Monitoring and Control in Scenario-Based Requirements Analysis

Emmanuel Letier
Dept. d'Ingénierie Informatique,
Université catholique de Louvain
Place Sainte Barbe, 2, B-1348,
Louvain-la-Neuve, Belgium
eletier@info.ucl.ac.be

Jeff Kramer, Jeff Magee, Sebastian Uchitel
Department of Computing, Imperial College London
and London Software Systems
180 Queen's Gate, SW7 2BZ,
London, UK
{jk, jnm, su2}@doc.ic.ac.uk

ABSTRACT

Scenarios are an effective means for eliciting, validating and documenting requirements. At the requirements level, scenarios describe sequences of interactions between the software-to-be and agents in the environment. Interactions correspond to the occurrence of an event that is controlled by one agent and monitored by another.

This paper presents a technique to analyse requirements-level scenarios for unforeseen, potentially harmful, consequences. Our aim is to perform analysis early in system development, where it is highly cost-effective. The approach recognises the importance of monitoring and control issues and extends existing work on implied scenarios accordingly. These so-called input-output implied scenarios expose problematic behaviours in scenario descriptions that cannot be detected using standard implied scenarios. Validation of these implied scenarios supports requirements elaboration. We demonstrate the relevance of input-output implied scenarios using a number of examples.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Verification, Design, Languages, Documentation.

Keywords

Implied scenarios, message sequence charts, scenario-based requirements elaboration.

1. INTRODUCTION

Scenario-based specification languages such as Message Sequence Charts (MSCs) are popular among software engineers for eliciting, documenting and validating software requirements. Scenarios describe typical examples of system executions. They can be used to infer concurrent state machine models of a system [15, 22] and declarative specifications of system goals [12]. They can also be used to identify possible exceptions that may occur at each step of the scenarios and requirements to deal with such exceptions [19]. Because scenarios are used at a very early stage of system development, it is certainly worthwhile to perform analysis before inferring other models. This has motivated the development of a variety of analysis techniques including detecting race conditions and timing conflicts [1], non-local choices [4], implied scenarios [3, 21], and performing pattern matching [16], and model-checking [2].

A *scenario* corresponds to a single temporal sequence of interactions between components of a system. The nature of the interactions and components involved in a scenario depends on the kind of system being modelled and the chosen level of abstraction. When modelling concurrent software architectures, scenarios describe sequences of message exchanges between concurrent software processes; when modelling an object-oriented software system, scenarios describe sequences of method calls between implementation-level objects.

At the requirements level, scenarios describe sequences of interactions between *agents* of a system composed of software agents, human agents, and hardware devices such as sensors and actuators [7, 13]. Each interaction in such scenario corresponds to the occurrence of an event that is synchronously *controlled* by an agent and *monitored* by another agent [18, 11, 14].

Our objective is to provide a technique for the early analysis of such requirements-level scenarios. The technique consists in detecting a novel kind of implied scenario, called an input-output implied scenario, that arises because of the monitoring and control characteristic of agents. Input-output implied scenarios are unspecified scenarios that are exhibited in *every* concurrent state machine model structurally and behaviourally consistent with the specified scenarios. Exposing input-output implied scenarios to stakeholders provides early feedback about inevitable and possibly unforeseen consequences of their specified scenarios.

The notion of an implied scenario was first introduced in [3]. Its definition is intimately connected to the underlying model of concurrent state machines that produces the system behaviours. Algorithms and complexity results for detecting implied scenarios have been studied for various models of concurrency including synchronous and asynchronous communication models [3]. Tool support for detecting implied scenarios in the synchronous

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-58113-963-2/05/0005…$5.00.

communication model for elaborating scenarios and behaviour models is described in [21].

However, the standard notion of implied scenarios is not suitable for analysing requirements-level scenarios because it is based on models of concurrent state machines that do not distinguish between the monitored and controlled events of agents. In such models, an agent can prevent the occurrences of events it monitors. As a result, checking for standard implied scenarios may fail to detect some critical problems in requirements-level scenarios.

The contributions of this paper are as follows. *Firstly*, we define a novel kind of implied scenario called input-output implied scenario that ensures that an agent cannot inhibit the occurrence of events it monitors. *Secondly*, we show how to detect input-output implied scenarios and provide tool support to do so. *Thirdly,* we demonstrate the relevance of flaws that input-output implied scenarios can reveal through a number of examples taken from the literature. *More generally,* this work demonstrates that monitoring and control are important issues when analysing requirement-level scenarios.

Note that there is no need to extend scenario-based specification languages to be more expressive; an input-output scenario is the result of analysing a scenario-based model for unspecified behaviours, much like a counter-example generated by a model checker is the result of verifying whether a state-machine model satisfies a system property.

The paper is organized as follows. Section 2 reviews background work on MSC and implied scenarios. Section 3 presents input-output implied scenarios and how to detect them. Section 4 presents the analysis of several published scenario-based models. Lessons learned from these case studies are discussed in Section 5.

2. BACKGROUND

2.1 Specifying Scenarios with MSC

In the MSC framework, a scenario corresponds to a single temporal sequence of interactions between components of a system. The MSC notation provides a convenient visual notation to express multiple scenarios by allowing specifiers to structure scenarios by episodes (called basic MSCs) and to combine episodes into a directed graph (called a high-level MSC) that defines the possible continuations and loops between episodes.

A *basic Message Sequence Chart* (bMSC) is composed of vertical lines representing components time lines and horizontal arrows representing interactions between components. Each bMSC denotes a partial ordering of components interactions. For requirements-level scenarios, each interaction in a bMSC corresponds to the occurrence of an event that is synchronously controlled by the source agent and monitored by the target agent.

