Negative Scenarios for Implied Scenario Elicitation

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

ABSTRACT

Scenario-based specifications such as Message Sequence Charts (MSCs) are popular for requirement elicitation and specification. MSCs describe two distinct aspects of a system: on the one hand they provide examples of intended system behaviour and on the other they outline the system architecture. A mismatch between architecture and behaviour may give rise to implied scenarios. Implied scenarios occur because a component's local view of the system state is insufficient to enforce specified system behaviour. An implied scenario indicates a gap in the MSC specification that needs to be clarified. It may simply mean that an acceptable scenario has been overlooked and should be added to the scenario specification. Alternatively, it may represent an unacceptable behaviour which should be documented and avoided in the final implementation. Thus implied scenarios can be used to iteratively drive requirements elicitation. However, in order to do so, tools for coping with rejected implied scenarios are needed. The contributions of this paper are twofold. Firstly, we define a language for describing negative scenarios. Secondly, we complement existing implied scenario detection methods with techniques for accommodating negative scenarios.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Requirements/Specifications – *Languages, Tools*

General Terms

Algorithms, Languages, Verification.

Keywords

MSC, negative scenarios, implied scenarios.

1. INTRODUCTION

Scenario-based specifications such as Message Sequence Charts (MSCs) [7] describe how system components, the environment and users interact in order to provide system level functionality. Their simplicity and intuitive graphical representation facilitate stakeholder involvement and make them popular for requirement specification.

SIGSOFT 2002/FSE-10, Nov. 18-22, 2002, Charleston, SC, USA.

MSCs describe two distinct aspects of a system. They depict a set of acceptable system traces and outline the system architecture by defining component structure and component interfaces. A mismatch between specified behaviour and architecture may give rise to implied scenarios [1, 16]. Implied scenarios are the result of specifying the global behaviour of a system that will be implemented component-wise. In other words, the interactions do not provide components with enough local information to enforce the specified system behaviour.

The existence of implied scenarios is an indication of unexpected system behaviour. An implied scenario may simply mean that an acceptable scenario has been overlooked and that the scenario specification needs to be completed. Alternatively, the implied scenario may represent an unacceptable behaviour and therefore require careful documentation of the undesired situation so it can be avoided in the final implementation. Ultimately, the decision of accepting or rejecting depends exclusively on the problem domain, and should therefore be validated with stakeholders.

In previous work [16], we have presented techniques for synthesising architecture models from MSC specifications. Architecture models are labelled transition systems (LTSs) that preserve the component structure and component interfaces defined in the MSC specification and that can exhibit all the traces described in the specification. We have shown that these synthesised models are minimal with respect to trace inclusion and may exhibit additional unspecified behaviours, i.e. implied scenarios. In addition, we have developed methods for detecting implied scenarios by model checking minimal architecture models synthesised from a MSC specification [16, 17][1].

In this paper we address the issue of providing stakeholders with suitable tools for elaborating their scenario specifications based on the feedback provided by implied scenarios. If detection and validation of implied scenarios are to be used iteratively to drive the completion of scenario-based specifications, stakeholders must be provided with a notation for documenting rejected implied scenarios as negative scenarios and with automated techniques for detecting further implied scenarios in the presence of elicited negative scenarios.

The contributions of this paper are twofold. Firstly, we define a notation and semantics for describing negative scenarios in MSC specifications. Secondly, we complement existing automated procedures for implied scenario detection [16, 17] with techniques for eliciting further implied scenarios in the presence of negative scenarios. The latter is achieved by constructing behavioural constraints (in the form of LTSs) from negative scenarios. These

[1] Note that in previous work we refer to these models as implementation models.

constraints, when composed with a minimal architecture model, prevent the latter from performing negative scenarios. Constrained architecture models can be model checked for implied scenarios using the detection method presented in [17].

In Section 2, we introduce MSC specifications together with architecture models and implied scenarios. Section 3 motivates and describes a simple negative scenario notation and the procedure for building behavioural constraints from them in order to detect implied scenarios. Section 4, motivates a more expressive negative scenario notation that is needed in order to make the iterative process of detecting and rejecting implied scenarios converge. Section 5 describes to the tool we have built. Finally, we discuss related work and conclusions.

2. BACKGROUND

In this section, we succinctly describe message sequence charts (MSCs), architecture models and implied scenarios. For a more detailed explanation, refer to [16]. We also introduce the Boiler Control system that is used to exemplify the different concepts presented in the paper. The Boiler Control system, shown in Figure 1, has several 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*.

2.1 Message Sequence Charts

The message sequence charts we use are based on those of the MSC ITU language [7]. For a detailed and formal description of the language we use refer to [16]. An MSC specification consists of several basic MSCs and one high-level MSC. A basic MSC (bMSC) describes a finite interaction between a set of components (see top of Figure 1). Each vertical line, called instance, represents a component. We use the terms component and instance interchangeably. Each horizontal arrow represents a synchronous message, its source on one instance corresponds to a message output and its target on a different instance corresponds to a message input. For simplicity, we shall require message labels to denote message types. In other words, a message uniquely characterizes a sending and a receiving component. In addition, as messages are considered to be synchronous we require arrows to be drawn horizontally and do not allow components to send messages to themselves.

A bMSC determines a partial ordering of messages. Thus, the behaviour of a bMSC is the set of sequences of message labels that are total orderings (or linearisations) of its partial order. For example, the behaviour of the bMSC *Analysis* of Figure 1 comprises only one sequence of labels: *query, data, command*.

Definition 1. (Traces specified by a bMSC) Given bMSC b, the language of b is a set $L(b)$ of traces (or words) over the alphabet of message labels of b (denoted $\alpha(b)$), where $L(b) = \{w \in \alpha(b)^* \mid w$ is a linearisation of the partial order determined by $b\}$.

A high-level MSC (hMSC) allows composing bMSCs. It is a directed graph where nodes represent bMSCs and edges indicate their possible continuations (see bottom of Figure 1). hMSCs also have an initial node represented with a black blob. The hMSC shows how the system can evolve from one bMSC to another.

Note that (in-line with standard interpretation of hMSCs e.g. [7]) components do not wait until all message interactions of a bMSC have occurred before moving on to the next bMSC, i.e. there is no implicit synchronisation that components use in order to know when a scenario is completed. Components move into subsequent scenarios in an unsynchronised fashion. Consequently, an edge in the hMSC can be regarded as concatenating syntactically (or composing sequentially) the two bMSCs it links. For example, the edge (*Analysis, Register*) determines a bMSC with two possible sequences of events *query, data, pressure, command* and *query, data, command, pressure*. Messages *command* and *pressure* must occur after *data* (because *command*/*pressure* occurs further down on the *Control*/*Database* instance) but there is no temporal relation between them.

