Detecting Implied Scenarios in MSCs Using LTSA.

(Draft Version)

Departmental Technical Report: 2001/4

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 becoming increasingly popular as part of a requirements specification. Scenarios describe how system components, the environment and users working concurrently interact in order to provide system level functionality. Each scenario is a partial story which, when combined with other scenarios, should conform to provide a complete system description. However, it is not always possible to build a set of components that provides exactly the same system behaviour as described with a set of scenarios. Implied scenarios may appear as a result of unexpected component interaction.

In this paper, we present an algorithm that builds a behaviour model that describes the closest possible implementation for a specification based on basic and high-level MSCs. We also present a technique for detecting and providing feedback on the existence of implied scenarios. We have integrated these procedures into the Labelled Transition System Analyser, which allows for model checking and animation of the behaviour model.

Keywords

Synthesis, message sequence charts, implementability, labelled transition systems, FSP, LTSA.

1. INTRODUCTION

Scenarios are becoming increasingly popular as tools for requirement elicitation and specification. Scenarios describe how system components, the environment and users interact in order to provide system level functionality. Each scenario is a story which, when combined with all other scenarios, should conform to provide a complete system description. Their simplicity and intuitive graphical representation allows stakeholder involvement and helps to build a common ground with the developers. Besides, as they are partial system descriptions, stakeholders can develop descriptions independently, contributing their own view of the system to those of other stakeholders.

The components participating in scenario-based specifications are assumed to work independently synchronising through message exchange. The resulting systems, called concurrent systems, are amenable to analysis through the construction of behaviour models. A behaviour model can be used as a precise specification of intended behaviour of a system, as a prototype for exploring the system behaviour and also to allow for automated checking of model compliance to properties (model checking). Numerous tools that allow model checking and animation of behaviour models exist (e.g. [6, 8]).

Our objective is to facilitate the development of behaviour models in conjunction with scenario-based specifications. Such models are complementary and provide an alternative view of how system components interact. In particular, we believe that there is benefit to be gained by experimenting with and replaying analysis results from behaviour models in order to help correct, elaborate and refine scenario-based specifications.

Scenario specifications depict a set of acceptable system behaviours and show how these behaviours are shared out among the system components. This information can be used to synthesise [2, 10, 14] a behaviour model for a possible implementation of such system. Initially one would expect the model to have exactly the same set of behaviours as those depicted in the scenario specification. However, this is not always the case. Scenarios can combine in unexpected ways and certain system behaviours not present in the scenario specification may appear in all possible implementations of the system. We call these behaviours implied scenarios, and they arise because components have a local view of what is happening in the system. If this view has insufficient information, a component may behave incorrectly in terms of the expected behaviour at a system level.

The existence of implied scenarios is an indication of unexpected system behaviour, and detecting them is a relevant issue. An implied scenario may simply mean that an acceptable scenario has been overlooked and the scenario specification needs to be completed. However, the implied scenario may represent an unacceptable behaviour and therefore imply a need to modify the scenario specification to avoid the undesired situation.

In this paper, we present a framework for synthesising implementation models for scenario-based specifications and for detecting the existence of and providing feedback on implied scenarios. Implied scenarios have been studied by Alur et al. [1] for a restricted scenario language. The issue of constructing an implementation and finding implied scenarios is limited to a set of message sequence charts (MSCs) that specify a finite set of (finite) system behaviours. We extend their work by providing a framework for a more expressive scenario language that provides for high-level MSCs for specifying an infinite number of (possibly infinite) system behaviours.

Our previous work on scenario-based languages [12] differs significantly from the approach presented in this report. In [12] we consider scenario-based languages as design languages, a view similar to work of other authors [2, 3, 10, 14]. The semantics of a scenario specification is defined directly in terms of labelled transition systems, in other words the scenarios specify how components should by designed. In this paper we consider a scenario specification as describing system behaviour and not a system design, thus the relevant issue of finding an adequate design for the specification.

The goal of this report is to present our framework for synthesising behaviour models for scenario-based specifications and detecting implied scenarios. We also aim to provide a proof for the results discussed in the report. In Section 2, we present a scenario-based specification language that uses basic and high-level MSCs. Section 3 describes a procedure for synthesising a behaviour model of an implementation for MSC specifications. In addition, we show the synthesised implementation model to be the closest possible to the specified system. Section 4 introduces the notion of implied scenario and in Section 5 we present a method for detecting the existence of implied scenarios. Section 6 discusses our implementation that integrates with the Labelled Transition System Analyser tool. Finally, in Sections 7 and 8 we discuss related work, conclusions and future work.

2. MESSAGE SEQUENCE CHARTS

In this section, we briefly describe message sequence charts (MSCs). We also introduce the example that is used to illustrate our approach. This example 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 sensor information while the control unit performs calculations and commands the actuator.

The language is a subset of the MSC ITU language [7]. A basic MSC (bMSC) describes a finite interaction between a set of components (see top of Figure 1). Each vertical line represents a component and is called an instance. 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. Placing MSC events (message inputs and message outputs) further down on an instance means that they occur later on.

Definition 1. (Basic Message Sequence Charts) A basic message sequence chart (bMSC) is a structure $(E, L, I, \lambda, <, tgt)$ where:

- E is a set of events partitioned into a set S of send events and a set R of receive events.
- L is a finite set of labels.
- I is a finite set of instances.
- $\lambda : E \rightarrow L \ x\ I$ maps events to their labels and instances. We define $i(E)$ to be the set of events e such that $\lambda(e) = (l, i)$.
- $<$ is a set of total orders $<_i \subseteq (i(E)\ x\ i(E))$ where $i \in I$. We define $\leq$ to be the transitive closure of $\cup <_i \cup\ tgt \cup tgt^{-1}$
- $tgt: S \rightarrow R$ is a bijective function that maps send events to receive events.

We will note $lbl(e) = l$ and $inst(e) = I$ if $\lambda(e) = (l, i)$.

For simplicity, throughout the paper, 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 synchronous we require arrows to be drawn horizontally and do not allow components sending messages to themselves.