A *high-level Message Sequence Chart* (hMSC) is a directed graph with nodes as bMSCs and edges denoting possible continuations between bMSCs. The semantics of a MSC model sc is a set of sequence of events, noted Bh(sc), that follows some maximal path in the hMSC.

Figure 1 shows an MSC model of a boiler control system [21]. It defines scenarios showing how a *Control* unit operates *Sensor* and *Actuator* components to control the pressure of a steam boiler. A *Database* is used as a repository to buffer pressure data while the *Control* unit performs calculations and commands the *Actuator*.

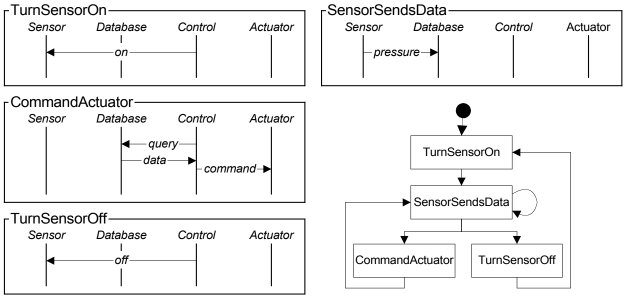

Figure 1 – The Boiler Control system.

2.2 Modelling System Behaviour with LTS

The formalism we use in this paper for modelling and reasoning about concurrent systems is that of Labelled Transitions Systems (LTS) [Mag99]. It allows a system to be described as a set of concurrent state machines where each component is characterized by a set of states and by the possible transitions between these states where each transition is labelled by an event. Figure 3 shows an example LTS for the Control component of the boiler control system.

Complex system behaviour can be modelled by parallel composition of the component LTS models. Parallel composition models components that execute asynchronously but synchronize on shared events. In the sequel, the set of sequence of events that can be generated by an LTS model M will be noted Bh(M).

2.3 Implied Scenarios

A concurrent state machine model inferred from a set of scenarios should be composed of LTSs modelling each of the components appearing in the scenarios. In addition, each component LTS should exhibit as sequences of events at least all scenarios projected to the time line of that component. This consistency constraint between an MSC model and an inferred LTS model is defined formally as follows.

Definition (MSC-LTS consistency). Let Sc be a scenario-based model with components 1, ..., n. A concurrent state machine model M = (M1 || ... || Mn) is *consistent* with Sc if, and only if, for each component i, $Bh(Sc)_{|events(i)} \subseteq Bh(Mi)$ where events(i) is the set of events involving components i in the scenarios.

Since scenario-based models describe only examples of system behaviours, it is natural that a composite LTS model consistent with those scenarios exhibits more behaviours than those explicitly captured in the scenarios. However, some of these additional behaviours may be present in *every* concurrent state machine model that is consistent with the specified scenarios. Such scenarios are called *implied scenarios*.

Definition (implied scenarios). A trace tr is an implied scenario of a MSC model sc if, and only if, (1) $tr \notin Bh(sc)$ and (2) $tr \in Bh(M)$ *for all* concurrent state machine model M that are *consistent* with sc.

The occurrence of implied scenarios is due to the fact that the scenario-based model describes allowed system behaviours from a global, system-wide perspective, whereas in the concurrent state machine model each agent acts locally based on local information.

For example, Figure 2 shows an implied scenario for the MSC description of Figure 1.

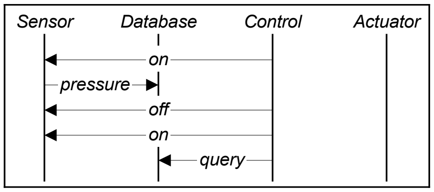

Figure 2 – Implied scenario of MSC in Figure 1

This scenario does not correspond to a behaviour that is specified in the MSC model of Figure 4 because after the second on event that initializes the *Sensor*, there must be a pressure event in which the sensor inputs data in the database before any query event can occur.

This scenario will however be present in every concurrent state machine model consistent with the MSC model, because as far as each component is concerned, the scenario proceeds as specified in the MSC model. For example the *Control* component that generates the unexpected query event believes that the scenario follows the acceptable path *StartSensor, SensorSendsData, StopSensor, StartSensor, SensorSendsData, CommandActuator*. The *Database* component on the other hand believes the scenario follows another acceptable path *StartSensor, SensorSendsData, CommandActuator*. In this implied scenario, when the query event occurs, the *Control* component is not aware that the *Sensor* has not yet sent data in the *Database* after it was last turned on, and the *Database* component is not aware that the *Sensor* has been turned off and on again.

2.4 Detecting Implied Scenarios

Checking a MSC model for implied scenario does not require the construction of every concurrent state machine model that is consistent with the scenarios. If the MSC model describes a regular language the presence of implied scenario can be verified by constructing a finite concurrent state machine model M_{min} that is *minimal* in the sense that every other concurrent state machine model M' consistent with the scenarios has at least as much behaviour as M_{min}, i.e. $Bh(M_{min}) \subseteq Bh(M')$.

The technique for detecting implied scenarios therefore consists in:

- constructing for each agent i a deterministic LTS model $M_{i\ min}$ that covers exactly the given scenario traces projected on the agent time line, i.e. $Bh(M_{i\ min}) = Bh(Sc)_{|events(i)}$;
- defining the minimal concurrent state machine model M_{min} as the composition of each of these agent model, i.e. $M_{min} = (M_{1\ min} \,||\, \ldots \,||\, M_{n\ min})$;
- constructing a monolithic LTS model T that captures exactly the set of global behaviours described by the scenarios, i.e. $Bh(T) = Bh(Sc)$; and
- using the model checking feature of the LTSA toolset to check whether $Bh(M_{min}) \subseteq Bh(T)$.