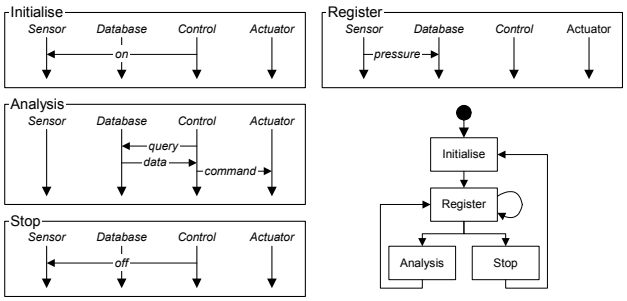

Figure 1 – MSC specification of Boiler Control system.

The behaviour of an MSC specification is given by a set of sequences of message labels: those determined by composing sequentially the bMSCs of any maximal path in the hMSC, where maximal path is a path that cannot be further extended. Note that this corresponds to the adoption of weak sequential composition, which is the standard interpretation of hMSCs [7].

Definition 2. (Traces specified by a MSC Specification) Given a MSC specification *Spec* the language of *Spec* is a set $L(Spec)$ of traces over the alphabet of message labels of *Spec* denoted $\alpha(Spec)$, where $L(Spec) = \{w \in L(b) \mid b$ is the sequential composition of a maximal path in *Spec*$\}$. We shall refer to the set of messages sent or received by an instance i, denoted $\alpha(i)$, as the instance's alphabet

2.2 Architecture Models and Implied Scenarios

A MSC specification not only determines a set of acceptable system executions, but also states what components participate in these executions and what responsibilities they have. Thus, building a set of components that can send and receive messages as in the MSC specification is a relevant issue. We model components as labelled transition systems (LTS) where labels represent messages that the components can input and output. We consider the system as the parallel composition of all components. In other words, the system is the result of composing components such that they execute asynchronously but synchronize on all shared message labels. For a detailed explanation of LTS and parallel composition, refer to [10].

Definition 3. (Labelled Transition Systems) A finite labelled transition system (LTS) P is a structure (S, L, Δ, q) where S is a set of states, $L = \alpha(P) \cup \{\tau\}$ is a set of labels where $\alpha(P)$ denotes the alphabet of P and τ denotes internal actions that cannot be observed by the environment of a LTS, $\Delta \subseteq (S \times L \times S)$, and $q \in S$ is the initial state.

Definition 4. (Executions, Traces, and Maximal Traces) Let $P=(S, L, \Delta, q)$ be a LTS. A sequence $e=q_1,l_1,q_2,l_2\ldots$ is an execution of P if $q_1 = q$ and $(q_i , l_i, q_{i+1})\in\Delta$ for $0<i<|e|$. An execution is maximal if it is not a proper prefix of any other execution. A word w over the alphabet $\alpha(P)$ is a (maximal) trace of P if there is a (maximal) execution of P such that $w = e|_{\alpha(P)}$. We also define $tr(P)$ and $mt(P)$ as the sets of traces and maximal traces of P.

The weakest condition that one can require from an architecture model of a MSC specification is that it must comprise a LTS for each instance, that LTS alphabets must coincide with those of the instances they model, and that the composed LTS must be able to execute all the traces specified in the MSC.

Definition 5. (Architecture models) Let *Spec* be a MSC specification with instances C, and P a LTS resulting from the parallel composition of LTSs P_c with $c \in C$. We define P to be a architecture model of *Spec* if $\alpha(P_c) = \alpha(c)$ and $L(Spec) \subseteq mt(P)$.

However, this is a rather weak notion of architecture model. In many cases one wishes to obtain an architecture model that has exactly the same behaviour as the specification. Unfortunately, such a model does not always exist [16]. For example, Figure 2 depicts the prefix of a trace that appears in all architecture models of the MSC in Figure 1. Nevertheless, there is no specified behaviour of Figure 1 that could start with such a prefix. Figure 2 shows how the *Control* component is accessing the *Database* and receiving information from a previous activation of the *Sensor*. This is not a specified behaviour because the MSC specification states that after initialising *Sensor* some data must be registered into the *Database* before any queries can be done. The problem is that the *Control* component cannot see when the *Sensor* has registered data in the *Database*. Thus if it is to *query* the *Database* after data has been registered at least once, it must rely on the *Database* to enable and disable queries when appropriate. However, as the *Database* cannot tell when the *Sensor* has been turned *on* or *off*, it cannot distinguish a first registration of data from others. Thus, it cannot enable and disable queries appropriately.

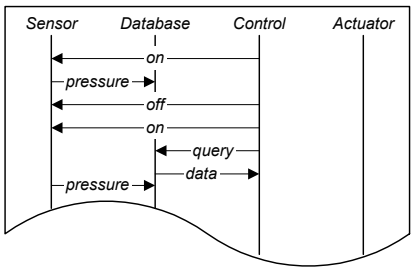

Figure 2 – Implied scenario

Succinctly, components do not have enough local information to prevent the system execution shown in Figure 2. Note that each component is behaving correctly according to some valid, but different, sequence of bMSCs. The *Sensor*, *Control* and *Actuator* are going through scenarios Initialise, Register, Terminate, Initialise, Analysis, and Register. However, the *Database* is performing Initialise, Register, Analysis, and Register. We will use the term "implied scenario" (taken from [1]) to refer to system traces such as the one shown in Figure 2.

Definition 6. (Implied Scenarios) Let *Spec* be a MSC specification with an alphabet L. An implied scenario is a word w over the alphabet L such that for all architecture models P of *Spec*, if $w.y\in mt(P)$ then $w\notin L(Spec)$

Implied scenarios are the result of a mismatch between system decomposition and system behaviour. They are not artefacts of a particular MSC language. Implied scenarios are the result of specifying the global behaviours of a system that will be implemented component-wise. They are also independent of the semantics given to hMSCs. Implied scenarios can arise when using strong sequential composition semantics, or even when the set of system traces is specified using regular expressions, or defined by extension.

2.3 Synthesis of Architecture Models

In [16] we present a synthesis algorithm that, given a MSC specification, builds a LTS model that is an architecture model of the MSC. In addition we prove that the resulting architecture model is minimal with respect to trace inclusion. The algorithm builds a LTS model for *each component* of the MSC specification, such that each component LTS has the same alphabet as its MSC counterpart. When all component models are composed in parallel, the resulting system exhibits all the behaviours described in the MSC. Furthermore, all traces that the composed component models can exhibit that are not specified in the MSCs can be proven to be implied scenarios.