Definition 2. (Consistent bMSCs) Let $b = (E, L, I, \lambda, <, tgt)$ be a bMSC, $s, s' \in S$ be send events and $r, r' \in R$ be receive events. We say that b is consistent if:

- $lbl(s) = lbl(tgt(s))$ *(connected events have same label)*
- $inst(s) \neq inst(t(s))$ *(instances cannot send messages to themselves)*
- if $s \leq r$ and $r \leq s$ then $tgt(s) = r$ *(message lines do not cross)*

The behaviour of a bMSC is a set of sequences of message labels. The set is determined by the casual precedence of events of the bMSC. Events are totally ordered within instances and are considered to occur simultaneously if the receive event corresponds to the send event (s and $tgt(s)$). This casual relation determines a partial order of events ($\leq$). Thus, any sequence of send events that respects this partial order gives rise to an acceptable behaviour of the bMSC. For example the behaviour of the bMSC *Analysis* of Figure 1 comprises only one sequence of labels: *Query*, *Data*, *Command*.

Definition 3. (Linearisations) Let $b = (E, L, I, \lambda, <, tgt)$ be a bMSC. A word $l_1, \ldots, l_{|S|}$ over the alphabet L is a linearization of b iff there is a word $s_1, \ldots, s_{|S|}$ over the alphabet S such that:

- $lbl(s_i) = l_i$ for $1 < i < |S|$.
- If $s_i \leq s_j$ then $i \leq j$.

Let w be a linearisation of b, we define $w|_i$ to be the projection of w on the alphabet of i.

Definition 4. (bMSC Languages) Let $b = (E, L, I, \lambda, <, tgt)$ be a bMSC. We define the language of b as a set $L(b)$ of words over the alphabet L, where $L(b) = \{w \mid w$ is a linearization of $b\}$.

A high-level MSC (hMSC) provides the means for 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 triangle. An MSC specification is a set of bMSCs and a hMSC. For simplicity we assume bMSCs have the same set of instances and that message labels are used consistently throughout the bMSCs.

Definition 5. (High-level Message Sequence Charts) A high-level message sequence chart (hMSC) is a graph of the form (N, A, s_o) where:

- N is a set of nodes.
- $A \subseteq (N\ x\ N)$ is a set of arrows.
- $s_0 \in N$ is the initial node.

A (possibly infinite) sequence of nodes $w = n_0, n_1, \ldots$ is a path if $n_i \in N$, $n_0 = s_0$, and $(n_i, n_{i+1}) \in A$ for $0 \leq i < |w|$. We say a path is maximal if it is not a proper prefix of any other path.

We can now define a MSC specifications. They consist of a set of bMSCs, a hMSC and a bijective function that maps every node in the hMSC to a bMSC. We shall assume that all bMSCs have the same set of instances and that message labels are used consistently throughout them.

Definition 6. (Message Sequence Chart Specifications) A message sequence chart (MSC) specification is a structure *(B, H, f)* where:

- *B* is a set of bMSCs.
- *H=(N, A, s_0)* is a hMSC.
- *$f : N \rightarrow B$* is a bijective map from hMSC nodes to bMSCs. We shall denote *$\alpha(i)$* the set of labels that can be received or sent by an instance *i*.

Definition 7. (Consistent MSC Specifications) Let *Spec = (B, H, f)* be a MSC specification. We say that *Spec* is consistent if:

- All bMSCs in *B* have the same set of instances.
- If $lbl_b(s) = lbl_{b'}(s')$ then $inst_b(s) = inst_{b'}(s')$ *(message labels characterize a unique sender)*
- If $lbl_b(r) = lbl_{b'}(r')$ then $inst_b(r) = inst_{b'}(r')$ *(message labels characterize a unique receiver)*

The behaviour of a MSC specification is also given by a set of sequences of message labels. The hMSC together with the mapping of nodes to bMSCs show how the system can evolve from one scenario to another. There are two usual interpretations of this evolution. The first is to assume that all components wait until all events of the previous bMSC have occurred before moving on to the next bMSC. This implies that there is some kind of implicit synchronisation scheme that components use in order to know when a scenario is completed. We believe that this is not a reasonable assumption and adopt the second and more accepted approach in which components move into subsequent scenarios in an unsynchronised fashion. We define the notion of sequential composition (or concatenation) of bMSCs accordingly.

Definition 8. (Sequential Composition of bMSCs) The sequential composition of two bMSCs *$b = (E, L, I, \lambda, <, tgt)$* and *$b' = (E', L', I', \lambda', <', tgt')$* is denoted *$(b \bullet b')$* and is defined by the bMSC *$(E \cup E', L \cup L', I \cup I', \lambda' \cup \lambda', <<, tgt \cup tgt')$*, where $<<$ is a set of total orders $<<_i$ such that $i \in I$, $<<_i = <_i \cup <_i' \cup \{(max(i(E), min(i(E'))\}$.

For example the bMSCs *(Analysis • Register)* determines two possible sequences of events *Query, Data, Pressure, Command* and *Query, Data, Command, Pressure*. These sequences occur because message *Pressure* is independent of *Command,* there is no causal relation between the send events of these messages in *(Analysis • Register)*. This is not surprising, they involve different components, and thus any interleaving of these messages could happen.

The sequences of event labels determined by a MSC specification are those belonging to the language of any maximal concatenation of bMSCs allowed by the hMSC.

Definition 9. (Maximal bMSCs) Let *Spec = (B, H, f)* be a MSC specification. We say that a *b* is a maximal bMSC in *Spec* if there is a maximal path $n_1, n_2, \ldots$ in *H* such that $b = f(n_1) \bullet f(n_2) \bullet \ldots$

Definition 10. (MSC Specification Languages) Let *Spec = (B, H, f)* be a MSC specification with a set of instances *I*. We define the language of *Spec* as a set *L(Spec)* of words, where $L(Spec) = \{w \in L(b) \mid b$ is a maximal bMSC of $Spec\}$. We also define $L(Spec)|_i = \{w|_i \mid w \in L(Spec)\}$

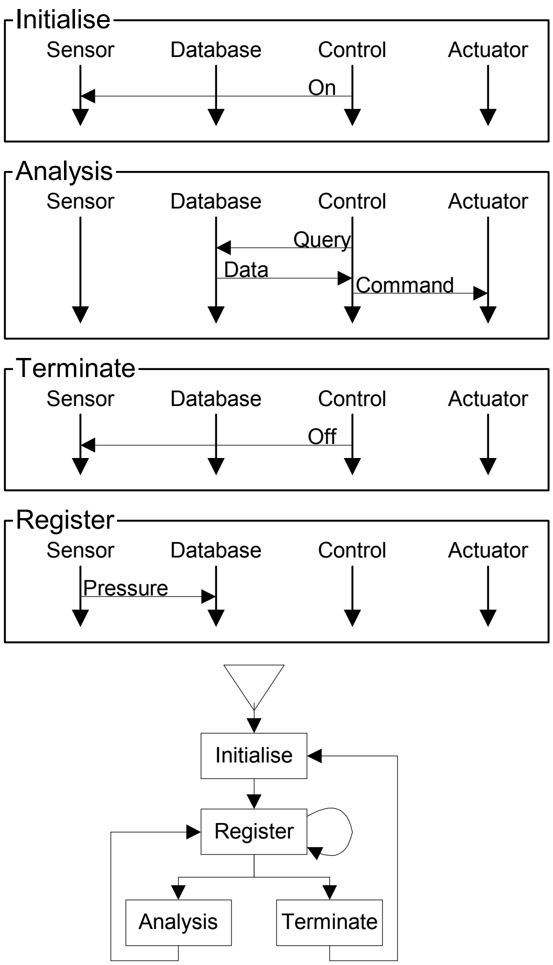


Figure 1. Message sequence chart specification.