If the inclusion does not hold, the model checker will generate a trace tr of M_{min} up to the point where it first deviates from T. By construction, M_{min} is consistent with the scenario-based model and it can be shown that this behaviour model is *minimal* [21]. It results that every counter-example tr generated by model checking the above inclusion is an implied scenario.

As an example, Fig. 3 shows the LTS model for the *Control* components that is synthesized from the MSC model of Figure 3.

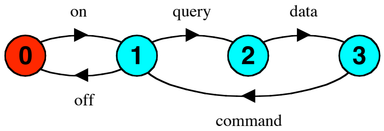

Figure 3 – LTS of Control component

The minimal concurrent state machine model for the scenarios in Figure 1 is obtained as the composition of the minimal LTS model for the *Sensor*, *Database*, *Control* and *Actuator* components. Model checking trace inclusion against the LTS model T for the Boiler control system generates the implied scenario of Figure 2.

2.5 Model Elaboration with Implied Scenarios

An implied scenario may or may not correspond to acceptable system behaviour. In [21], we proposed a model elaboration process in which stakeholders use the feedback provided by implied scenarios to elaborate the scenario-based model with additional positive and negative scenarios.

When an implied scenario is detected, stakeholders are requested to decide whether the sequence of events described by the implied scenario is allowed to happen or not. If they decide it is allowed to happen, they have to specify how the scenario should continue and add this scenario as a positive scenario in the hMSC graph. If they decide the sequence of events is not allowed to happen (or is impossible to occur in the application domain), they can specify the unwanted behaviour as a negative scenario.

For example, if the occurrence of the query event in Figure 2 is considered to be acceptable system behaviour, a new positive scenario that describes how the database should react to this query will be added to the set of positive scenarios. If on the other hand, the occurrence of the query event in Fig 4 represents an unacceptable behaviour, the scenario will be declared as a negative scenario.

The technique for detecting implied scenarios has been adapted so as to be able to detect implied scenarios in the presence of negative scenarios. The process of identifying implied scenarios and enriching the scenario-based model with positive and negative scenarios is iterative and may continue until no more implied scenarios are detected.

Note that documenting unacceptable system behaviours as negative scenarios does not actually solve the problem exposed by implied scenarios; any concurrent state machine consistent with the initial set of positive scenarios will still produce the rejected implied scenarios. The only way to remove the unwanted behaviours will be to eventually modify the behaviours described in the initial set of positive scenarios.

In the steam boiler example, one way to avoid the implied scenario of Fig 2 consists in changing the behaviours described in the MSC model of Fig 1 so that the *Database* component is informed by the *Control* component every time the *Sensor* is turned on and off.

Avoiding unacceptable implied scenarios requires making design decisions in which the system behaviours described in the initial scenarios are modified, possibly by removing some scenarios, by changing the sequential order of some events, by introducing new events or even by introducing new agents into the system. Such design decisions however require that the implied scenarios have first been validated as positive or negative scenarios.

3. INPUT-OUTPUT IMPLIED SCENARIOS

3.1 Motivation

The ability to declare which events are monitored and controlled by each agent is an essential feature of requirements modelling languages [18, 11, 14]. The events controlled by an agent are those that can be performed or initiated by that agent; the events monitored by an agent are those whose occurrences are observed by the agent. In MSC models, the monitoring and control capabilities of agents are naturally given by the directions of the arrows; an event is controlled by the source agent and monitored by the target agent.

The monitoring and control capabilities of an agent constrain what behaviours can be required of that agent. The description of the behaviours of an agent must satisfy the following three conditions [23]: (i) it must be defined entirely in terms of the agent's monitored and controlled events, (ii) it cannot constrain the occurrences of the agent's monitored events, and (iii) it cannot refer to the future occurrences of monitored events. (See [14] for a formal, characterization of such conditions.)

Any LTS model inferred from scenarios should satisfy these three conditions. The definition of consistency between MSC and LTS models in Section 2.3 takes into consideration condition (i) by requiring the LTS of a component to be defined only in terms of the events in which this component is involved in the scenarios. Condition (iii) is always satisfied in a formalism based on state transitions such as LTS. However, condition (ii) is not taken into consideration. The standard definition of implied scenarios is based on a model of concurrent state machines in which no distinction is made between the events that are monitored and controlled by an agent. The direction of the arrows in the MSC model is simply ignored in the LTS model. This may result in LTS models in which an agent prevents the occurrence of one of its monitored events, thereby violating condition (ii). Important flaws in the scenarios may therefore remain undetected.

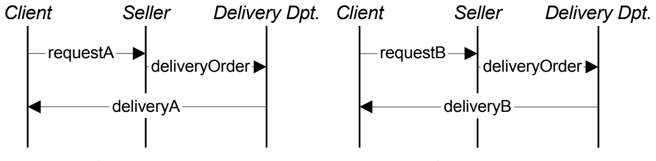

Figure 4 - Scenarios for an order delivery system

Consider for example the two very simple scenarios for a small order delivery system shown in Figure 4. Scenario A describes a client requesting product A to a seller agent, who then sends a delivery order to the delivery department that then delivers the requested product to the client. Scenario B is similar except that product B is requested and delivered.

A problem in these scenarios is that the delivery department receives the same deliveryOrder event regardless of whether the requested product is A or B so that it doesn't know which product needs to be delivered. This model however has no implied scenarios.

