OMEGA

Correct development of Real-time systems
IST-2001-33522

Title: Planning a production line with LSCs

Authors: Hillel Kugler and Gera Weiss

Date: 12 February 2005

Document Version: WP23/D2.3.4, Annex 8

Status: Final

Confidentiality: Public

Note: A preliminary version appeared as Weizmann Institute Technical Report MCS-04-05

Abstract

We display how LSCs can be used to analyze a production line systematically and synthesize a scheduler from the model. This work demonstrates how the features of a scenario based object oriented modeling language are exploited in this domain. It also shows advantages of the play-in/play-out methodology combined with verification methods.

Planning a production line with LSCs *

Hillel Kugler and Gera Weiss †

February 13, 2005

Abstract

Using the Cybernetix case study, we display how LSCs can be used to analyze a production line systematically. This work demonstrates how the features of a scenario based object oriented modeling language are exploited in this domain. It also shows advantages of the play-in/play-out methodology.

1 Introduction

Scenario-based modeling has become a common practice in many domains. With particular impact on the telecommunication and software engineering industries. Scenario based techniques are employed to capture requirements of reactive systems, to capture use cases in system documentation, to specify test cases, or to visualize runs of existing systems. They are often used to represent concurrent systems that interact via message passing or method invocations.

There is an increasing practical interest in scenario-based notations and techniques. The International Telecommunications Union has standardized Message Sequence Charts (MSC) notation [10] as a language for specifying distributed protocols. Sequence diagram notations play a dominant role within the UML software design methodology [12]. Currently, there is active research on extending scenario notations and their use such as the inclusion of real-time and probabilistic information, the inclusion of modalities, and the usage as play-in/play-out scenarios. A survey on scenario-based notations and methods appears in [3]

In this report, we show that scenario-based modeling can be effectively used in the process of evaluating designs and production schedules for an industrial production line. We take a smart-card personalization machine as an example and use the LSCs scenario based, formal graphical language to model it. We argue that the scenario-based nature of the modeling language allows using a design process and methodology that is intuitive and less error prone.

When designing production lines, such as the HPX machine manufactured by Cybernetix, one needs the ability to quickly come up with executable models that can be analyzed and simulated. Design options are numerous while time to marked presses. It is also important that models can be easily perturbed to check the affect of different variations. In this report, we argue that the LSCs scenario based language together with the play-in/play-out methodology supported by the PlayEngine prototype tool deliver these needs without compromising ease of use and accessability to non experts.

The paper is structured as follows. We begin with two sections introducing the reader with the language of LSCs and with the smart-card personalization machine case study. We then provide a detailed description of the LSC model that we have constructed for the case study.

*This research was supported in part by the European Commission projects AMETIST (IST-2001-35304) and OMEGA (IST-2001-33522) and by the John von Neumann Minerva Center for the Verification of Reactive Systems.

†Weizmann Institute of Science, {hillel.kugler,gera.weiss}@weizmann.ac.il

We conclude with a section explaining our method of extracting schedules from an LSC model and a section containing some conclusions.

2 LSCs, play-In/play-Out and the Play-Engine

We are adopting an inter-object, scenario-based modeling approach, using the language of live sequence charts (LSCs) [4] and the play-in/play-out methodology supported by the Play-Engine modeling tool [7].

The decision to take this approach, rather than the state-based one, emerged from the consideration of how to best represent the personalization machine formally, and how to best carry out the formalization process. LSCs constitute a visual formalism for specifying sequences of events and message passing activity between objects. The language allows to specify scenarios of behavior that cut across object boundaries and exhibit a variety of modalities, such as scenarios that can occur, ones that must occur, ones that may not occur (called anti-scenarios), ones that must follow others, ones that overlap with others, and more.

Technically, there are two types of LSCs, universal and existential. The elements of LSCs (messages, locations, conditions, etc.) can be either mandatory (called hot in LSCs terminology) or provisional (called cold). Universal charts are the more important ones for modeling, and comprise a pre-chart and a main chart, the former triggering the execution of the latter. Thus, a universal LSC states that whenever the scenario in the pre-chart occurs (e.g., the user has flipped a switch), the scenario in the main chart must follow it (e.g., the light goes on). Thus, the relation between the pre-chart and the chart body can be viewed as a dynamic condition-result: if and when the former occurs, the system is obligated to satisfy the latter.