3. IMPLEMENTATION SYNTHESIS OF MSC SPECIFICATIONS

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. In this section we shall show how this can be done and explain to what degree these components can comply with the original specification. We shall model components as labelled transitions systems where labels represent messages that the components can input and output. We consider the system they conform as the parallel composition of all components. In other words the system is the result of putting components together and forcing them to synchronize on all common message labels. For a detailed explanation of LTS and parallel composition refer to [8].

Definition 11. (Labelled Transition Systems) A finite labelled transition systems (LTS) *P* is a structure *(S, L, Δ, q)* where:

- *S* is a set states.
- $L = \propto(P) \cup \{\tau\}$ is a set of labels where $\propto(P)$ denotes the alphabet of *P* and τ denotes internal actions that cannot be observed by the environment of an LTS.

- $\Delta \subseteq (S \text{ x } L \text{ x } S)$. We will write $s \xrightarrow{l} t$ if $(s, l, t) \in \Delta$.
- *$q \in S$ is the initial state.*

Definition 12. (Parallel Composition of LTS) The parallel composition of two LTSs P_1 and P_2 where $P_i = (S_i, L_i, \Delta_i, q_i)$ is denoted $P_1 || P_2$ and is defined by the LTS (S, L, Δ, q) where:

- $S = S_1 \text{ x } S_2$
- $L = L_1 \cup L_2$
- $q = (q_1, q_2)$
- Δ is the smallest relation satisfying the rules

$$\frac{s \xrightarrow{a}_1 t}{(s,s') \xrightarrow{a} (t,s')} (a \notin L_2) \qquad \frac{s \xrightarrow{a}_2 t}{(s',s) \xrightarrow{a} (s',t)} (a \notin L_1)$$

$$\frac{s \xrightarrow{a}_1 t \quad s' \xrightarrow{a}_2 t'}{(s,s') \xrightarrow{a} (t,t')} (a \in (L_1 \cap L_2) \setminus \{\tau\})$$

Parallel composition is both commutative and associative. Consequently, we will note parallel composition of multiple LTSs as follows: $P_1 || ... || P_n$

Definition 13. (Hiding) The hiding of a set of labels H in a LTSs $P = (S, L, \Delta, q)$ is denoted $P_1 \backslash H$ and is defined by the LTS $(S, L \backslash B \cup \{\tau\}, \Delta', q)$ where Δ' is the smallest relation satisfying the rules

$$\frac{s \xrightarrow{a} t}{s \xrightarrow{\tau} t} (a \in B) \qquad \frac{s \xrightarrow{a} t}{s \xrightarrow{a} t} (a \notin B)$$

Given a LTS we wish to compare the executions it models with the executions specified in a MSC specification. Thus we introduce the notion of trace.

Definition 14. (Traces) Let $P = (S, L, \Delta, q)$ be a LTS. A (possibly infinite) word $w = l_1, l_2, ...$ over the alphabet L is a trace of P iff there is a word $q_1, q_2, ...$ over the alphabet S such that:

- $(q_i, l_i, q_{i+1}) \in \Delta$ for $0 < i < |w|$
- $q_1 = q$

We say a trace is maximal if it is not a proper prefix of any other trace. Let w be a trace of P, we define $w|_i$ to be the projection of w on the alphabet of P_i. We also define $L(P) = \{w \mid w$ is a maximal trace of $P\}$.

The weakest condition that one can require from an implementation of a MSC specification is that it must comprise a component for each instance, that each component must have the interface determined by the specification (i.e. inputs and outputs according to the send and receive events of its instance) and that the complete system must be able to execute all the sequences determined by the specification.

Definition 15. (Implementations) Let $Spec = (B, H, f)$ be a MSC specification with instances I, and P a LTSs resulting from the parallel composition of LTSs P_i with $i \in I$. We define P to be a implementation of *Spec* if

- $\propto(P_i) = \propto(i)$
- $L(Spec) \subseteq L(P)$.

The algorithm of Figure 2 can be used to build a LTS for each component specified in a MSC specification. Furthermore, if the LTS models for all the components are composed in parallel, then we obtain an implementation of the MSC specification.

```
void Synthesise(Specification S, Component c) {
   Grammar G = new Grammar();
   ForEach bMSC b in S.getbMSCs()
      G.add(buildProduction(S, c, b);

   G.removeUnreachableNonTerminals();
   print "\\-----" + c.name()+ "-----";
   print "determinisitic " + c.name()+ " = ";
   print Continuations(S, "Init");
   ForEach Production p in G
      print p.getLeftHandSide()+ "= (";
      print p.getRightHandSide();
      print Continuations(S,p.getLeftHandSide());
      print ")";
   print "/{hiddenAction}.";
}

Production buildProduction(Specification S,
                           Component c, bMSC b) {
   Production p = new Production();
   String s = "";
   p.setLeftHandSide(b.name());
   Instance i = b.getInstance(c);
   for (int a = 0; a<i.size();a++) {.
      s = s + i.getEvent(a).label() + "->";
   p.setRightHandSider(s);
   return p;
}

String Continuations(Specification S, String
name)
   SetOfbMSCs C=S.getMaximalContinuations(name);
   if (C.size() == 0)
      return "STOP";
   else if (C.size() == 1)
      return C.getElement().name();
   else
      String s = "(";
      ForEach bMSC b in Cont
        s = s +"silentAction->" + b.name() + "|";
      s = s + ")";
   return s;
}
```

Figure 2. Synthesis Algorithm.

```
Initialise: on
Register:
Terminate: off
Analyse: query->data->command
```

Figure 3 – Initial FSP productions for *Control*

The algorithm works by translating the MSC specification into a behaviour model specification in the form of Finite Sequential Processes (FSP) [8], which is the input language of the Labelled Transition System Analyser (LTSA) [8]. Using LTSA one can visualise the LTS for each component for the complete system or animate the system model. Furthermore, as we shall see, LTSA can be used to check if the model satisfies certain properties [9].