Figure 5 shows the LTS of the minimal concurrent state machine model inferred from these scenarios. The LTS for the *DeliveryDprt* shows that after a deliveryOrder, this agent is free to perform either a deliveryA or a deliveryB. However, the *System* LTS model, obtained as the parallel composition of the *Client*, *Seller* and *DelDprt* LTSs, shows that after each requestA or requestB, the correct product will be delivered. The reason is that in the *Client* LTS after a requestA the LTS only accepts deliveryA and after a requestB it only accepts deliveryB. In the LTS model, it is therefore the client who ensures that the correct product is delivered by controlling the occurrences of deliveryA and deliveryB events. The synthesized LTS model is in contradiction with the scenario-based model in which the occurrences of these events are controlled by the delivery department, not the *Client*.

A fundamental rule -corresponding to condition (ii) above- that should be satisfied when synthesizing concurrent state machine models from scenarios is that *an agent should not constrain the occurrences of its monitored events*.

In the context of LTS models, what is meant for an agent to constrain an event is defined as follows.

Definition (control of events in LTS models). An event is said to be constrained in a LTS if and only if there is some state in the LTS from which there is no outgoing transition labeled with that event.

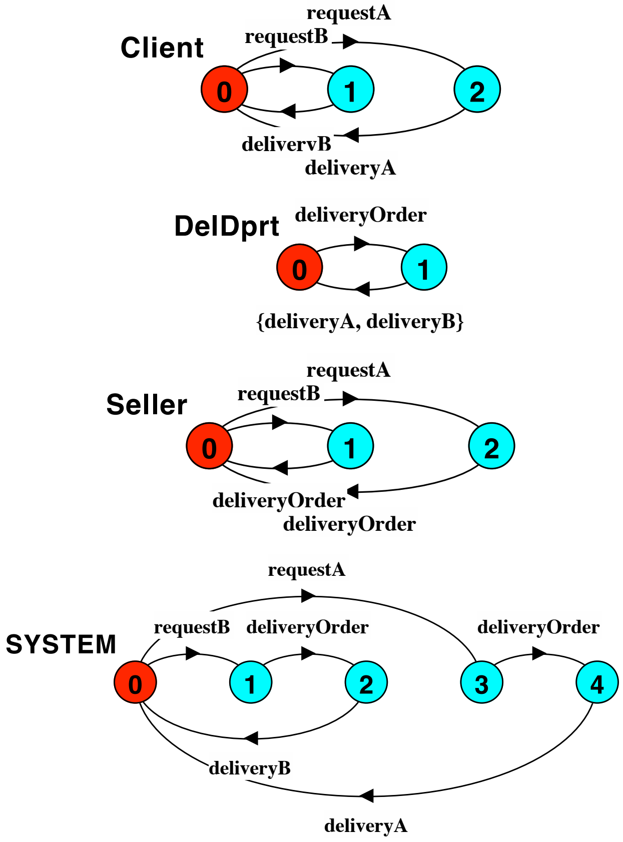

Figure 5 - LTS models for the Delivery system

For example, the *Client* LTS model in Figure 5 constrains the occurrences of all its events, including its monitored events deliveryA and deliveryB. The modified *Client* LTS model in Figure 7 constrains only the occurrences of its controlled events requestA, requestB.

The basic idea of input-output implied scenario is to consider the presence of implied scenario by taking into consideration the additional constraint that the LTS model of each component can only constrain its controlled events and can therefore not constrain its monitored events.

3.2 Definition

The concept of input-output implied scenario is therefore based on the following stronger notion of consistency between scenarios and concurrent state machines.

Definition (IO consistency between MSC and LTS models). Let sc be a scenario-based model with components 1, …, n. A concurrent state machine model $M = (M1 \| \ldots \| Mn)$ is *IO-consistent* with sc if, and only if, for each component i, $Bh(sc)|_{events(i)} \subseteq Bh(Mi)$ and M_i does not constrain the occurrences of events in Monitored_events(i) where Monitored_events (i) is the set of events received by i in the MSC model.

This definition is similar to the definition of consistency introduced in Section 2.3 except that it adds the constraint that the state machine model of a component cannot control its monitored events. Every system behaviour model that is IO-consistent with a scenario-based model is consistent with this scenario-based model, but not reciprocally.

Input-output implied scenarios are then defined as follows.

Definition (Input-Output Implied Scenarios). A trace tr is an *input-output implied scenario* of a scenario-based model sc if, and only if, (1) $tr \notin Bh(sc)$ and (2) $tr \in Bh(M)$ for all concurrent state machine model M that is *IO-consistent* with sc.

This definition is similar to the definition of standard implied scenario except that the concept of consistency between scenarios and LTS models has been replaced by the concept of IO-consistency. Since IO-consistency is stronger the consistency, every standard implied scenario is also an IO-implied scenario. Conversely, there might be IO-implied scenarios that are not standard implied scenarios.

For example, the scenario in Figure 6 is an input-output implied scenario of the two scenarios in Figure 4, but it not an implied scenario.

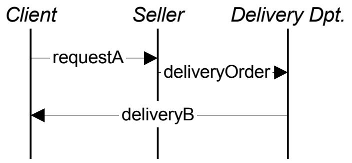

Figure 6 – IO-implied scenario for MSC of Figure 4

This input-output implied scenario shows that a consequence of the two scenario of Figure 4 is that it is possible that a request for product A is followed by a delivery of product B.

This input-output implied scenario represents undesirable system behaviour. It can be avoided by changing the scenarios in Fig. 4 so that the deliveryOrder event in scenarios A and B is changed to deliveryOrderA and deliveryOrderB, respectively. Other examples of input-output implied scenarios in real scenario-based models will be seen in Section 4.