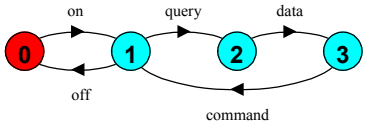

Figure 3 – LTS and FSP of *Control* component

Each component LTS model is constructed to be the minimal deterministic LTS that includes all projections of the MSC language on the components alphabet. The algorithm of [16] builds a behaviour model specification in the form of Finite Sequential Processes (FSP) [10]. FSP is the input language of the Labelled Transition System Analyser (LTSA) [10], which can be used to visualise the LTS for the component or for the complete system or to animate the system model. Furthermore, as we shall see, LTSA can be used to check if the model satisfies certain properties including existence of implied scenarios. In Figure 3 we show the LTS for the *Control* component. The LTS for the composed LTS, i.e. the architecture model, is shown in Figure 4.

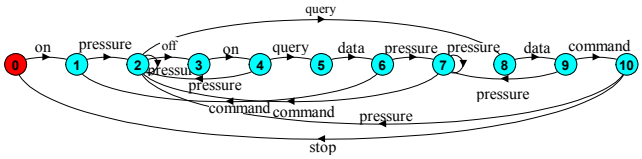

Figure 4 – LTS of architecture model

2.4 Synthesis of Trace Models

The architecture model for the Boiler Control system constructed by the synthesis algorithm mentioned above has at least one implied scenario (shown in Figure 2). In order to detect implied scenarios we compare the behaviour of the architecture model against a model that captures precisely the behaviour specified in the MSCs. We call the latter a *trace model*. Using LTSA the comparison would provide us with examples of behaviours that

the architecture model exhibits and that are not exhibited by the trace model. These examples correspond to implied scenarios.

Thus, the problem is to build an LTS that models exactly the system behaviour specified by an MSC specification. We omit a description of how this can be done due to space limitations. Readers can refer to [17] for a detailed explanation. In Figure 5 we show the behaviour actually described by the MSC specification. Note that the trace model in Figure 5 is not an architecture model as it is not the result of composing in parallel components that have the communication capabilities described in the scenario specification. The model of Figure 5 is a monolithic, system level description of the behaviour (ignoring the architectural aspects) of the scenario specification.

If the trace model is declared as a safety property in LTSA and the architecture model is model-checked to see if it satisfies the property, LTSA produces the output of Figure 6, which corresponds to the implied scenario depicted previously.

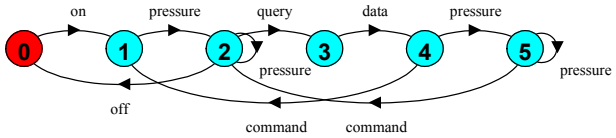

Figure 5 – Trace of MSC model

```
Trace to property violation in
ImpliedScenarioCheck:
        on
        pressure
        off
        on
        query
```

Figure 6 – LTSA output

3. SIMPLE NEGATIVE SCENARIOS

In previous sections, we have briefly discussed MSCs, architecture models and implied scenarios. We now motivate the need for documenting negative scenarios and for detecting implied scenarios in the presence of negative scenarios.

Let us suppose that the implied scenario of Figure 2 is taken back to stakeholders for validation. In addition, suppose that stakeholders understand that a *query* is retrieving old information from the *Database* because the last *pressure* reading is from a previous *Sensor* session. Consequently, they decide that the implied scenario is unacceptable system behaviour.

A first reaction could be to require stakeholders to review the MSCs they have provided and to correct them so as to avoid the mismatch that gives rise to the detected and rejected implied scenario. However, assuming scenarios provided by stakeholders were correct, the only way to avoid the implied scenario is through design. That is, stakeholders would have to contrive mechanisms such as additional synchronisation messages and/or new components to prevent the implied scenario from occurring.

This approach is unsatisfactory as stakeholders may well lack the technical skills to make such design decisions. Besides, even if stakeholders possess such design skills or if appropriately skilled people are called in to do the job, taking such a design decision at early-requirements stages may be undesirable. Additionally, stopping the process of eliciting requirements from stakeholders to solve design issues may be inappropriate.

We believe that documenting rejected implied scenarios and then producing more feedback in terms of implied scenarios can help to iteratively elicit new requirements. This iterative process results in a (positive) scenario-based specification that describes traces that future designs must provide, and negative scenarios, which serve as safety properties that those designs must satisfy. In the remainder of the section we define a simple form of negative scenario and explain how to detect implied scenarios when this form of negative scenarios is used.

We depict basic negative scenarios as bMSCs with three additional characteristics (see Figure 7). The first is that the last message is crossed out, we call it the proscribed message; the second is that the crossed out message is separated from previous messages by a dashed line. The part of the bMSC that is above the dashed line is called the precondition. The third difference is the black dot and arrow on the top part of the scenario. Intuitively, the meaning of basic negative scenarios is that once a trace described by the precondition has occurred, the *next* message cannot be the proscribed message. The dotted line is to convey that the precondition must occur completely before the crossed out message is proscribed (i.e. interleavings between the proscribed message and messages in the precondition are not considered). The black dot and arrow are to emphasise that the precondition describes system behaviour from its initial state. As an illustration, consider the negative scenario of Figure 7 that would result from the feedback given by stakeholders when rejecting the implied scenario depicted in Figure 2. This negative scenario specifies that the system may not exhibit traces with the following prefix: *on*, *pressure*, *off*, *on*, *query*.

It is clear that moving from the implied scenario to the negative scenario notation is very simple for whoever decides on rejection. This is our intention; simplicity in rejecting implied scenarios is crucial if we are to have a dynamic elicitation process.

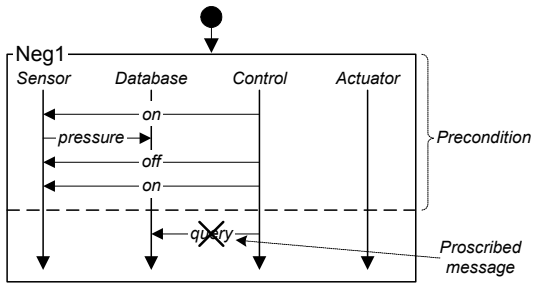

Figure 7 – A simple negative scenario

Formally, a negative scenario can be defined as a pair (*b, m*) where *b* models the scenario's precondition and *m* models the proscribed message.

Definition 7. (Simple Negative Scenarios) A negative scenario *Neg* is a pair (*p, m*) where *p* is a bMSC, and *l* is a message label. We shall use $\alpha(Neg)$ to denote $\alpha(p) \cup \{l\}$. The set of traces proscribed by *Neg* is $L(Neg) = \{w.l.w' \in \alpha(Spec)^* \mid w \in L(p)\}$.

In the remainder of this section, given a MSC specification *Spec* and an implied scenario *w* outputted by LTSA, we shall say that *Neg* is the negative scenario that results from rejecting *w* if *Neg=(p, l)* where *l* is the last message in *w* and *p* is a bMSC constructed with the remaining messages of *w*. For example, the negative scenario in Figure 7 results from rejecting the implied scenario in Figure 2.