We present a simplified version of the algorithm in [12]. Main differences are due to the fact that this version does not have to deal with state labels in MSC specifications. Although the version in [12] can be used for the current approach, the algorithm presented here produces clearer FSP productions.

The algorithm synthesises one component at a time, and we shall present it briefly by applying it to component *Control* of Figure 1.

Given a MSC specification and a component to be synthesised, the algorithm constructs a deterministic FSP process that can has the same language as the projection of the MSC language on the alphabet of the component. First, the algorithm builds one FSP production for each bMSC in the specification. The non-terminal on the left hand side of the production is the name of the bMSC, while the right hand side is the sequence of events (reading top-down) that the component's instance can perform. The productions generated for the *Control* component are show in Figure 3.

Second, the algorithm calculates the maximal continuations for each bMSC. A bMSC b_2 is a continuation of b_1 (denoted $b_1 \Rightarrow b_2$) if it possible to get to b_2' from b_1 in through one edge on the hMSCs or if there is a b_3 such that $b_2 \Rightarrow b_3$, $b_3 \Rightarrow b_1$ and the component's instance in b_3 has no events. To exemplify, we calculate the continuations of bMSC *Initialise* for component *Control.* According to the hMSC of Figure 1, bMSC *Register* is a continuation of *Initialise*. However, as *Control* does not participate in *Register*, the bMSCs *Analysis* and *Terminate* are also continuations of Analysis. Consequently, we have *Analysis, Terminate* and *Register* as continuations of *Initialise*. A maximal continuation of b_1 is a bMSC b_2 such that $b_1 \Rightarrow b_2$ and for all b_3, $b_1 \Rightarrow b_3$ implies $b_3 \Rightarrow b_2$. Of the three continuations of *Initialise*, only *Analysis* and *Terminate* are maximal continuations because they are also continuations of *Register*. The maximal continuations of all bMSCs for component *Control* are shown in Figure 5.

Third, the right hand side of each FSP production is appended with a string of the form `silentAction->b1 | … | silentAction->bn` where `b1…bn` are the names of bMSC that are maximal continuations of `b`.

Finally, productions with unreachable left hand side non-terminals are eliminated and the remaining FSP productions are printed according to FSP syntax. The action `silentAction` is hidden in the final FSP process because it represents an internal component action that is not visible to other components. In addition, the process is declared `determinisitic` using the corresponding FSP keyword. The final FSP process for the *Control* component is shown in Figure 4.

The result of the synthesis algorithm can be fed into LTSA and the LTS model for the *Control* component can visualised (see Figure 6). Once all components have been synthesised, the complete system is the parallel composition of all components: `||System = (Control || Sensor || Database || Actuator)`.

Definition 16. (FSP Synthesis) Let *Spec* be a MSC specification and let $P = (P_1 \parallel P_2 \parallel \ldots \parallel P_n)$ where P_i corrsponds to the LTS resulting from the algorithm of Figure 2. We say that P is the synthesised model of *Spec*.

A simple argument can be used to show that the model synthesised by our algorithm is an implementation of the MSC specifications used as input. First, we can prove that each synthesised component can produce all projections of words in $L(Spec)$ on its alphabet. Second, we prove that this is enough to guarantee that the composed system can produce all words in $L(Spec)$.

```
Init: Initialise
Initialise: Analysis, Terminate.
Register: Analysis, Terminate.
Analysis: Analysis, Terminate.
Terminate: Initialise.
```

Figure 5 – Maximal continuations for *Control*.

```
deterministic Control = Initialise,
Initialise = (on -> (silentAction -> Analysis |
                     silentAction -> Terminate)),
Analysis = (query -> data -> command ->
                     (silentAction -> Analysis |
                     silentAction -> Terminate)),
Terminate = (off -> Initialise)\{silentAction}.
```

Figure 4 – FSP specification for *Control*

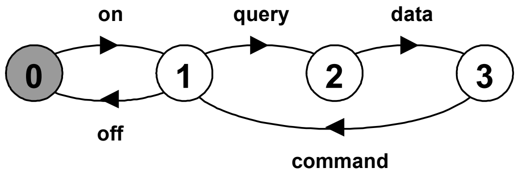


Figure 6 – LTS for *Control*

Proposition 1. Let $Spec = (B, H, f)$ be a MSC specification with instances I, and let $n_1, n_2, \ldots$ be a maximal path in H. If $w \in L(b_1 \bullet b_2 \bullet \ldots)$ where $b_j = f(n_j)$ then $w|_i = v_1 v_2 \ldots$ where v_j is the sequence of labels determined by the total ordering $<_i$ of b_j.

Demonstration: If $w = l_1, l_2, \ldots \in L(b)$ with $b = b_1 \bullet b_2 \bullet \ldots = (E, L, I, \lambda, <, tgt)$ and $b_k = (E_k, L_k, I_k, \lambda_k, <_k, tgt_k)$. From the definition of bMSC languages, there is a word $\sigma = s_1, s_2, \ldots$ such that $lbl(s_j) = l_j$ and $s_j \leq s_{j'}$ implies $j \leq j'$. From the definition of ≤ in sequential composition we know that as $<_k \subseteq \leq$, if $s_j, s_{j'} \in b_k$ and $j \leq j'$ then $s_j <_k s_{j'}$. Furthermore, if $s_{j,} \in b_k$ and $s_{j,} \in b_{k'}$ with $k < k'$, then $j < j'$.

Let us consider a new word σ' resulting from eliminating all events s in σ such that $lbl(s) \notin \alpha(i)$ with i an instance. Let $\sigma' = s_1, s_2, \ldots$ obtained by renumbering events in σ. It is clear that $w|_i = lbl(\sigma')$. Furthermore, if we replace all send events s not corresponding to instance i with their receive counterparts ($tgt(s)$) we will still have the following situation: $w|_i = lbl(\sigma')$, $s_j, s_{j'} \in b_k$ and $j \leq j'$ implies $s_j < s_{j'}$, and that if $s_{j,} \in b_k$ and $s_{j,} \in b_{k'}$ with $k < k'$, then $j < j'$. This is because for all send event, we know $lbl(s) = lbl(tgt(s))$, $s \leq tgt(s)$ and $tgt(s) \leq s$.