Note that the notion of input-output implied scenarios is not equivalent to that of implied scenarios in an asynchronous setting. Asynchrony would introduce implied scenarios that are not input-output implied scenarios. In fact, synchrony/asynchrony and monitoring/control are orthogonal concepts: An agent could be modelled to observe an event at a later logical time than when the event was performed by the controlling agent. Such an assumption would require modelling the interaction as asynchronous and guaranteeing that the monitoring agent never constrains the occurrence of the event.

3.3 Detecting IO Implied Scenarios

The technique for detecting input-output implied scenarios is similar to the technique for detecting standard implied scenarios described previously. It is based on constructing a minimal concurrent state machine model $M_{IO\text{-}Min}$ given as the parallel composition of the minimal state machine models of all components.

In this case, however, the components LTS may not constrain the occurrences of their monitored events. For detecting input-output implied scenarios, the minimal LTS model of each component is constructed by augmenting the minimal LTS model as built for detecting standard implied scenario with (i) an additional "sink" state with no outgoing transition but with self-transitions for every events monitored by the agent, and (ii) for each monitored event and every state in which there is no transition labeled with that event, an additional transition from this state to the sink state labeled with the monitored event. Figure 7 shows the minimal LTS model thereby generated for the *Client* from the scenarios in Figure 4. This LTS model augments the *Client* LTS model in Figure 5 with an additional sink state (state 1) and additional transitions to the sink state so that the monitored events deliveryA and deliveryB are not constrained by the LTS.

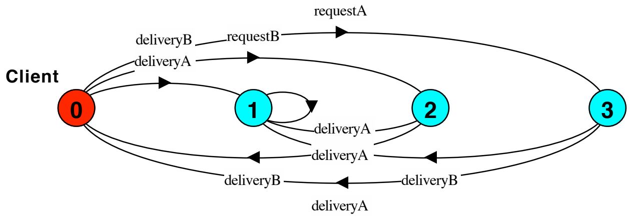

Figure 7 - Input-output LTS for Client

The parallel composition of component LTS as described above can be shown to be minimal with respect to trace inclusion compared with any other concurrent state machine that is IO-consistent with the MSC model.

The presence of implied scenarios is then analysed by model checking whether $Bh(M_{IO\text{-}Min}) \subseteq Bh(T)$ where T is the LTS model that captures the set of behaviours described by the scenario-based model. This LTS is the same as the one used for detecting standard implied scenarios. Space complexity of the analysis is exponential with respect to the number of components (See [21] for a full discussion).

4. EXAMPLES

The purpose of this section is to show through examples of scenario-based models taken from the literature that input-output implied scenarios appear in real scenario-based models and that their detection provides relevant feedback for further elaboration of requirements models.

We present the analysis of 4 scenario-based models: two different ATMs [4, 22], a web interface application [6], and a toaster [15]. Other models that have been analyzed but not reported here due to space restrictions are the steam boiler [21], a GSM mobility management protocol [Leue98], scenarios for an automated transport system [21], for a safety injection system of a nuclear power plant [9], and for a web-based payment system. All these models can be downloaded with the tool at http://www.doc.ic.ac.uk/ltsa.

Except for the steam boiler and the transportation systems, none of these models has standard implied scenarios. Input-output implied scenarios on the other hand were detected in all models, except for the web-based payment system. Lessons learned from the case studies are discussed in Section 5.

4.1 A First ATM model

A scenario-based model describing user interaction with an ATM machine is presented in [4]. This model is composed of 13 bMSCs representing small episodes of ATM behaviours (such as *GetPinCode, ProcessPinCode, RefusePinCode, Withdraw*, etc.) and a complex hMSC that aims at providing a complete description of the ATM behaviours. A prefix of a typical scenario that can be obtained from this model is shown in Figure 8.

The model has no standard implied scenarios but does have input-output implies scenarios. Checking the model for input-output implied scenario generates the scenario in Figure 9.

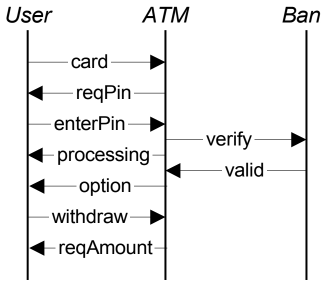

Figure 8 - Prefix of an ATM scenario [4]

This scenario is an input-output implied scenario because as shown in Fig. 8 after the verify event, the MSC model expects a processing event to occur before the valid event. The processing event means that the ATM display a screen indicating that it is processing the user's pin code.

This input-output implied scenario shows that a consequence of the specified MSC model is that the ATM may fail to inform the user that its PIN code is being processed before receiving the validation event from the bank. This input-output implied scenario occurs because the ATM does not control the occurrences of valid events and can therefore not prevent the Bank from performing this event before it has performed the processing event.

Validation with stakeholders is needed to determine whether the input-output implied scenario is an acceptable behavior or not. If the behaviour is not acceptable (for example because it is expected that a user will always see a processing message after entering the PIN) one way to avoid it would be to change the initial MSC model so that the ATM generates the processing event before the verify event instead of after. The input-output implied scenario may also be considered to be impossible in the application domain if it is assumed that the ATM never fails and is always faster than the Bank. (This assumption corresponds to the synchrony hypothesis [5] adopted by many state machine specification languages [17]). In this case, detecting the input-output implied scenario has helped to identify this assumption and to make it explicit for validation.

Alternatively, if the behaviour is acceptable (for example because it is expected that if validation from the Bank is faster than the ATM reaction time users do not need to see a processing message) then a new positive scenario could be added to the MSC model.