How do simple negative scenarios alter the notions presented previously? Firstly, we need to extend MSC specifications to include simple negative scenarios. A MSC specification now comprises a positive and a negative part. The positive part consists of a high-level MSC and its corresponding basic MSCs (see definition of MSC specifications in previous section). The negative part consists of a set of simple negative scenarios. Additionally we will require message labels to denote message types both in positive and negative parts of a MSC specification.

In principle, the behaviour specified by a MSC specification could be considered as all the behaviours specified by the positive part, which are not proscribed by the negative part of the specification. However, this could introduce inconsistency as on one side a behaviour is being specified as acceptable (in the positive part) but on the other side the same behaviour is being specified as unacceptable (in the negative part).

Definition 8. (Inconsistent MSC Specifications) Given a MSC specification *Spec* with a positive part *Pos* and simple negative scenarios $Neg_1,\ldots,Neg_m$. The specification is said to be inconsistent if $L(Pos)\cap L(Neg_i)\neq\emptyset$ for some $1\leq i\leq m$.

The addition of a simple negative scenario need not result in an inconsistent MSC specification. In particular, if the simple negative scenario is the result of rejecting an implied scenario then no inconsistency is introduced. By definition, an implied scenario is not part of the specified behaviour, thus a simple negative scenario resulting from it will not subtract behaviours from the positive part of a MSC specification. In the Boiler Control example, the behaviour in Figure 6 is an implied scenario because no specified behaviour can start exhibiting the sequence *on, pressure, on, on, query*. Thus, it follows that the negative scenario of Figure 7 will not subtract any behaviour from that specified in Figure 1.

Property 1. Let *Spec* be a MSC specification with a positive part *Pos*. Let *w* be an implied scenario of *Spec* and *Neg* the simple negative scenario resulting from *w*. Adding *Neg* to *Spec* results in a consistent specification.

The preceding property makes simple negative scenarios useful when elaborating the system design. They can be thought of as properties that design models must satisfy. In our example, these models should not allow the following initial behaviour *on, pressure, on, on, query*.

In this paper we are interested in using negative scenarios to document rejected implied scenarios thus we shall be working with consistent MSC specifications. However, managing inconsistent specifications is a relevant issue that is receiving attention by researchers (e.g. [6]).

3.1 Implied Scenario Detection

Having documented, using the simple negative scenario notation from above, that a detected implied scenario corresponds to undesired system behaviour, it is crucial to verify if there are further implied scenarios that should be validated. We now describe how this can be done.

We already have procedures for constructing two LTS models: a minimal architecture model, which for the Boiler Control system we know contains at least one implied scenario, and a trace model, which captures precisely the behaviour specified in the MSC specification. We now need to constrain the architecture model in order to prevent it from exhibiting the new simple negative scenario. In this way when the constrained architecture model is compared against the MSC behaviour model, the LTSA model checker will verify if there are any more implied scenarios.

Thus, we now discuss how to build a LTS constraint from a simple negative scenario such that the LTS composed with an architecture model prevents the latter from exhibiting the simple negative scenario. For the negative scenario shown in Figure 7, the LTS that we build is that of Figure 8.

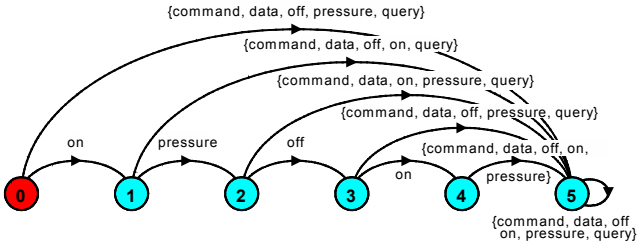

Figure 8 – Constraint for simple negative scenario of Figure 7

The LTS must keep track of the occurrence of the negative scenario's precondition and disallow the proscribed message from occurring. To do so, the LTS must have a state for each prefix of a trace specified in the precondition plus an additional state. The state modelling the empty prefix is the LTS initial state. The additional state models the case when there is no need to keep track of the behaviour, as the negative scenario cannot occur anymore. In Figure 8 the LTS states map to prefixes of the negative scenario as follows:

0	Empty prefix
1	*on*
2	*on, pressure*
3	*on, pressure, off*
4	*on, pressure, off, on*
5	Additional state

The LTS transitions are labelled with message labels and link states such that the prefix modelled by the target state is the concatenation of the prefix modelled by the source state and the transition's label. This ensures that the LTS keeps track of behaviours that could lead to the proscribed behaviour. For example, the transition from state 3 to state 4 is labelled *on* which is consistent with the prefixes modelled by the states.

The actual constraint is achieved by adding a transition from states modelling traces of the precondition to the additional state for each message label that is not proscribed message label. This is why in state 4, there is no enabled transition labelled *query*.

In addition, the LTS must not constrain any other behaviour except those specified by the simple negative scenario. Thus, for every state *s* that is proper prefix, and every message label *a* not enabled in *s*, a transition labelled *a* to the additional state is added. For instance, transition labelled *pressure* from state 3 to 5 has been added according to this criterion. More formally, the LTS constraint for a simple negative scenario is as follows:

Definition 9. (Simple Negative Scenario Constraint) Given a MSC specification with alphabet *L*, and a simple negative scenario $Neg=(p, l)$, the LTS constraint for *Neg* is (S, A, Δ, q) where:

- $S = P\cup\{e\}$ where $P = \{w \mid w \text{ is a prefix of } w' \in L(p)\}$.
- $q \in S$ such that $|q| = 0$.

- $\Delta \subseteq (S \times L \times S)$ such that $(s, a, s') \in \Delta$ if
  - $s \in P \backslash L(p)$, $s' \in P$, and $s.a = s'$, or
  - $s \in L(p)$, $s' = e$, and $a \neq l$, or
  - $s = s' = e$, or
  - $s \in P \backslash L(p)$, $s' = e$, and there is no $s'' \in P$ such that $s.a = s''$

The constraint c of a negative scenario *Neg* is to be composed in parallel the architecture model. As $tr(c) = L^* \setminus L(Neg)$, synchronisation on shared message labels results in a constrained architecture model that has lost only proscribed behaviours. This can be proved in a straightforward manner

Property 2. Let P be an architecture model of the positive part of a MSC specification *Spec*, and c the constraint of a simple negative scenario *Neg*. If $P' = (P \| C)$ is the parallel composition of P and c, then $mt(P') = mt(P) \backslash L(Neg)$.

We are thus guaranteed that if we compose an architecture model of a MSC specification with a LTS constraint of a simple negative scenario that is the result of a rejected implied scenario, then the constrained architecture model will continue to provide the specified behaviour of the positive part of the MSC specification and will not exhibit the behaviour of the rejected implied scenario.