Consequently, it is possible to split σ' into words $\sigma_1, \sigma_2, \ldots$ where σ_i is a sequence of events of the instance i in bMSC b_k, and where σ_i is ordered according to $<_i$ in b_k. We also know that there are no missing events because all send events of the original maximal bMSC b must have appeared in σ and every event in instance i of bMSC b_k is either a send event or a reveive event of an event appearing in σ.

Proposition 2. Let *Spec* be a MSC specification with instances I. If $P = (P_1 \parallel P_2 \parallel \ldots \parallel P_n)$ is the synthesized model of *Spec* then $L(Spec)|_i \subseteq L(P)|_i$ for all $i \in I$.

Demonstration: We first prove to prove that $L(Spec)|_i \subseteq L(P_i)$ for all $i \in I$. Let P be the synthesised LTS of a MSC specification $Spec = (B, H, f)$ with instances I. Let $w \in L(Spec)$. We shall prove that $w|_i \in L(P_i)$:

As $w \in L(Spec)$, there is a maximal path $n_1, n_2, \ldots$ in H such that $w \in L(f(n_1) \bullet f(n_2) \bullet \ldots)$. Using the previous result, we then know that $w|_i = v_1 v_2 \ldots$ where v_j is the sequence of labels determined by the total ordering $<_i$ of $f(n_j)$.

We also know from the FSP synthesis algorithm that the FSP respresentation of P_i comprises productions of the form $f(n_j) =$

v_j.(*internalAction*->b_1 |...| *internalAction*->b_k) where . $b_1, ..., b_k$ are all the continuations of $f(n_j)$. Thus, as $f(n_{j+1})$ is a continuation of $f(n_j)$, $f(n_j)$ can produce sequence of labels v_j followed by *internalAction*and then continuing as production $f(n_{j+1})$. In addition as $f(n_1)$ is a continuation of Init, P_i must be of the form (...| *internalAction*-> $f(n_1)$ |...). Finally, as *internalAction* is hidden in process P_i it is clear that $v_1v_2... \in L(P_i)$. Thus, we have $w|_i \in L(P_i)$.

We now prove that $L(P_i) \subseteq L(Spec)|_i$: let $w \in L(P_i)$. We shall prove that there is a word $v \in L(Spec)$ such that $w = v|_i$. By construction of P_i we know that $w = v_1v_2...$ where there is a production of the form $f(n_j) = v_j.f(n_{j+1})$, $n_{j+1} \in H(n_j)$ and either w is infinite or $f(n_{|w|}) = \varnothing$. Let $v \in L(f(n_1) \bullet f(n_2) \bullet ...)$, we know that $v \in L(Spec)$ because $n_1, n_2,...$ is a maximal path in H. It is also clear that $w = v|_i$.

Proposition 3. Let *Spec* be a MSC specification with instances *I*, and *P* be an LTS resulting from the composition of P_i with $i \in I$. If $L(Spec)|_i \subseteq L(P)|_i$ for all $i \in I$ then $L(Spec) \subseteq L(P)$.

Demonstration: First we introduce the following notation: If (S, L, Δ, q) is an LTS, s and t are states in S, and w is a word over the alphabet L then $s \xrightarrow{w} t$ notes that there is a sequence $s_0, s_1, ...s_{|w|}$ such that $s_0 = s$, $s_{|w+1|} = t$, and $(s_{j-1}, w_j, s_j) \in \Delta$ for $0<j<|w|+1$ and w_j the j^{th} element in w. Thus, if w is an empty word $s=t$.

Let $P = (S, L, \Delta, q)$ be an LTS resulting from the composition of P_i with $i \in I$, such that $L(Spec)|_i \subseteq L(P_i)$ for all $i \in I$. We shall assume $w \in L(Spec)$ and prove that $w \in L(P)$. We shall assume that w is infinite, the proof for finite words follows the same reasoning.

The proof is by induction on the n for the following property: If $w \in L(Spec)$ then there are two words v, v' and a state q' such that $|v| = n$, $w=v.v'$, and $q \xrightarrow{v} q'$. It is clear that if the property is true for all n then $w \in L(P)$.

Supose $n = 0$, then v is an empty. If we choose q' to be q we trivially have that $q \xrightarrow{v} q$.

Suppose that the following holds: there are two words v, v' and an q' such that $|v| = n$, $w=v.l.v'$, and $q \xrightarrow{v} q'$. We wish to show that there is a state q'' such that $q \xrightarrow{v.l} q''$. By definition of parallel composition we can assume that $q = (q_1 \,||\, ... \,||\, q_n)$, $q' = (q_1' \,||\, ... \,||\, q_n')$ and that $q_i \xrightarrow{v|_i} q_i'$. We also know that $q_i \xrightarrow{v|_i.l} q_i''$ for all $i \in I$ such that $l \in \alpha(i)$. We now have $q_i \xrightarrow{v|_i} q_i' \xrightarrow{l} q_i''$. Thus we have that $q' \xrightarrow{l} q''$ and finally that $q \xrightarrow{v.l} q''$.

Corollary 1. If *P* is the synthesised model of a MSC specification *Spec*, then *P* is an implementation of *Spec*.

4. IMPLIED SCENARIOS

In the previous section we defined a rather weak notion of implementation. We only require model to include all possible behaviours that have been described in the MSC specification ($L(Spec) \subseteq L(P)$). In many cases any implementation will not do. One wishes to obtain an implementation that is as close as possible to the language of the specification. Moreover, why not have an implementation that provides exactly the same language as the specification? The problem is that such implementation does not always exist. In this section we exemplify how this happens and define the notion of implied scenario. In the next section we show how these situations can be detected.

Applying the synthesis algorithm presented above, we can construct the complete FSP model for the MSC specification of Figure 1. Once the FSP specification is fed into the LTSA tool, we can play with our model to see how it behaves. In Figure 7 we show a trace (using the bMSC syntax for clarity) which can be executed in our system model. The trace shows how the *Control* component is accessing the database and receiving information from a previous activation of the sensor. This is clearly not an intended behaviour and does not belong to the language of our MSC specification. The MSC specification clearly states that after initialising *Sensor* there must be some data registered into the *Database* before any queries can be done. Note that as the pressure message and the query message involve the database, the trace of Figure 7 cannot be a result of the interleaving of messages in some maximal bMSC determined by the high-level MSC diagram of Figure 1.