Play-in/play-out is a recently developed process for modeling in LSCs, with which one can conveniently capture inter-object scenario-based behavior, execute it, and simulate the modeled system in full. The play-in part of the method enables people who are unfamiliar with LSCs to specify system behavior using a high level, intuitive and user-friendly mechanism. The process asks that the user first build the graphical user interface (GUI) of the system, with no behavior built into it. The user then 'plays' the GUI by clicking the graphical control elements (in electronic systems these might be buttons, knobs, and so on), in an intuitive manner, in this way giving the engine sequences of events and actions, and teaching it how the system should respond to them.

As this is being done, the Play-Engine continuously constructs the corresponding LSCs automatically. While play-in is the analogue of writing programs, play-out is the analogue of running them. Here the user simply plays the GUI as he/she would have done when executing the real system, also by clicking buttons and rotating knobs, and so on, but limiting him/herself to end-user and external environment actions. As this is going on, the Play-Engine interacts with the GUI and uses it to reflect the system state at any given moment.

The scenarios played in using any number of LSCs are all taken into account during play-out, so that the user gets the full effect of the system with all its modeled behaviors operating correctly in tandem. All specified ramifications entailed by an occurring event or action will immediately be carried out by the engine automatically, regardless of where in the LSCs it was originally specified. Also, any violations of constraints (e.g., playing out an anti-scenario) or contradictions between scenarios, will be detected if attempted. This kind of integration of the specified condition-result style behavior is most fitting for modeling industrial manufacturing systems as objects participate in different scenarios.

Scenario based languages, such as LSCs, fit very well into an object oriented framework. In particular the Play-Engine is fully object oriented. In the domain of structuring production lines, this feature proves very useful because the user can form alternative designs by connecting objects differently.

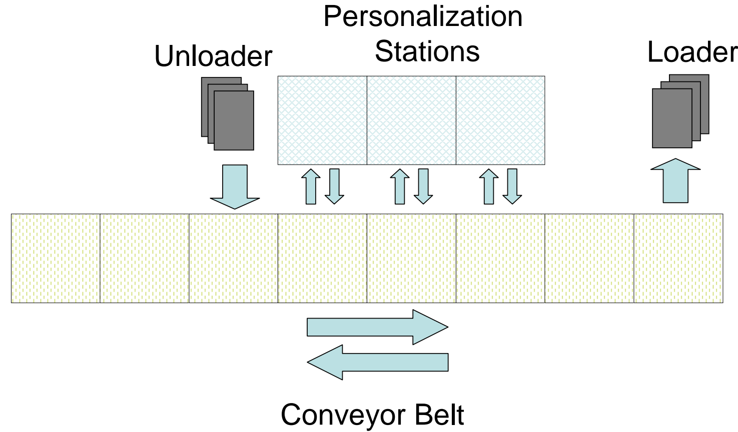

Figure 1: Schematic illustration of the personalization machine

3 The Smart-Card Personalization Machine

This section contains a short description of the smart-cart personalization machine. For a more comprehensive description see [2].

Cybernetix is manufacturing machines for smart card personalization. These machines take piles of blank smart-cards as raw material, program them with personalized data, print and test them. The machines have a throughput of thousands of cards per hour. It is required that the output of cards occurs in a predefined order. Unfortunately, some cards are defective and they have to be discarded, but without changing the output order of personalized cards. Decisions on how to reorganize the flow of cards must be taken within fractions of a second, if no production time is to be lost. The aim of this case study is to model the desired production requirements, the timing requirements of operations of the machine and on this basis synthesize the coordination of the tracking of defective cards. The goal is to maximize the throughput of the machine under certain error assumptions. Another design objective, specified by Cybernetix, is to shorten the machine, i.e. use less slots. This means that we whould like to show that it s possible to handle all errors using the minimal number of belt slots.

We handle a simplified version of the case study proposed by Cybernetix. Fig. 1 shows a schematic overview of the personalization machine that we discuss in this paper. Cards are transported by a Conveyer belt. There are three personalization stations where cards can be personalized. The conveyor belt is nine positions long. The Unloader puts blank cards on the belt. The Loader removes personalized cards from the belt. The order in which the cards are loaded from the belt should be same as the order in which they were personalized.

