completed successfully before the iteration ended. This may occur due to a real problem or just as a consequence of arbitrarily stopping the run when the designated LSC marking the end of the iteration is completed.

After several rounds of correcting the LSCs in the model and running the test suite, the user converges to a model with no test results showing violations of universal charts. The log file is next examined to determine if, for all tests, all relevant existential charts are satisfied. For existential charts that have not been satisfied, the user tries to determine whether they represent a possible outcome that was not observed (possibly due to probabilistic choices and a limited number of iterations) or an impossible outcome. The user can run this test again with a larger number of iterations if the former reason is suspected.

Runs that satisfy existential charts provide evidence of desirable behavior. To strengthen one's confidence that these are indeed the expected results, it is useful to examine the recordings for some of these runs and observe the simulation via the GUI. Runs that do not satisfy any existential charts represent potential new behaviors. In software or system development, the user should examine the run to decide if it is indeed a desirable behavior, and, if so, an appropriate matching existential chart can be added to the test suite. Otherwise the LSC model should be corrected to prevent this run. In the modeling of biological systems, if this run is clearly at odds with previous biological observations, the model must be corrected. Otherwise, this is a possible prediction of the model that can be tested experimentally. When the set of existential charts is rich enough, it is easier to find such interesting runs and then examine them more carefully.

7 Related Work

Scenarios have been recognized to be useful as part of the testing process. In [18] a methodology and tool called TestConductor is introduced that uses a subset of LSCs to monitor and test a UML model whose behavior is specified using statecharts. TestConductor is integrated into the Rhapsody tool [14]. A mapping and automatic translation for generating test descriptions in the standard test description language TTCN-3 directly from message sequence charts is described in [6]. Story boards are an alternative means for capturing scenarios; an approach utilizing these scenarios for testing, integrated into the Fujaba tool, is described in [7]. An approach for modeling use cases and scenarios in the abstract state machine language with applications for testing is described in [2]. Common to all of these approaches is that the scenarios serve as a requirement language, while the model or system that is to be tested is described in another language, e.g., statecharts or code. In this paper we use the same scenario-based language (LSCs) for describing the system model and the requirements. In fact, in our approach there is not such a clear separation between the system and the requirements, since the executable system model is directly based on running the requirements. Model-based testing [13, 3] is a general approach in which a model of the system under test is used for supporting and automating common testing tasks such as test generation, test execution and test evaluation. A major goal is to reduce the manual effort involved in the testing activity.

Acknowledgments

We would like to thank Dan Barak for his help in the implementation of the testing module and its integration into the Play-Engine, and Na'aman Kam for helpful discussions on the need for enhanced automation in the analysis of complex biological models. This research was supported in part by NIH grant R24 GM066969.

References

1. D. Amyot and A. Eberlein. An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. *Telecommunications Systems Journal*, 24(1):61–94, 2003.
2. M. Barnett, W. Grieskamp, Y. Gurevich, W. Schulte, N. Tillmann, and M. Veanes. Scenario-Oriented Modeling in AsmL and its Instrumentation for Testing. In *Proc. 2nd Int. Workshop on Scenarios and State Machines (SCESM'03)*, 2003.
3. M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a Tool Environment for Model-Based Testing with AsmL. In *Proc. 3rd Int. Workshop on Formal Approaches to Software Testing (FATES '03)*, volume 2931 of *Lect. Notes in Comp. Sci.*, pages 252–266. Springer-Verlag, 2004.
4. P. Combes, D. Harel, and H. Kugler. Modeling and Verification of a Telecommunication Application using Live Sequence Charts and the Play-Engine Tool. In *Proc. 3rd Int. Symp. on Automated Technology for Verification and Analysis (ATVA '05)*, volume 3707 of *Lect. Notes in Comp. Sci.*, pages 414–428. Springer-Verlag, 2005.
5. 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).
6. M. Ebner. TTCN-3 Test Case Generation from Message Sequence Charts,. In *Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04:WITUL)*, 2004.
7. L. Geiger and A. Zündorf. Story driven testing - SDT. In *Proceedings of the fourth international workshop on Scenarios and state machines: models, algorithms and tools (SCESM '05)*, pages 1–6. ACM Press, 2005.
8. 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.
9. 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.*, pages 26–42. Springer-Verlag, 2005.
10. D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *Proc. 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, Fort Worth, Texas, 2002.

11. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
12. 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.
13. A. Hartman and K. Nagin. The AGEDIS tools for model based testing. In R.L. Grossman, A. Nerode, A. Ravn, and H. Rischel, editors, *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '04)*, pages 129–132. ACM Press, 2004.
14. Rhapsody. http://www.ilogix.com/, 2006.
15. N. Kam. *Formal Modeling of C. elegans Vulval Development: A Scenario-Based Approach*. PhD thesis, Weizmann Institute, 2006.
16. N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E.J.A. Hubbard, and M.J. Stern. Formal Modeling of C. elegans Development: A Scenario-Based Approach. In Corrado Priami, editor, *Proc. Int. Workshop on Computational Methods in Systems Biology (CMSB 2003)*, volume 2602 of *Lect. Notes in Comp. Sci.*, pages 4–20. Springer-Verlag, 2003. Extended version appeared in Modeling in Molecular Biology, G.Ciobanu (Ed.), Natural Computing Series, Springer, 2004 .
17. N. Kam, H. Kugler, L. Appleby, A. Pnueli, D. Harel, M.J. Stern, and E.J.A. Hubbard. Hypothesis Testing and Biological Insights from Scenario-Based Modeling of Development. Technical report, 2006.
18. M. Lettrari and J. Klose. Scenario-based monitoring and testing of real-time UML models. In *4th Int. Conf. on the Unified Modeling Language, Toronto*, October 2001.
19. S. Leue and T. Systä. Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2005.
20. 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.
21. OMEGA - Correct Development of Real-Time Embedded Systems. http://www-omega.imag.fr/.
22. S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Software Engin. Methods*, 13(1):37–85, 2004.
23. UML. Documentation of the unified modeling language (UML), 2006. Available from the Object Management Group (OMG), http://www.omg.org.
24. Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.