Consequently, we can compare the constrained architecture model and the trace model with respect to maximal trace equivalence to detect further implied scenarios. In our example, if we compose the minimal architecture model of the Boiler Control system (see Figure 4) with the LTS constraint shown in Figure 8, and then check for maximal trace equivalence against the model of the MSC behaviour (see Figure 5) we obtain the following LTSA output.

```
Trace to property violation in
ImpliedScenarioCheck:
        on
        pressure
        query
        data
        command
        off
```

Figure 9 – LTSA output

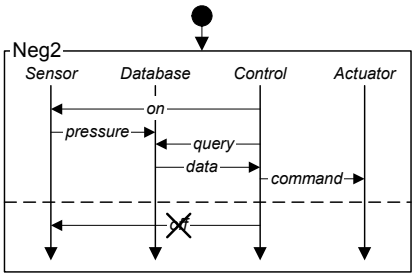

Figure 10 – Simple negative scenario

The newly detected implied scenario reveals another consequence of the mismatch between the specified behaviour and architecture: the *Control* component can terminate the *Sensor* session immediately after having queried the *Database*. This is not a specified behaviour as the hMSC in Figure 1 states that after an *Analysis* bMSC a *Register* bMSC must occur. As in the previous implied scenario, the causes lie in the components lack local information as to what is occurring at system level. The *Control* component cannot see when new information has been stored in the *Database* so it cannot tell when it is allowed to turn the *Sensor* off. The *Sensor* is not aware of queries to the *Database*, thus has insufficient information to disallow being turned off by *Control*.

As before, this implied scenario may or may not be acceptable system behaviour. Consequently, it must be validated with stakeholders; and according to response, a negative or positive scenario would be added to the MSC.

In order to continue exemplifying the use of negative scenarios let us assume that the new implied scenario is also deemed as undesired behaviour by stakeholders. Thus, we obtain a new negative scenario (Figure 10) from which a new constraint can be built to check for further implied scenarios.

4. AFTER/UNTIL NEGATIVE SCENARIOS

By repeating the process described above, the specification can be incrementally elaborated with positive and negative scenarios. One would expect to eventually reach a specification that has had all its implied scenarios validated. In other words that its architecture model constrained by all negative scenarios does not exhibit implied scenarios. However, with the simple negative scenario notation presented before, this is rarely the case.

Consider the MSC specification of the Boiler Control system of Figure 1 together with negative scenarios of Figure 7 and Figure 10. If we check for further implied scenarios we obtain the following LTSA output.

```
Trace to property violation in
ImpliedScenarioCheck:
        on
        pressure
        pressure
        off
        on
        query
```

Figure 11 – LTSA output

The implied scenario that LTSA has detected is a variation of the first implied scenario we encountered (Figure 6). The difference is that *two* pressure messages are occurring in this scenario. Clearly, if we add another negative scenario for this case, we will then find a similar implied scenario with *three* pressure messages. Consequently, this process will not converge to a specification with no implied scenarios.

Stakeholders may add several simple negative scenarios rejecting examples with increasing number of *pressure* messages. However, at some point, stakeholders will come up with a more abstract explanation of what these negative scenarios have in common that makes them unwanted behaviour. The number of *pressure* messages is irrelevant; the reason why these implied scenarios are unwanted behaviour is that *query* should never occur immediately after *on*; or more precisely, once the *Sensor* has been turned on, *query* should not occur until *pressure* has.

Simple negative scenarios are not powerful enough to express these kinds of restrictions on behaviour. Consequently, we need a more expressive negative scenario notation that allows one to express message interactions that should not occur within a context or scope. Thus, we define negative scenarios as having three sections called *After*, *Not* and *Until*. Sections *After* and *Until* are bMSCs that determine the scope in which the message in the *Not* section is proscribed. In Figure 12 we depict another negative scenario for the Boiler Control example. It states that after *on* has occurred, *query* is not allowed until *pressure* occurs.

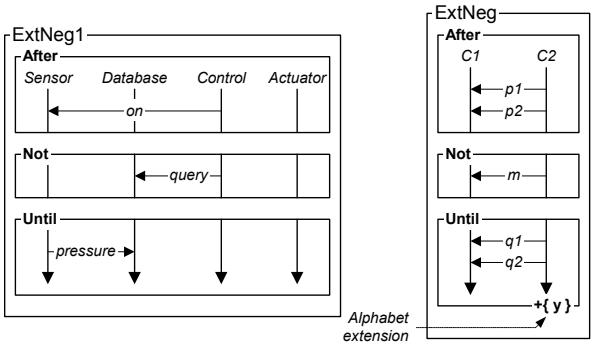

Figure 12 – A more expressive negative scenario notation

Figure 13 – Negative scenario with alphabet extensions

Although Figure 12 includes only one message in each section, multiple messages can be used in each section *After* and *Until*. Nonetheless, there can be only one message in the *Not* section. Multiple messages in a negative scenario section raise an important question regarding the semantics of negative scenarios. Consider Figure 13, it is clear that a sequence containing …*p1, p2, m*… is forbidden by it. However, what happens with …*p1, x, p2, m*…? Is the negative scenario forbidding this sequence? The semantics we give to our notation says the following: *After* and *Until* sections allow interleaving of message labels that are not in their alphabet. This means that, if message *x* differs from *p1* and *p2* then …*p1, x, p2, m*…is forbidden by the negative scenario as the occurrence of *x* is allowed in section *After*. In another example, …*p1, p2, q1, x, q2, m*…is not prohibited by the negative scenario because the *Until* section has occurred before *m*. The *Until* section does not have *x* in its alphabet, thus occurrences are allowed between *q1* and *q2*.

We also allow for explicit extension of the alphabet of *After* and *Until* sections. In Figure 13, the alphabet of the *Until* section has been extended with *y* (note the "+{*y*}" at the bottom of the *Until* section). This means that *m* is not allowed until *p1* and *p2* have occurred and *y* has not occurred in between them. Thus, …*p1, p2, q1, y, q2, m*…is prohibited by the negative scenario.

In short, both *After* and *Until* sections specify behaviours with respect to their own alphabet. We now define negative scenarios and their semantics formally. We use $m|_A$ to note the projection of word *m* on the alphabet *A*, and m[*i,j*] to note the sub-trace of *m* that starts on message *i* and ends on message *j*.

Definition 10. (Negative Scenarios) A negative scenario is a structure (a, E_a, l, u, E_u) where *a, u* are bMSCs, *l* is a message label and E_a, E_u are sets of message labels. In addition, we require $l \notin \alpha(u) \cup E_u$.

Definition 11. (Negative Scenario Semantics) Given a MSC specification with alphabet *L*, a negative scenario $Neg = (a, E_a, l, u, E_u)$, $A = \alpha(a) \cup E_a$, and $U = \alpha(u) \cup E_u$, the set of traces proscribed by *Neg* is defined as $L(Neg) = \{w \in L^* \mid \exists ijk: i<j<k, w[i,j]$ is a minimal sub-trace of *w* such that $w[i,j]|_A \in L(A)$, $w[k,k] = l$, and for all sub-trace *w*' of $w[j+1,k-1]$, $w'|_U \notin L(U)\}$.