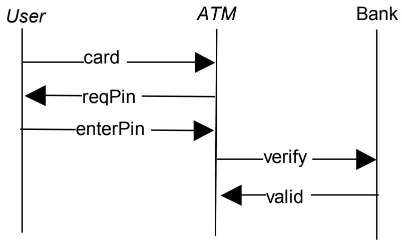

Figure 9 – IO-implied scenario of the ATM [4]

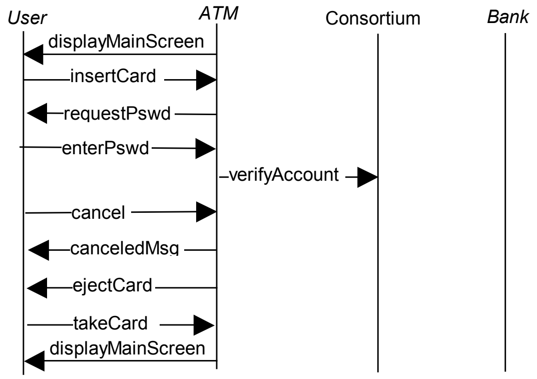

Figure 10 – Scenario for an ATM [22]

4.2 Another ATM model

Another set of scenarios for an ATM machine is presented in [22]. This paper contains only four scenarios for the ATM. Figure 10 shows one of these scenarios, one in which a user inserts its card, enters a password then cancels the transaction.

The model has no standard implied scenarios. However, checking for input-output implied scenario generates the scenario in Figure 11.

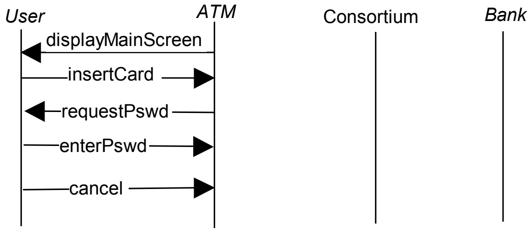

Figure 11 – IO-implied scenario of the ATM [22]

This scenario is not among the specified scenarios because, as shown in Figure10, after the occurrence of an enterPswd event, the model expects the occurrence of a verify event before the cancel event.

The input-output implied scenario shows a situation in which the *User* performs a cancel event before the *ATM* has time to perform the verifyAccount event. In principle, this can occur because the *ATM* cannot constrain the *User* from canceling the operation. If we assume that the ATM agent never fails and is infinitely fast compared to the user (synchrony assumption), the implied scenario cannot happen and will be specified as a negative scenario. Detecting the input-output implied scenario has therefore helped in making this assumption explicit.

However, if we want to take into consideration the possibility of ATM failure to perform the verify event sufficiently fast, we need to view the implied scenario as the prefix of a positive scenario that must be completed by specifying how the system should continue when the cancel events occurs before the verify event is performed by the ATM.

The full model in [22] contains other input-output implied scenarios, all of which are related to the occurrences of cancel events at various stages in the ATM scenarios that were not specified in the initial set of scenarios. These implied scenarios raised questions as to how the ATM should react to cancel events at various points of a transaction. These are crucial aspects to consider in the design of interactions between the user and the ATM. On the downside, handling these implied scenarios by specifying additional positive scenarios quickly became unmanageable, as it requires specifying positive scenarios for all interleaved combinations of cancel events with other events in the scenarios. We comment on this limitation in the next section.

4.3 A Web Interface Application

The next example comes from an industrial case study concerning the development of a web interface application allowing access to an existing Enterprise Resource Planning (ERP) system of a commercial company. Scenarios for this application were written by a research assistant involved in collaboration with the clients of the system [6].

The scenario-based model consists of 7 basic MSCs describing how a user of the system can log in the system and use the interface to access the ERP. Two basic MSCs for successful and failed login issued from this model are shown in Figure 12.

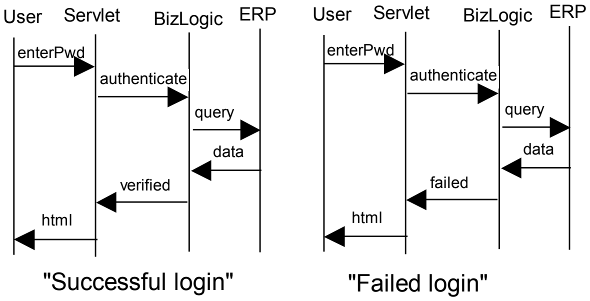

Figure 12 - Scenarios for an ERP system [6]

The hMSC model, not shown here due to space limitation, specifies that after a successful login, the user can use the web interface to perform a search on the ERP database, and after a failed login, the user can only retry to login by entering a new password.

The model contains no standard implied scenarios. However, checking for input-output implied scenarios generates the implied scenario of Figure 13 in which the user can perform a search after a failed log in attempt.

This input-output implied scenario occurs because in the Succesful Login and Failed Login bMSCs, the same html event is used both to signal a failed login and a successful login to the user. In order to avoid this undesirable behaviour, the html event in the Succesfull Login and Failed Login bMSC can be changed into the events displaySucessfullLogInPage and display FailedLogInPage, respectively. The resulting scenario-based model contains no more input-output implied scenarios.

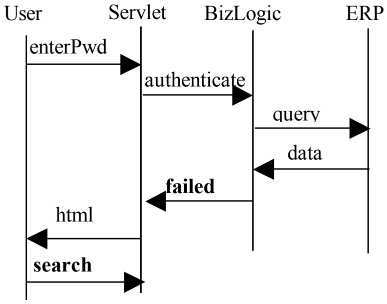

Figure 13 – IO-implied scenario for the ERP sytem

4.4 A Toaster

A scenario-based model of a toaster is presented in [15]. This model is composed of five small scenario episodes connected though a hMSC graph intended to provide a complete description of the toaster behaviours.