The conveyer can move a step to the right or a step to the left each takes one time unit. The conveyor belt cannot move while cards are unloaded onto the belt or loaded from the belt or taken up/down to/from a personalization station. Unloading and loading can be done in parallel. Unloading and loading takes 2 time units. If after a conveyer move, a blank card is under a personalization station, the card might be taken up to the personalization station. This operation takes one time unit. The personalization of the card will start immediately. The personalization of a card takes five time units. After personalization a card can be taken down, an operation that takes another time unit. The goal is to find an optimal schedule to maximize the throughput of the machine (number of personalized cards per time unit).

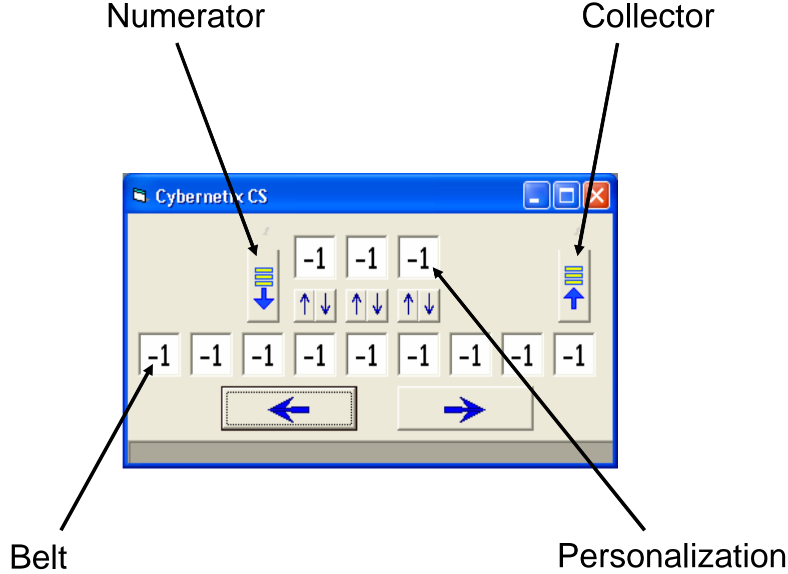

Figure 2: Mockup GUI for the smart-card personalization machine

4 LSCs model

4.1 Object Model

The personalization machine is modeled by the following objects: belt slots, personalization slots, a numerator and a collector. The visual appearance of these objects is shown in Fig. 2. We also defined one internal object named Controller.

Each belt and personalization slot has an attribute named *card*. This attribute represents the card in this slot. The legend for the *card* attribute is as follows: -1 means empty slot, 0 means that there is a blank card in the slot and a positive number indicates the identification of an initialized card.

In addition, the personalization slots and the three belt slots below them have another attribute called *PersId*. This attribute is used to pair these objets. It is a constant attribute set at design time such that the *PersId* of matching objects is equal.

The numerator object is used to record the last initialized card. Its *nextCard* attribute is incremented whenever a card is personalized. The collector records the identification of the last unloaded card. When a card is loaded, the recorded information is used to ensure order at the output pack.

Note that the form in Fig. 2 contains some buttons. These are standard buttons used to trigger scenarios for testing and manual operation. In order to enable symbolic formulation of generalized scenarios, the buttons between a personalization station and its belt has the *PersId* attribute set to match the corresponding personalization and belt slots.

We also defined some classes: Pers.Class, Belt.Class, Up.Class, Down.Class. These classes wrap their respective objects with a standard interface to allow symbolic references to personalization and belt slots and the pairs of buttons between them. We will use this abstraction later to define generic scenarios [9].

The scenarios of the system are grouped into use cases which serve as organizational units. In the following subsections we describe the different scenarios according to this grouping.

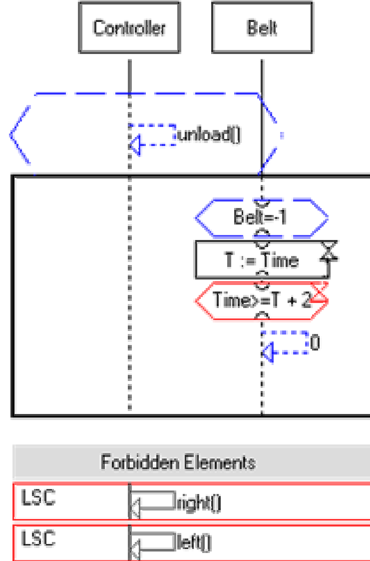

Figure 3: Unload Scenario

4.2 System Dynamics use case

System dynamics scenarios model the actual dynamics of the system. There are seven scenarios in this category: right, left, up, down, load, unload and personalization.