As before, MSC specifications can be defined to have a positive part consisting of bMSCs and an hMSC, and a negative part consisting of negative scenarios. However, we no longer have any guarantees (as we did with simple negative scenarios) that certain negative scenarios originated from rejected implied scenarios will not introduce inconsistency. Note that the LTS constraints described in the next section can be used to detect such inconsistencies.

4.1 Implied Scenario Detection

We now describe how to verify if there are further implied scenarios when negative scenarios are introduced into a MSC specification. As before, we need to constrain an architecture model in order to prevent it from exhibiting behaviours proscribed by the negative scenario. In this way when a constrained architecture model is compared against its corresponding MSC behaviour model, the LTSA model checker will verify if there are any more implied scenarios.

We need to build a LTS from a negative scenario such that the LTS composed with an architecture model prevents the latter from exhibiting the negative scenario. The formal definition is shown below.

Definition 12. (LTS Constraint) Given an *After*/*Until* negative scenario $n = (a, E_a, l, u, E_u)$ and sets $A = \alpha(a) \cup E_a$ and $U = \alpha(u) \cup E_u$. The LTS constraint for *n* is $(S, A \cup U \cup \{l\}, \Delta, q)$ where:

a. $S = P_a \cup P_u$ where $P_a = \{w \mid w$ is a proper prefix of some trace in $L(a)\}$ and $P_u = \{(w_a, w_u) \mid w_a$ is a prefix of some trace in $L(a)$, and w_u is a proper prefix of a trace in $L(u)\}$.

b. $q \in P_a$ such that $|q| = 0$.

c. $\Delta \subseteq (S \times A \cup U \cup \{l\} \times S)$ such that $(s, m, t) \in \Delta$ if all the following hold:

   c1. If $s \in P_a$, and $s.m|_A \notin L(a)$, then *t* is the largest suffix of $s.m|_A$ in P_a

   c2. If $s \in P_a$, and $s.m|_A \in L(a)$, then $t = (t_a, t_u) \in P_u$, $t_a = s.m|_A$, and $|t_u| = 0$

   c3. If $s = (s_a, s_u) \in P_u$, $s_a.m|_A \notin L(a)$, $s_u.m|_U \notin L(u)$, then $t = (t_a, t_u) \in P_u$, t_a is the largest suffix of $s_a.m|_A$ in P_a, and t_u is the largest suffix of $s_u.m|_U$ in P_u

   c4. If $s = (s_a, s_u) \in P_u$, $s_a.m|_A \notin L(a)$, and $s_u.m|_U \in L(u)$, then $t = s_a.m|_A$

   c5. If $s = (s_a, s_u) \in P_u$, $s_a.m|_A \in L(a)$, and $s_u.m|_U \notin L(u)$, then $t = (t_a, t_u) \in P_u$, $t_a = s_a.m|_A$, and $|t_u| = 0$

   c6. If $s = (s_a, s_u) \in P_u$, $s_a.m|_A \in L(a)$, $s_u.m|_U \in L(u)$, and $m \in A$, then $t = (t_a, t_u) \in P_u$, $|t_u| = 0$, and $t_a = s_a.m$

   c7. If $s = (s_a, s_u) \in P_u$, $s_a.m|_A \in L(a)$, $s_u.m|_U \in L(u)$, and $m \notin A$, then *t* is the largest suffix of s_a in P_a

There are three main differences between the construction of constraints for *After*/*Until* negative scenarios and basic negative scenarios. The first important difference is that the LTS must now keep track of the *After* section, and then restrict the proscribed message from occurring while keeping track of the *Until* section. Thus the LTS must have states for prefix traces in the *After* and *Until* sections.

The second difference is that the *After* and *Until* sections do not predicate over traces that start at the system's initial state as preconditions do. Suppose, we are keeping track of the *After* section shown in Figure 13 and *p1* has occurred. The LTS will be in a state modelling prefix *p1* (State 1 in Figure 14). If *p2* occurs, then the LTS should move to a state modelling *p2*. This is similar to what happened when keeping track of preconditions. However, if *p1* were to occur, then the LTS must stay in the same state as it is, because an occurrence of *p2* would signify that the *After* section has been satisfied. For this reason rules c1 to c7 refer to

largest suffixes. Rule c1 states that if in state "*p1*", a transition labelled "*p1*" should lead to a state that models a maximal suffix of "*p1.p1*" that is a prefix of some trace specified by the *After* section. This prefix is *p1*.

The third important difference with respect to basic negative scenarios is that *After*/*Until* negative scenarios specify traces relative to the alphabets of their sections. For this reason, rules c1 to c7 project traces onto the alphabet of the section being tracked. For instance, c1 projects extensions of the prefix modelled by state *s* with the transition label onto the alphabet of the *After* section. Consequently, if in state "*p1*" a transition labelled *q1,* gives us the trace *p1.q1,* which projected onto the alphabet {*p1, p2*} results in *p1*. In other words, from state "*p1*" messages labelled *q1* are ignored.

Note that the states used for keeping track of the *Until* section are pairs (see P_u in rule a). This is because while keeping track of the *Unitl* section, the *After* section must be tracked too: it may be possible that a sequence of actions satisfying the *Until* section may include message labels of the *After* section. Consider the case where labels *q2* and *p1* are the same. Once the *Until* section is satisfied only *p2* is required to satisfy the *After* section again. Thus the LTS that keeps track of the *Until* section must also do so of the *After* section in order to know where to loop back to. Consequently, the states in the *Until* LTS model two prefixes. One is a prefix of a behaviour specified in the *Until* section, the other is a proper prefix of a behaviour specified by the *After* section.

In Definition 12, rule c1 tracks the *After* section and c2 is for when the *After* section has been completed, and the *Until* section must begin to be tracked while proscribing *l*. Rules c3, c5 and c6 are for tracking the *Until* and *After* sections, and c4 and c7 are for when the *Until* section has been satisfied and the LTS must go back to tracking the *After* section and allowing the occurrence of the proscribed message. The final LTS for the negative scenario of in Figure 13 is shown in Figure 14.

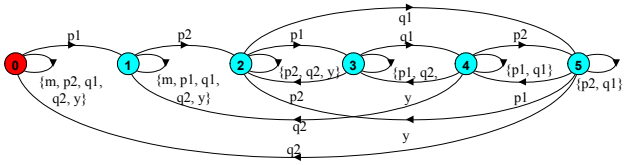

Figure 14 – Constraint for negative scenario in Figure 13

As with basic negative scenarios, we can prove that architecture models that are composed with constraints synthesised from *After*/*Until* negative scenarios will only have lost those traces that are proscribed by the negative scenario.