This means that we have a problem with our implementation. It is allowing some system executions that are unacceptable. We could try to build another implementation that does not include this trace. However, we can show that the implementations synthesised by the algorithm presented above are the implementations that allow the least system executions that are not in the language defined by a MSC specification. In other words our synthesised model of the system specified in Figure 1 is minimal with respect to inclusion of system traces.

Proposition 4. If *P* is the synthesised LTS of a MSC specification *Spec*, then *P* is the minimal implementation of *Spec* (i.e. for all implementation P', $L(P) \subseteq L(P')$).

Demonstration: Let *Spec* be a MSC specification with instances *I*, let $P = (P_1 \,||\, ... \,||\, P_{|I|})$ be the synthesised LTS of *Spec*, and $P' = (P'_1 \,||\, ... \,||\, P'_{|I|})$ an implementation of *Spec*.

We have already shown that $L(Spec)|_i = L(P_i)$ for all $i \in I$. In addition it is easy to show that $L(Spec) \subseteq L(P')$ implies $L(Spec)|_i \subseteq L(P'_i)$. Thus we have $L(P_i) \subseteq L(P'_i)$ for all $i \in I$. It follows that $L(P) \subseteq L(P')$.

We must now conclude that the unwanted trace will appear in any implementation of our system. How can this be possible? If we

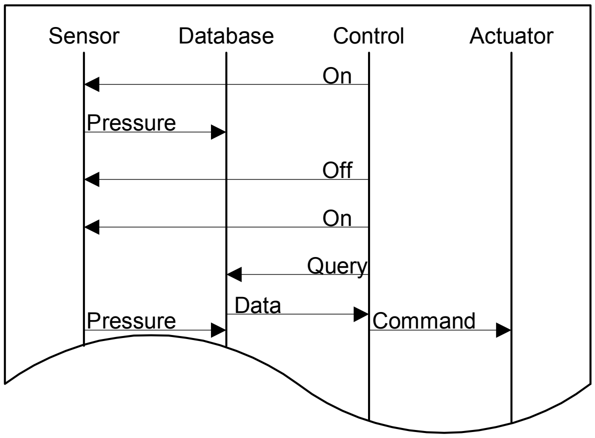


Figure 7 – Implied Scenario

analyse the MSC specification closely, we can see the following: 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 between a first registration of data from others. Thus, it cannot enable and disable queries appropriately. Succinctly, components do not have enough local information to prevent the system execution shown in Figure 7. 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*, *Register*. However, the *Database* is doing *Initialise*, *Register*, *Analysis*, *Register*. We will use the term implied scenario to refer to system executions like the one shown in Figure 7.

Definition 17. (Implied Scenario) Let *Spec* be a MSC specification with an alphabet *L*. An implied behaviour is a word $w \notin L(Spec)$ over the alphabet *L* such that $w \in L(P)$ for all implementations of *Spec*.

Implied scenarios are not necessarily unwanted situations they can simply be acceptable scenarios that have been overlooked by stakeholders. Thus, once implied scenarios have been detected (discussed in the next section), the MSC specification can be completed with the overlooked scenarios and refined to avoid unwanted ones. Eventually an MSC specification that has not implied scenarios may be reached. Thus we can use our synthesis algorithm to build an implementation that behaves exactly as the specified system. We can guarantee this as the algorithm builds minimal implementations. We say that an implementation is safe if it behaves exactly as the specified system.

Definition 18. (Safe Implementation) Let *Spec* be a MSC specification, and *P* an implementation. *P* is a safe implementation of *Spec* if *L(Spec) = L(P)*. We shall say that a *Spec* is safely implementable if there is a safe implementation of *Spec*.

The following corallary follows from the fact that synthesised models are minimal:

Corollary 2. If *P* is the synthesised model of a safely implementable MSC specification *Spec*, then *P* is safe implementation of *Spec.*

5. DETECTING IMPLIED SCENARIOS

We have shown how a minimal implementation can be constructed for a MSC specification. But we have also shown that it is possible to obtain unexpected behaviours from such implementations. These implied scenarios can help complete the MSC specification with unforeseen situations or indicate that the specification must be refined to prevent unwanted executions. Consequently, detecting implied scenarios is a relevant issue.

Having developed an algorithm that builds a system model within an analysis tool such as LTSA, we have focused on using such a tool to detect implied scenarios. The simplest approach would be to build a safety property that has exactly the same behaviours as the MSC specification and check that our LTS model satisfies such property using LTSA. If the property is satisfied, then the model cannot perform any more executions than that of the property. As the property behaves exactly as the specified system, we could conclude that the LTS model does not have implied behaviours.

However, this naïve approach would be extremely expensive in computational terms. This is due to the fact that the language of an MSC specification cannot be built compositionally. Concatenating the languages of bMSCs according to the hMSC does not provide the specified behaviour, nor does combining partial orders for each bMSC. The way to build the language of a MSC specification would be to construct all maximal bMSCs, and find all linearisations for each one. However, the number of maximal bMSCs may be infinite and so may be the length of each one.

We avoid calculating the whole language of an MSC specification by finding a safety property that accepts a simpler language, which if satisfied by the LTS model guarantees the non-existence of implied scenarios. Furthermore, if the property is not satisfied, the counter-example provided by LTSA is an example of an implied scenario. The problem, of course, is to find such a simpler language to use as our safety property.

The general reasoning we use is the following: If there is an implied scenario then the model can behave properly according to the specification up to a certain point and then deviate from acceptable behaviour. This deviation must be able to occur in a finite prefix of some trace. Furthermore, this deviation must be able to occur before the system reaches a state in the LTS system model twice. Accordingly, we build a safety property that accepts traces that behave correctly according to the MSC specification up to a point where a system state has been reached twice.

Building such a property requires listing all acceptable behaviours of the MSC specification and truncating them when a loop in the LTS model is detected. This is not a simple task because there are infinite number acceptable behaviours, and because the cost of simulating each one on the LTS model to check when a state is reached twice is too big. The method we now present builds a safety property without using the LTS model.

We assume that the MSC specification has been normalised to

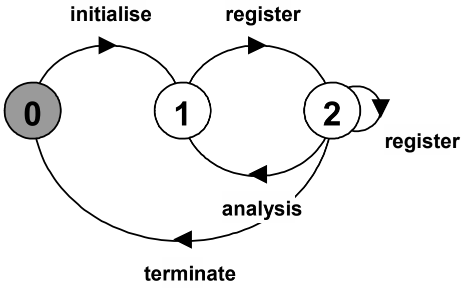


Figure 8 – hMSC viewed as an LTS.