The unload scenario given in Fig. 3 models the process of taking a card from the pile of blank cards. The scenario is triggered by an *unload*() controller command. It quits instantly if the belt slot under the loader is not empty. Otherwise, after two time ticks, an empty card is put on that slot. No belt movements are allowed during this scenario. This is specified by designating $right$ and $left$ messages as forbidden while the unload chart is active.

The right scenario given in Fig. 4 models the movement of the belt to the right. The scenario is triggered by an *right*() controller command. It exists with no action if the rightmost belt slot is not empty. Otherwise, after one time tick, the *card* attribute of every belt slot is copied to its right neighbor using a local variable *crd*. All controller commands are forbidden while this scenario is active. The left scenario is almost identical to the right scenario and therefore omitted from this description.

The up scenario given in Fig. 5 defines how cards are taken up to the personalization stations. This is a generic scenario relevant to all the personalization stations. The chart is triggered by an *up*(*pid*) command issued by the controller. This is a parametric command where the controller specifies the identification of the station involved. An actual value for the parameter allows the classes to be associated with concrete instances. In our case, a specific personalization slot together with its associated belt slot (paired by the $persID$ attribute). If the personalization slot is not empty or the belt slot does not contain a blank card - the scenario is ended. Otherwise, after one time tick, the belt slot is emptied and the blank card is positioned at the personalization slot. Belt movements are not allowed during this process. The down scenario is similar and thus omitted from this description.

The personalization scenario is given in Fig. 6. It starts when a blank card is put on a personalization slot. The first (left) object of type Pers.Class is instantiated by this message. The *nextCard* attribute of the numerator is copied to a local variable *crd*. The function *nxt* which increases a card identity by one is applied to *crd* and the result is copied back to *numerator.nextCard*. The effect of this procedure can be stated by the formula $crd = numerator.nextCard{+}{+}$.

Note the sub-chart denoted by black box around the first two steps of the scenario. This