Property 3. Let *P* be an architecture model for the positive part of a MSC specification, and c the LTS constraint of an *After*/*Until* negative scenario *Neg* of the same specification. If $P' = (P \| C)$ is the parallel composition of *P* and *C*, then $mt(P') = mt(P) \backslash L(Neg)$.

Consequently, if we return to our Boiler Control example, we can constrain the architecture model with the LTS that corresponds to the negative scenario shown in Figure 12 (see Figure 15), and compare it against the trace model with respect to maximal trace equivalence to detect further implied scenarios. The LTSA output is shown in Figure 16.

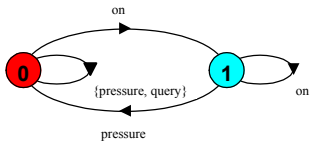

Figure 15 –Constraint for negative scenario in Figure 12

```
Trace to property violation in
ImpliedScenarioCheck:
        on
        pressure
        pressure
        query
        data
        command
        off
```

Figure 16 – LTSA output

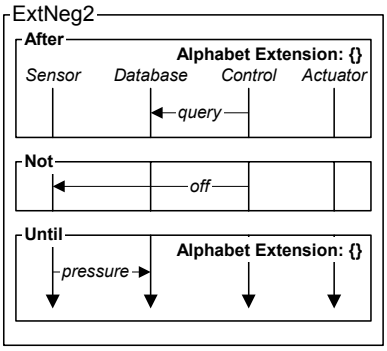

Figure 17 – Negative scenario

The newly detected implied scenario depicted in Figure 16 is similar to that of Figure 10. Again the difference is that two pressure messages are occurring in this scenario. If a more expressive scenario such as the one in Figure 17 is added to the MSC specification together with the other three negative scenarios for the Boiler Control system, LTSA reports that the trace model and the constrained architecture are equivalent with respect to maximal traces. Thus, we can conclude that we have found all implied scenarios.

We now have a precise description of the intended system behaviour and of two abstract properties that the architecture must satisfy. In subsequent stages of the development process, design decisions shall need to be made in order to provide the required behaviour while preventing negative scenarios from occurring. If these designs are modelled with behaviour models, then designers can verify the correctness of their decisions against the requirements and properties expressed in the MSC specification. Furthermore, the architecture model can serve as the basis for developing and reasoning about these design decisions.

5. TOOL SUPPORT

In previous papers [16, 17] we have described how we extended the Labelled Transition System Analyser (LTSA) tool [10] to support positive MSC specifications, architecture model synthesis, MSC behaviour synthesis and implied scenario detection. We have extended the tool to support both kinds of negative scenarios and the construction of behaviour constraints as described in Section 4.

The tool, implemented in Java, allows push-button implied scenario detection from MSC specifications provided in textual format [7]. Algorithms build FSP specifications and properties

[10], which can be model-checked by LTSA. The architecture, together with some examples (including the one used throughout this paper), is available at `http://doc.ic.ac.uk/~su2/`.

6. DISCUSSION AND RELATED WORK

In this paper we have presented a notation and semantics for negative scenarios. Our motivation has been to provide a language for documenting rejected implied scenarios and eliciting further implied scenarios. We have presented two negative scenario notations. The first is a very simple notation that allows stakeholders to rapidly reject implied scenario in order to continue checking for more implied scenarios in their specification. With this simple notation, the process of rejecting implied scenarios may not converge to a specification with no implied scenarios. Additionally, the limited expressiveness of this notation may lead to stakeholders repeatedly encountering implied scenarios that refer to a same aspect of the problem domain. By identifying implied scenarios that correspond to a same class of unacceptable behaviours, stakeholders can elaborate more abstract negative scenarios that capture the essence of why the scenarios should be rejected. The second negative scenario notation has been designed for this purpose; it allows specifying *After*/*Until* constraints that prohibit a particular message from occurring within a scope. Our initial experimentation indicates that the expressiveness of this notation is enough to allow MSC specifications to converge to a state where no more implied scenarios can be found.

The negative scenario language here defined has not been designed to be used in conjunction with positive scenarios to narrow down the expected behaviour of a system. Although it can be used for such purpose, it has been designed for documenting rejected implied scenarios and facilitating the elaboration of abstract system properties. Thus, our negative scenarios may not constrain behaviour specified in positive scenarios. Furthermore negative scenarios make sense independently of the existence of positive scenarios. Consequently, it differs from other approaches that introduce notations for constraining behaviour. For example, in LSCs [4] hot and cold conditions are used within the positive specification to restrict the occurrence of positive behaviours.

The *After*/*Until* notation is inspired on work on property specification patterns [14]; however, we have used MSC syntax to stay within the scenario-based approach facilitating the adoption of negative scenarios by scenario users. We have not used temporal logic for negative scenarios for the same reason and because of the difficulty of describing sequences of events in these logics. Regular expression can be used to describe sequences of events in an elegant manner. However, in contrast with bMSCs, they can become cumbersome for describing partial ordering of events [5].

By providing two negative scenario notations, our intention is to facilitate the jump from rejecting individual behaviours to the formulation of more abstract safety properties. This is also the intention of Van Lamsweerde et. al. [9], who aim to infer declarative requirements from scenarios in order to link operational descriptions with goal-based specifications. However, there are significant differences. Firstly, their use of negative scenarios and exclusion rules is to avoid over-generalisations resulting from the inference rules applied to positive behaviours. This is different from our use of negative scenarios that are used to document mismatches (not over-generalisations) between positive behaviour and architecture. In addition, we integrate negative scenarios into the behaviour models with no need for stakeholder manipulation. Exclusion rules may however require weakening through human intervention to avoid introducing inconsistencies. Moreover, this weakening must be done in the logic level instead of the scenario level. Finally, the more complex a scenario is, the more candidate goal specifications generated are likely to be irrelevant to some specific concern. Thus, several rules (that must be applied by stakeholders) are provided for cleaning up scenarios. Implied scenarios are relevant issues regarding the concurrent nature of the system being specified and that result from a partial specification of such system. These issues should be addressed and result in an extended system description or, in the case of rejected implied scenarios, in relevant system safety properties. We believe that van Lamsweerde's approach is complementary to ours. While we focus on elaboration of partial operational scenario descriptions, and the construction of models to enable reasoning on design, van Lamsweerde et al. focus on the elaboration of declarative requirements and reasoning on goals.

The notion of implied scenario has been studied by Alur et. al. [1], for a subset of the MSC language presented here. In [1] only bMSCs are allowed; however, communication between components is considered to be asynchronous. In [3], a related problem called non-local branching choice is studied. In this case authors are interested in checking when components can choose different scenarios in a scenario graph. We are interested in observable behaviour, thus the notion of implied scenarios is defined more abstractly, in terms of traces of observable events and not on MSC specifications. Interestingly, non-local choices are special cases of implied scenarios.