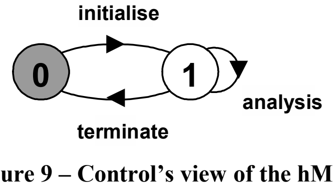


Figure 9 – Control's view of the hMSC.

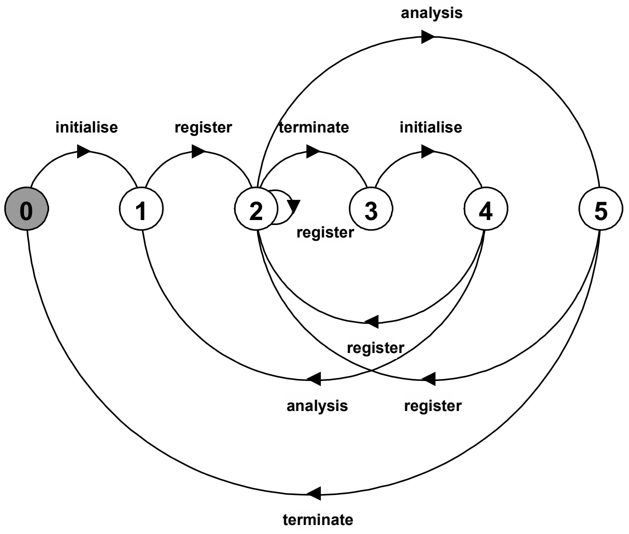


Figure 10 – Composition of component hMSC views.

avoid choices of bMSCs in the hMSC that can start with a common message. Normalisation can be automated by applying the method described by Hélouet and Le Maigat [5].

In the rest of the section shall use an alternative representation of hMSCs. We shall view hMSCs as labelled transitions systems instead of graphs. The transitions of the LTS are labelled with bMSC names and the language accepted by the LTS is the set maximal bMSCs of the MSC specification. Synthesising such LTS from an hMSC is very simple; in Figure 8 we show the LTS view of the hMSC of Figure 1 as an example. In the remainder of this section hMSC will refer to the LTS representation of hMSCs

From a components perspective, the hMSC may refer to bMSCs in which it does not participate. For example, the *Control* component only participates in bMSCs *Initialise*, *Analysis* and *Terminate*. Thus, the occurrence of bMSC *Register* is completely transparent to it. We define a component view of an hMSC as an LTS: We require the LTS to accept all projections of the words accepted by the hMSC on the alphabet of bMSCs in which the component participates. In Figure 9 we show the *Control* hMSC view. Note that a component hMSC view shows the relation between the synthesised FSP productions of the component (Compare Figure 9 with Figure 4). Furthermore, as the component LTS is deterministic, we can show that the component's view of the hMSC is an abstraction of its behaviour: Each state in the component hMSC view represents a state in the component LTS. The state reached after accepting the events determined by a sequence of bMSCs in the component LTS is the state represented by the one reached after accepting the same sequence of bMSCs in the component hMSC view. For example state 1 in Figure 9 represents the component state 1 shown in Figure 6. This means that after executing the events determined by a sequence of bMSCs that leads to state 1 in Figure 9, the component will always reach state 1 of Figure 6.

The fact that component hMSC views are abstractions of their own behaviour is important because we can use it to build an abstraction of the system LTS. If we compose component hMSC views in parallel we obtain an abstraction of the system model that assumes that components will synchronise in their choices of bMSCs. (see Figure 10). Components do not actually synchronise on bMSCs, they synchronise via messages. A component could choose to go through a different bMSC than the rest of the components if the events involved are the same. Nevertheless, because components are deterministic, the state in which the component would be in after choosing either bMSC is the same. So it is as if the component chose the right bMSC and synchronised with the rest of the system components.

Following this reasoning, we can show the state reached by the system model after accepting the events determined by a sequence of bMSCs is the state represented by the one reached after accepting the same sequence of bMSCs in the composed hMSC view. For example, the state reached by the implied scenario shown in Figure 7 is being represented by state 2 in Figure 10. Returning to the explanation of why the implied scenario of Section 4 occurred, we mentioned that the *Sensor*, *Control* and *Actuator* were going through scenarios *Initialise*, *Register*, *Terminate*, *Initialise*, *Analysis*, *Register*. While the *Database* was doing *Initialise*, *Register*, *Analysis*, *Register*. Note that both sequences of bMSCs lead to the same state in Figure 10.

We now have an abstraction of our system model that allows us to detect when the synthesised implementation model of a MSC specification has looped. We will build a safety property that accepts traces that behave correctly (according to the MSC specification) and do not go more than once through a state represented by the composed hMSC. If there is an implied scenario in which deviated behaviour occurs after passing one of these states more than once, the system must have been able to perform this deviated behaviour in its first pass through the state. Moreover, as we have assumed a normalised hMSC, the behaviour without the loop must also be an implied scenario. Thus, our safety property would have detected it. .

Definition 19. (Safety Language) Let *Spec = (B, H, f)* be a MSC specification with a set of labels *L*. The safety language of *Spec*, denoted *SL(Spec)* is a set of words over the alphabet *L* such that $w \in SL(Spec)$ if and only if there is a prefix $s_1, s_2, \ldots, s_n$ of *w* such that $s_1, s_2, \ldots, s_n, \ldots$ is a linearisation of a maximal bMSC $b_1, b_2, \ldots, b_m, \ldots$with $b_1, b_2, \ldots, b_m$ a maximal non-looping sequence of the composed hMSC view and either s_n is the first event appearing in *w* of bMSCs $b_{m+1}, b_{m+2}, \ldots$ or $w = s_1, s_2, \ldots, s_n$

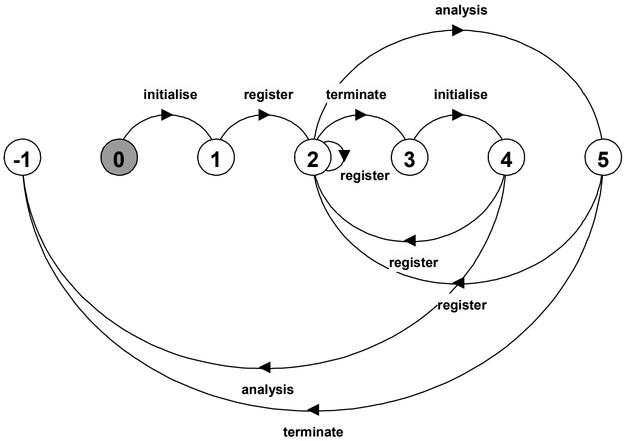


Figure 11 – Composed hMSC with violations to hMSC.