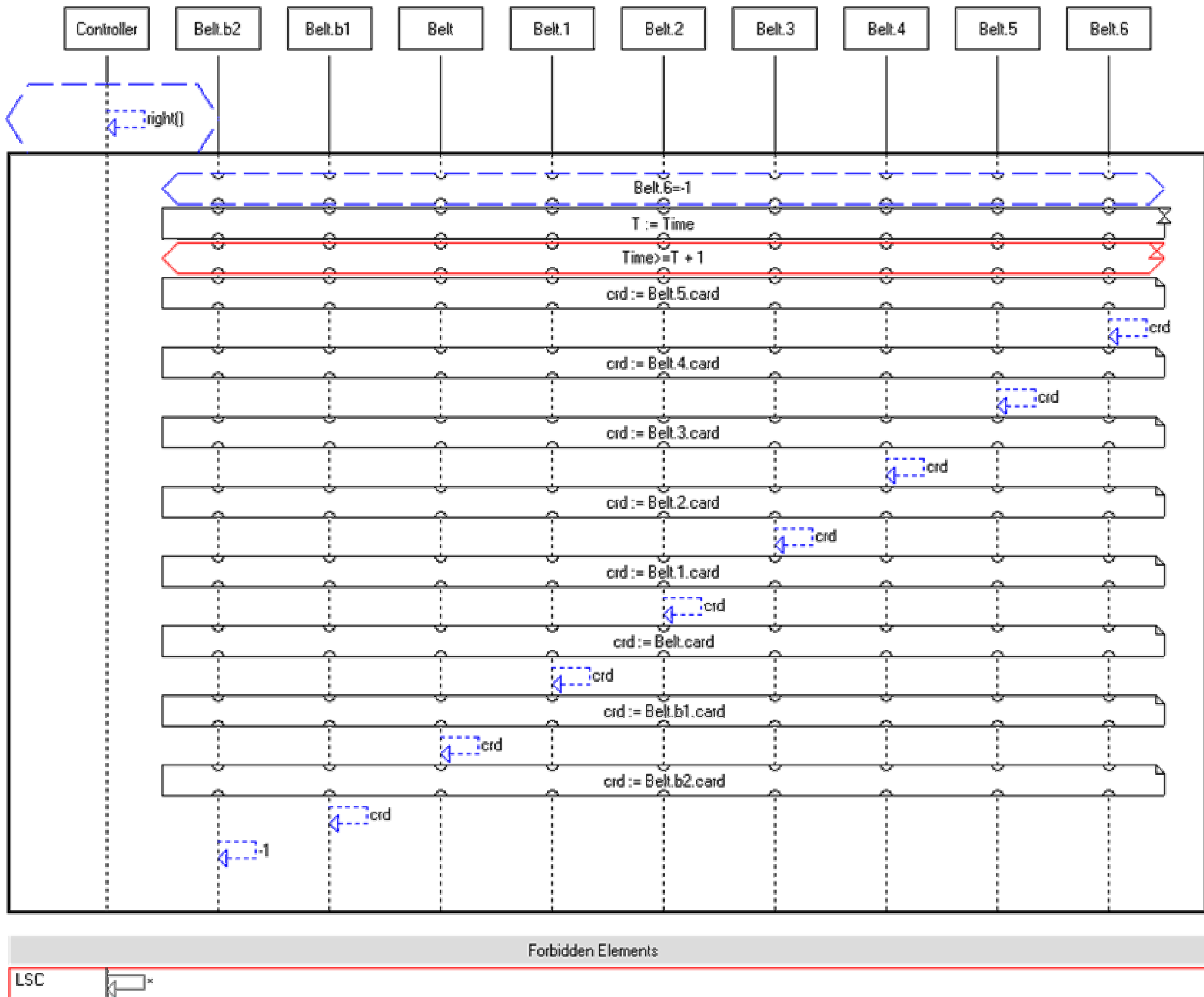

Figure 4: Right Scenario

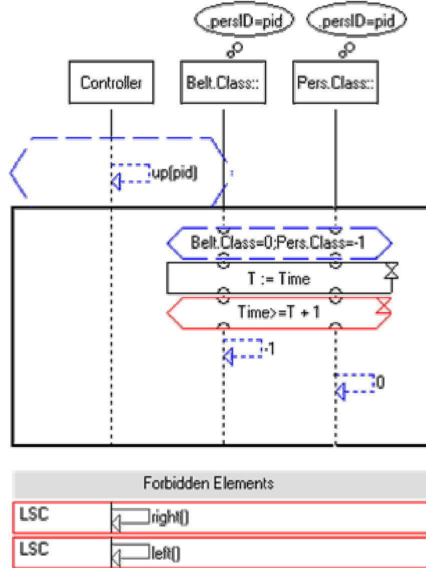

Figure 5: Up Scenario

sub-chart is used to exclude other copies of the scenario to share the same execution segment. Mutual exclusion is forced by disallowing the message before the sub-chart while the sub-chart is active. The semantics of instantiation say that the second Pers.Class object refers to all other personalization slots. Thus, a personalization scenario on other stations can only start after the assignment of a unique personalization id.

The *crd* variable is used again, after five time ticks, to put the personalized card on the personalization slot.

4.3 Manual operation use case

The manual operation use cases are intended for testing and manual feeding of scenarios with the Play-Engine. Fig. 7 gives an example of such scenario. When the operator presses the "unload" button, the controller is forced to trigger the unload scenario. Most of the manual scenarios have a similar structure. Only the *manual_up* and *manual_down* charts are slightly different because they are used to invoke parametric scenarios. The *manual_up* scenario is given in Fig. 8. The up.Class is instantiated with a specific button by the *Click* message. The *persID* attribute of the specific object is used to form a parameter to the $up(pid)$ message.

4.4 Congruence use case

The system described by system-dynamics use cases (section 4.2) is not a finite state system because identification numbers of personalized cards can grow indefinitely. To allow model checking techniques we introduced a congruence relation with finite index. Our approach is to decrease all positive numbers on the board by one whenever a card is loaded at the output post. If we do this, we get that the transition relation is only among the smallest representatives of the equivalence class defined by the following congruence relation: two states are equivalent if one is obtained from the other by adding some positive constant to all positive cards.