Protocol synthesis research (e.g. [13]) studies the automated construction of distributed implementations from a given service specification. Thus, the problem of avoiding unspecified behaviours due to uncoordinated components is crucial. In our setting, service specifications are provided as MSCs and implied scenarios are the unspecified behaviours that arise due to the chosen distributed architecture. However, we do not assume that the service specification is complete. Thus, an implied scenario may not necessarily be undesired behaviour that is to be avoided. Consequently, our focus on implied scenario detection and documentation of rejected implied scenarios helps drive requirement elicitation rather than focus purely on automating design techniques that avoid implied scenarios.

Harel et al. [4] propose a method for decomposing a global automaton synthesised from a LSC specification into distributed components. However there is no guarantee that the decomposition will comply to the architecture given in the LSC specification. We believe and work on the premise that architecture constraints should be considered upfront when building behaviour models.

The idea of iterative construction of scenario-based specifications by presenting stakeholders with new scenarios has also appeared in the work of Mäkinen and Systä [11]. Their work focuses on a component level, presenting stakeholders with component behaviour that may be the result of over-generalisation. Our work is on a system level, presenting stakeholders with system behaviour that is the result of a mismatch between behaviour and architecture

There has been much work on synthesis of behaviour models from scenario-based specifications. However, many of these approaches do not make a distinction between specification and implementation. In one sense, these approaches consider the scenario specification to be more of a design language that uniquely determines an implementation up to a certain level of abstraction. We consider that a scenario-based specification can have many implementations. Many approaches provide algorithms for generating statechart models from MSCs [8, 11, 18]. Others provide a formal semantics based on state machines such as the one provided by Cobens et al. [7], which is part of the Z.120 recommendation for MSCs. Along these lines, we have presented in [15] an MSC language that integrates these kinds of approaches by providing a simple mechanism for tailoring MSC specifications to specific interpretations by the use of state labels.

Another line of research is the development of methods and tools for analysis and verification of scenario-based specifications. There are a number of efforts in this direction focusing on verification of specific system properties such as process divergence [3], timing conditions [2], and more general properties based on recognising patterns of communication [12]. Our work focuses on a different aspect, existence of implied scenarios, which is complementary to other efforts in analysis and verification of scenario-based specifications.

7. CONCLUSIONS AND FUTURE WORK

We have presented a language for documenting negative scenarios in MSC specifications and developed automated procedures for implied scenario detection in the presence of such scenarios. Our motivation has been to provide a language for documenting rejected implied scenarios, and to develop techniques that help elicit further requirements by presenting stakeholders with implied scenarios. By defining two notations for negative scenarios we have provided stakeholders with the following: a means to rapidly reject feedback from our implied scenario detection tool in order to continue checking for more implied scenarios; and a means to elaborate abstract negative scenarios that specify system behavioural properties. We have also developed synthesis algorithms for building constraints from negative scenarios. These algorithms have been implemented within the LTSA tool providing push-button implied scenario detection and support for scenario-based specification and behaviour model elaboration.

Our current work is mainly focused on validating our approach in industrial strength case studies. Additionally, we are implementing a GUI interface for editing MSC specifications. We intend to develop a front-end tool supporting requirement elicitation based on implied scenario detection that uses LTSA as a back-end verification engine.

8. REFERENCES

[1] Alur, R., K. Etessami, and M. Yannakakis. *Inference of Message Sequence Charts* in *22nd IEEE International Conference on Software Engineering (ICSE'00)*. 2000. Limerick.

[2] Alur, R., G.J. Holzmann, and D. Peled. *An Analyser for Message Sequence Charts* in *2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*. 1996. Passau: Springer.

[3] Ben-Abdhallah, H. and S. Leue. *Syntactic Detection of Process Divergence and Non-Local Choice in Message Sequence Charts* in *3rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*. 1997: Springer.

[4] Harel, D. and W. Damm. *LSCs: Breathing Life into Message Sequence Charts* in *3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*. 1999. New York: Kluwer Academic.

[5] Holzmann, G.J., *Protocol Synthesis (Chapter 10)*, in *Design and Validation of Computer Protocols*. 1990, Prentice Hall.

[6] Hunter, A. and B. Nuseibeh, *Managing Inconsistent Specifications: Reasoning, Analysis and Action.* ACM Transactions on Software Engineering and Methodology, 1998. **7**(4): p. 335-367.

[7] ITU, *Message Sequence Charts*, 1996, International Telecommunications Union. Telecommunication Standardisation Sector.

[8] Krüger, I., *et al.*, *From MSCs to Statecharts*, in *Distributed and Parallel Embedded Systems*, F.J. Rammig, Editor. 1999, Kluwer Academic Publishers. p. 61-71.

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

[10] Magee, J. and J. Kramer, *Concurrency: State Models and Java Programs*. 1999, New York: John Wiley & Sons Ltd.

[11] Mäkinen, E. and T. Systä. *MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML,* in *23rd IEEE International Conference on Software Engineering (ICSE '01)*. 2001. Toronto.

[12] Muscholl, A., D. Peled, and Z. Su. *Deciding Properties of Message Sequence Charts* in *1st International Conference on the Foundations of Software Science and Computation Structure, (FOSSACS'98)*. 1998. Lisbon: Springer.

[13] Saleh, K., *Synthesis of Communication Protocols.* Computer Communication Review, 1996. **26**(5): p. 40-59.

[14] Smith, R.L., *et al. Propel: An Approach Supporting Property Elucidation* in *IEEE International Conference on Software Engineering (ICSE'02)*. 2002. Orlando.

[15] Uchitel, S. and J. Kramer. *A Workbench for Synthesising Behaviour Models from Scenarios* in *International Conference on Software Engineering (ICSE'01)*. 2001. Toronto: IEEE Computer Society.

[16] Uchitel, S., J. Kramer, and J. Magee. *Detecting Implied Scenarios in Message Sequence Chart Specifications* in *Joint 8th European Software Engineering Conference (ESEC'01) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'01)*. 2001. Vienna: ACM Press.

[17] Uchitel, S., J. Kramer, and J. Magee. *Detecting Implied Scenarios in the Presence of Behaviour Constraints* in *International Workshop on Validation and Implementation of Scenario-based Specifications (VISS'02) a satellite workshop of the 5th European Joint Conferences on Theory and Practice of Software (ETAPS'02)*. 2002. Grenoble: Electronic Notes in Theoretical Computer Science.

[18] Whittle, J. and J. Schumann. *Generating Statechart Designs from Scenarios* in *22nd International Conference on Software Engineering (ICSE'00)*. 2000. Limerick, Ireland.