Two typical scenarios that can be extracted from the hMSC model are shown in Figure 14. The first scenario shows the normal interactions between a user, a control component and a heating component that result in successful toasting. The second scenario shows what happens when there is no bread in the toaster.

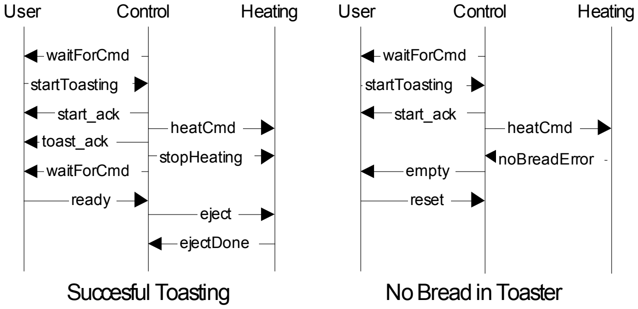

Figure 14 – Scenarios for a Toaster [15]

The model has no standard implied scenarios. However, checking the complete for input-output implied scenarios generates the scenario in Figure 15.

This input-output implied scenario shows the possibility of a noBreadError event occurring after the occurrence of a toast_ack event. This implied scenario occurs because up to the toast_ack event, the Control component follows the "Successful Toasting" scenario whereas the Heating component follows the "No Bread in Toaster" scenario.

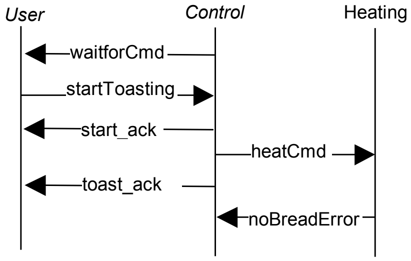

Figure 15 – IO-implied scenario for the Toaster model

This implied scenario may correspond to an undesirable system behaviour; the event toast_ack signals to the user that the toasting process is being carried out normally, yet the heater is signaling that there is no bread in the toaster through the noBreadError event.

Since scenarios show sequences of events only and omit the state information, there might be different ways to interpret this implied scenario. For example, we might, as above, assume that there is no bread in the toaster when the heatCmd event occurs. Alternatively, there might actually be bread in the toaster at the moment of the toast_ack event, and the noBreadError could be a malfunction of the Heating component that incorrectly signals an error, or the noBreadError might be caused by the removal of the bread from the toaster by the user. Each interpretation equally makes sense and may lead to different decisions as to whether the implied scenario is a positive or negative scenario.

If this implied scenario is considered to represent undesirable system behaviour, it could be avoided by extending the interface between the Heater and Control component with an explicit confirmation from the Heater that everything is ready for a successful toasting scenario. This can be done through a noError event in the "Sucessful Toasting" scenario sent by the heater after receiving the heatCmd event, to explicitly signal the Control that there is no error.

5. DISCUSSION

As described, input-output implied scenarios are an effective aid in detecting unforeseen consequences in scenario-based models and in prompting the elaboration of further system requirements. Detection and validation of input-output implied scenarios as undesirable system behaviours led to elaborating the scenario-based models by changing the sequential order of events (in the first ATM model), identifying missing events in the scenarios (in the toaster model), and making some undistinguishable events distinguishable (in the web interface application). Input-output implied scenarios assumed to be impossible in the application domain (as in the first and second ATM examples), allowed us to expose critical domain assumptions that were implicit in the scenarios-based model. Finally, unforeseen positive scenarios prompted the elaboration of system behaviour to cope with situations that had not been considered (in the second ATM example).

The issues exposed by input-output implied scenarios could not have been identified by checking for standard implied scenarios.

Our experience on the case studies has also led us to identify some difficulties with the current elaboration process using input-output implied scenarios.

- The simple process of validating implied scenario by adding positive and negative scenario tends to result in large, unstructured models that become difficult to master. These need some form of refactoring.
- Correcting problems exposed by implied scenarios requires design decisions that may result in different scenarios in which, for example, the order of the events is changed and new events or new components are introduced. There is currently no support for such an elaboration process.
- The classification of an implied scenario as positive or negative often proved to be difficult. This was partly due to lack of domain knowledge. However, there were also other reasons:

  - Implied scenarios show sequences of events only; they don't show what state the system is in when an event occurs. This makes it difficult to interpret the scenarios and there may be more than one interpretation depending on the assumption made about the state of the system (see Toaster example).

  - Stakeholders may have conflicting views about the required system properties, so that an implied scenario might be classified as positive from one point of view and negative from the other. The rationale for rejecting or accepting implied scenario is currently not explicitly recorded. For example in the first ATM model, the implied scenario will be classified as positive or negative depending on whether or not stakeholders consider that a user should always get a 'processing' screen after having entered the pin code.

These difficulties suggest developing a richer elaboration process in which the rationale for accepting or rejecting implied scenarios would be explicitly recorded as a declarative goal specification. These goals would then provide the basis for the application of goal-oriented requirements elaboration techniques such as techniques for refining goals, handling goal conflicts and exploring alternative system designs [13] that ultimately would lead to the specification of a concurrent state machine model with improved scenarios.

Although we recognize the limitations of scenarios [12, 8] when used in isolation, our aim is to exploit a notation widely adopted by practitioners and to support the process of requirements elaboration. Hence, we do not extend scenario-based specification languages to be more expressive nor aim to use a scenario-based formalism as the main model for requirements specification. Scenarios are rather viewed as typical examples of system behaviours from which to eventually infer a concurrent state machine model of the system [15, 22] and declarative specifications of system goals [12]. We believe that checking for input-output implied scenarios is an effective mean to detect and correct gaps and potential errors in requirements-level scenarios *preferably before these scenarios are used to infer state-machine and goal models*.