Fig. 9 contains an example of a generic congruence scenario. This scenario is triggered when a card is loaded at the output post. It applies to every personalization slot with a positive *card* attribute. This attribute is copied to a local variable name *crd*. The function *prv* is used to decrease this variable by one and the result is sent back to the *card* attribute. The result can be

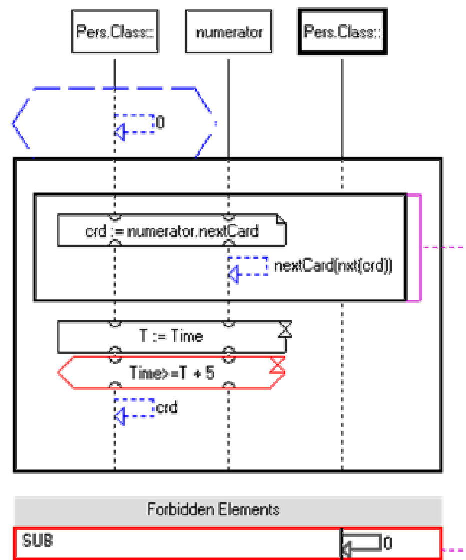

Figure 6: Personalization Scenario

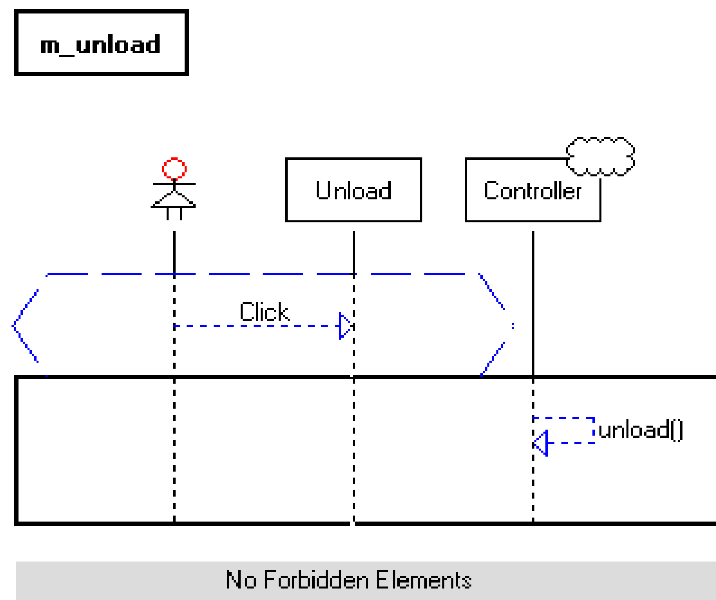

Figure 7: Manual Unload Scenario

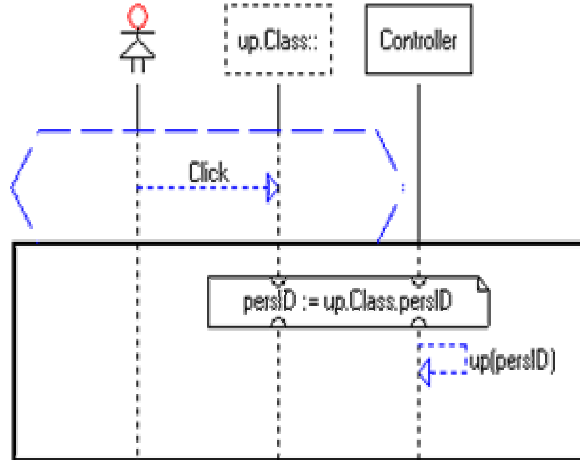

Figure 8: Manual Up Scenario

formulated by $Pers.Card = Pers.Card > 0\,?\,Pers.Card{-}{-} : Pers.Card$. The same pattern is used to decrease also all belt slots and the numerator.

4.5 Error Message use case

In manual operation the operator needs to get feedback for illegal actions. Consider, for example, what happens when the right button is pressed while the rightmost belt slot is not empty. The *manual_right* scenario is triggered and then the controller is forced to trigger the *right* scenario. The *right* scenario will abort immediately without any action. The problem is that the user will not get any feedback. To allow more user-friendly debugging, we added charts to display error messages on such scenarios.

Fig. 10 gives the chart that pops a message on illegal "right" requests. The scenario is triggered by the *Click* message of the "right" button. If the rightmost slot is not empty the *message* attribute of the messageBar object (an additional object used to display messages) is set with an error string. All message conditions are formulated in a similar way.

5 Scheduler synthesis

The model described above can be used for schedulability analysis of the machine. One can 'play' with an interactive simulation of the system and test different schedules.

The term synthesis usually refers to automatic generation of a schedule. This ambition is only achievable for relatively small designs ([5, 11, 8]). In order to cope with industrial sized problems, we choose to aim at a less ambitious goal. Our road toward applicable schedules involves an iterative process in which the designer changes the model until a schedule is achieved. The automatic tools come as design aids when needed. More specifically, an advanced query tool (based on model checking techniques) can be used to guide the search. Usually, This tool cannot solve complete problems due to state space explosion.

To make the above idea more explicit, we demonstrate how the iterative process goes by an example. As an example we describe how the super-single operation mode can be discovered.

Starting from the model detailed in the preceding section, one needs to come up with a good rules for the controller. A natural way to go is to start with greedy rules ([1]). This amounts to adding some scenarios that are triggered whenever a controller command can be issued and executed correctly. For example, one can think of the the GreedyUp scenario. This scenario is triggered whenever a card is positioned beneath an empty personalization slot. The main chart

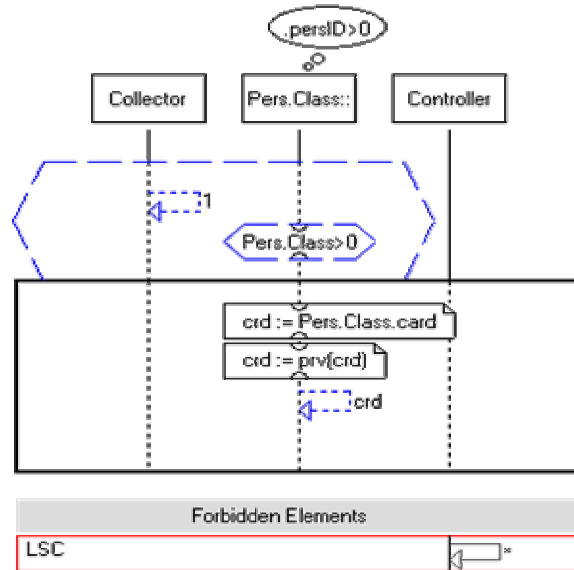

Figure 9: Decrease Personalization on Load Scenario

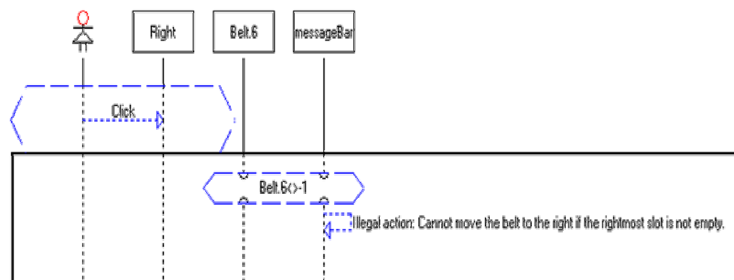

Figure 10: Show Message on Illegal Right Scenario

describes how the card is taken up. It is not hard to imagine how such a chart is played-in via the mockup GUI and then decorated with few annotations.

After playing-in greedy scenarios, one can play-out the model and examine the performance of the resulting controller. As usual, the first attempt will not work. One can resolve problems by adding more rules or changing existing rules. Note that the execution of a set of scenarios is not always deterministic. It can happen that more than one system action is enabled because several charts are traced in parallel. In this case, the play-out mechanism implemented in the play-engine will choose one arbitrarily. Thus, another way to improve the model is to change the precedence relations on system events and on scenarios.

At this point the play-out [6] mechanism comes useful. This tool allows the designer to form advanced queries that can direct the search towards good schedules and refute errors. The query engine is based on a translation of the LSCs model to an equivalent transition relation that can be analyzed using model-checking techniques. Our implementation (described in details in [6]) is based on translation to the SMV modeling language and utilization of the SMV model-checker.

For example, in our search for good schedules we came across a problem that we wanted to resolve. In case more than one card can be taken up (if two personalization slots are empty at the same time) we wanted to specify which goes up first. The way we resolved this question is by issuing a question to the model checker. The model checker ruled that only one option leads to acceptable schedule so we encoded this decision in the charts.