6. CONCLUSION

Software engineers frequently use scenarios to elicit, document and validate system requirements. Because they are often the very first models produced, early analysis is highly cost-effective.

This paper builds upon previous work on implied scenarios for detecting scenarios that are not part of the scenario-based model but that are present in every concurrent state machine model consistent with the scenarios. We have shown that the standard concept of implied scenario is not suitable for analysing requirements-level scenarios because it is based on a model of concurrent state machines that ignores the distinction between monitored and controlled events. We have proposed the concept of input-output implied scenarios that takes such a distinction into consideration. We have also developed a technique and tool support for detecting input-output implied scenarios.

In practice, input-output implied scenarios detect important problems in scenario-based models that cannot be detected by standard implied scenarios. Several examples of input-output implied scenarios from published scenario-based models were used to illustrate the kind of problems detected and their relevance for requirements elaboration.

The detection of input-output implied scenarios is only one among other techniques to be used for elaborating requirements from scenarios. In particular, the detection of input-output implied scenarios could be used in combinations with techniques for inferring state machine models [Som95, 15, Kru99, 22], declarative specification of system goals [12] and requirements dealing with exceptions [19] from scenarios.

Unlike model-checking, the detection of implied scenarios does not rely on the preliminary identification and formalization of system properties. As mentioned in Section 5, implied scenarios can however be used to identify some of the relevant system properties. Once identified, such properties could be used to model-check future evolution of the scenario-based model (by model checking the concurrent state machine inferred from the scenarios), or more constructively to guide further elaboration of the requirements. We are currently working in that direction by trying to define a formal framework integrating scenario-based and goal-oriented requirements elaboration techniques within the LTSA toolset.

7. REFERENCES

The work reported herein was partially supported by the Belgian “Fond National de la Recherche Scientifique” (FNRS) and EPSRC grant READS GR/S03270

8. REFERENCES

[1] R. Alur, G.J. Holzmann, and D. Peled, "An analyzer for message sequence charts", *Software Concepts and Tools*, vol. 17, no. 2, pp. 70(77), 1996.

[2] R. Alur and M. Yannakakis, “Model checking of message sequence charts", in 10th International Conference on Concurrency Theory, 1999, LNCS 1664, pp. 114-129.

[3] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts, in 22nd International Conference on Software Engineering, 2000.

[4] H. Ben-Abdallah, S. Leue: MESA: Support for Scenario-Based Design of Concurrent Systems , in 4th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 1998.

[5] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: Design, semantics, implementation. *Sci. of Comp. Prog.*, 19(2):87–152, 1992.

[6] R. Chatley, J. Kramer, J. Magee and S. Uchitel, Model-based Simulation of Web Applications for Usability Assessment, in *ICSE Workshop on Bridging the Gaps between Software Engineering and Human-Computer Interaction*, 2003.

[7] M. Feather, “Language Support for the Specification and Development of Composite Systems”, *ACM Trans. on Programming Languages and Systems* 9(2), Apr. 87, 198-234.

[8] Harel, D. Marelly, R. *Come, Let’s Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer, New York. 2003.

[9] C. Heitmeyer, R.D. Jeffords and B. G. Labaw, “Automated Consistency Checking of Requirements Specifications”, *ACM Trans. on Software Eng. and Methodology* 5, 3, July 1996, 231-261.

[10] International Telecommunications Union Telecommunication Standardisation Sector, Message Sequence Charts, 1996.

[11] M. Jackson, Software Requirements & Specifications - *A Lexicon of Practice, Principles and Prejudices*. ACM Press, Addison-Wesley, 1995.

[12] A. van Lamsweerde and L. Willement, Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering*, *24* (12). 1089-1114, 1998.

[13] A. van Lamsweerde, Requirements Engineering in the Year 00: A Research Perspective, *22nd International Conference on Software Engineering*, Limerick, 2000.

[14] E. Letier and A. van Lamsweerde, Agent-Based Tactics for Goal-Oriented Requirements Elaboration, in *24th Intl. Conference on Software Engineering*, 2002.

[15] S. Leue, L. Mehrmann, M. Reza: Synthesizing ROOM Models from Message Sequence Chart Specifications, 13th

IEEE Conference on Automated Software Engineering, 1998.

[16] A. Muscholl, D. Peled, and Z. Su, Deciding properties of message sequence charts," in *Foundations of Software Sci. and Comp. Structures*, 1998.

[17] J. Niu, J. Atlee, N. Day, Template Semantics for Model-Based Notations, *IEEE Transactions on Software Engineering,* 29(10) 866-882, 2003.

[18] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems", *Science of Computer Programming*, Vol. 25, 1995, 41-61.

[19] Sutcliffe A.G., Maiden N.A.M., Minocha S. & Manuel D. (1998), Supporting Scenario Based Requirements engineering, *IEEE Transactions on Software Engineering*, 24(12), pp 1072-1088.

[20] Object Management Group, *Unified Modelling Language*, 2002.

[21] S. Uchitel, J. Kramer, and J. Magee, Incremental elaboration of scenario-based specifications and behavior models using implied scenarios, *ACM Transactions on Software Engineering and Methodology*, pp 37-85, 13(1), 2004

[22] Whittle, J. and Schumann, J. "Generating Statechart Designs from Scenarios" in *22nd International Conference on Software Engineering*, 2000.

[23] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering", *ACM Transactions on Software Engineering and Methodology*, 1997, 1-30.