Such questions come frequently along the design phase. Most queries are answered in a matter of seconds by the tool. Sometimes more time is needed to search the states space. If the search takes too long, the designer can try to form simpler questions. In many case one can do with a simple simulation via the naive play-out mechanism that resolves nondeterminism arbitrarily (the next action in taken from the list of active actions according to a fixed precedence relation).

To sum up, we offer a semi-automatic way to extract schedules. The process begins with a coarse strategy which gets more specific as the designer adds charts and details to existing scenarios. The progression is more like a software development process where automatic tools guide the designer towards an implementation of an high level specification.

5.1 Error Handling

Besides the synthesis of the super single mode, described above, we also tested our methodology against another optimization problem. In addition to speed requirements, the size of the machine is also an important factor. Customers need to reserve expensive floor area for the machines, so it is desirable to design machines with minimal belt length.

In this subsection, we describe a method to test if the machine can handle all errors within a certain number of extra slots. If the answer is positive, the analysis produces also a trace that can be used to derive a control program for the machine.

We begin with the problem description. Assume that the machine operates in the super single mode described in the previous section. We know that this is a good schedule if no defected card are present. Since defective cards are rare enough, we can stick to this schedule and ignore the time overhead of handling defective cards. However, we must reserve extra belt slots for error handling. The number of reserved slots must be big enough such that all errors can be accommodated. The discussion is limited to a single error at a time.

The state exploration mechanism delivered by the smart-playout proved ideal for the task. We used the naive playout to run the schedule and introduced an error at some point. From this state, we asked the smart-playout to dump the defective card and return to the next state of the super single mode. The smart playout provided us with a trace that can be used to program a controller for the machine.

We didn't automate this process, but one can imagine an automatic process that spawns verification processes on a distributed network that explore all possible errors and verify that the belt is long enough. This process can collect the traces and produce a control program that

handle all relevant errors - provably.

The general idea that we have used is: once a schedule is obtained, error handling can be done by state space exploration of a path that returns to the cycle. Different errors can be explored in parallel, using grid computing mechanisms.

6 Conclusions

A scenario-based model for a smart-card personalization machine is presented. We believe that this model is easy to understand and maintain. Such models can be used together with simulation and formal verification tools in various design phases. Designers can 'play' with different architectures, apply model-checking techniques to extract schedules and gain intuitive feeling and insight. Because the model is fully graphical, object-oriented and scenario-based it is easy to manipulate it and test variations of the original design. We believe that these features makes our approach suitable for designing production lines. We have also demonstrated a way to extract useful schedules out of the model. Schedulability analysis is carried out semi-manually, namely, the designer is guided by tools that allow her to run queries. The answers to the queries are used to gradually improve the schedule until a complete scheduler is extracted. The resulting scheduler can be run by the play-out which can be considered as an execution machine. One can also translate it to some standard executable language.

References

[1] M. Agopian. A simulation tool for the SuperSingle mode, 2003. Not a paper, a tool.

[2] S. Albert. Cybernetix case-study: Informal description. AMETIST web page http://ametist.cs.utwente.nl/RESEARCH/CYBERNETIX/smartcard.ps, 2002.

[3] D. Amyot and A. Eberlein. An evaluation of scenario notations for telecommunication systems development. In *Int. Conf. on Telecommunication Systems*, 2001.

[4] 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).

[5] B. Gebremichael and F. Vaandrager. Smart card personalisation machine in SMV. AMETIST web page http://ametist.cs.utwente.nl/RESEARCH/BINIAM/cyber.ps, 2002.

[6] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Proc. 4th 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.

[7] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[8] A. Mader. Cybernetix case study: approaching scheduling by decomposition and mixed strategies. AMETIST web page http://ametist.cs.utwente.nl/INTERNAL/PUBLICATIONS/UTPublications.htm, 2003.

[9] 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.

[10] ITU-TS Recommendation Z.120 (11/99): MSC 2000. ITU-TS, Geneva, 1999.

[11] T. C. Ruys. Optimal scheduling using branch and bound with spin 4.0. In *Proceeding of the 10th SPIN workshop, Portland, Oregon*, 2002.

[12] UML. Documentation of the unified modeling language (UML). Available from the Object Management Group (OMG), http://www.omg.org.