The construction of our safety property is quite straightforward. *First* all maximal non-looping sequences of bMSCs in the composed hMSC view that are valid according to the hMSC are constructed. We do this by considering the hMSC as a property and checking the traces of the composed hMSC that violate this property. This is shown in Figure 11, where traces in the composed hMSC that are not valid according to the hMSC lead to state –1. An example for the composed hMSC of Figure 10 is the sequence *Initialise*, *Register*, *Terminate*, *Initialise*, *Register*.

Second, for each sequence of bMSCs, we calculate the set of possible first messages that can occur starting at the ending state of the sequence. These messages will be an indication that an acceptable behaviour can be truncated and we call them truncating messages of the bMSC sequence. As an example, we look at sequence shown above. This sequence ends in state 2, which is the first state to be reached twice. We are interested in the first messages that can occur starting at state 2. These can be found by looking at the partial orders determined by valid bMSC sequences starting at state 2. We know that there is no need to look for messages after a bMSC is repeated once so this is relatively cheap. The messages that can occur first starting at state 2 are: *Pressure*, *Off* and *Query*.

Third, each sequence is composed sequentially together each one of its truncating messages. For the sequence *Initialise*, *Register*, *Terminate*, *Initialise*, *Register* three bMSC would be constructed, one for *Pressure*, another for *Off* and a third for *Query*. The first bMSC would be the result of sequentially composing (*Initialise* • *Register* • *Terminate* • *Initialise* • *Register*) with a bMSC that only has the message *Pressure*.

Finally, the safety property can is built by simply enumerating the linearisations of all constructed bMSCs. These linearisations are truncated as soon as the truncating message is reached.

Once the safety property for detecting implied scenarios is built, the synthesised implementation can be checked for implied scenarios using LTSA. The result is a trace that leads to the violation of the safety property and that is (a prefix of) an implied scenario of the Sensor system MSC specification. The trace returned by LTSA is shown in is shown in Figure 12 and corresponds to the implied scenario depicted in Figure 7.

Summarising the results presented in this section, we have presented a method for building a safety property, which combined with the synthesised implementation presented before can check if the implementation is safe and if not provide an example of implied scenario.

Theorem 3. Let *P* be the synthesised LTS of the normalised MSC specification *Spec* and *SP* be the safety property that accepts *SL(Spec)*,

A. If *P* satisfies S*P* then *P* is a safe implementation of *Spec*.

B. If *P* does not satisfy *SP* and *w* is a trace violating *SP* then *Spec* is not safely implementable and *w* is the prefix of an implied scenario of *Spec*.

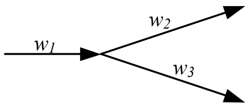
```
Trace to property violation in DetProperty:
        start
        pressure
        stop
        start
        query
```

Figure 12 – Implied Scenario detected by LTSA.

Demonstration (Part A): We shall prove the contra-positive of *P* satisfies *SP* then *L(P) = L(Spec)*. However, as $L(Spec) \subseteq L(P)$ we shall prove that if $(L(P) \setminus L(Spec)) \neq \varnothing$ then *P* does not satisfy *SP*.

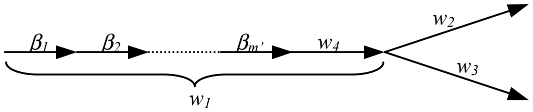
Let *Spec* be a MSC specification, *P* be the synthesised LTS of *Spec* and *w* such that $w \in (L(P) \setminus L(Spec))$ we shall show that there is a word in $L(P) \setminus L(SP)$.

We can split *w* just where it can no longer be extended into a word in *L(Spec)*. Let $w = w_1w_2$ such that w_1 is a maximal prefix of *w* that can be extended with by some w_3 such that $w_1w_3 \in L(Spec)$.



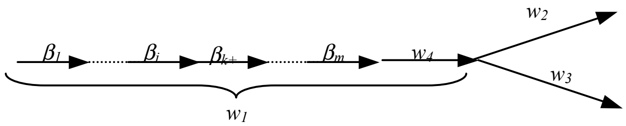
As $w_1w_3 \in L(Spec)$, there is a sequence $s_1, s_2, ..., s_n, ...$ which is a linearisation of a maximal bMSC $b_1, b_2, ..., b_m, ...$ such that $w_1 = lbl(s_1, s_2, ..., s_n)$ and for all $0 < I \leq n$ there is $j \leq m$ such that $s_i \in b_j$. We also know that we can reorder all s_i so to have events of the same bMSC together and still have $w_1w_2 \in (L(P) \setminus L(Spec))$ with w_1 maximal prefix of *w* that can be extended with by some w_3 such that $w_1w_3 \in L(Spec)$.

Suppose we have an event from a bMSC $b_1, b_2, ..., b_m$ that does not appear in $s_1, s_2, ..., s_n$. Then it must appear in the rest of the sequence of events s_i. If this event is independent from the first event of w_2 then we must be able to move it from w_3 to w_1. So we can reach the following situation, where β_i is a word over the events of b_i for $i \leq m' \leq m$.



Suppose we have an event in w_4 that is independent of the first event in w_2, we can move it to the beginning of w_3, while still having $w_1w_2 \in (L(P) \setminus L(Spec))$ with w_1 maximal prefix of *w* that can be extended with by some w_3 such that $w_1w_3 \in L(Spec)$. But in addition, we now have that none of the $b_{m'+1}, ..., b_m$ are complete whithin w_1.

Furthermore, we can show that if we have $b_1, ..., b_j, ...b_k, ..., b_{m'}$ and $b_1, ..., b_j$ leads to the same state as $b_1, ..., b_k$ in the composed hMSC then we can eliminate $b_{j+1}, ..., b_k$ from w_1 and still have $w_1w_2 \in (L(P) \setminus L(Spec))$ with w_1 maximal prefix of *w* that can be extended with by some w_3 such that $w_1w_3 \in L(Spec)$. This is because the original MSC specification was assumed to be normalised.



This shows that if there is an implied scenario, there is also one which will the deviate before fully completing the bMSCs that appear in a non-looping trace of the composed hMSC

This implied scenario will not be accepted by our safety property. We shall show this by assuming the contrary and reaching a contradiction: If the implied scenario were accepted by our safety property then there would be a prefix $s_1, s_2, ..., s_n$ of w_1w_2 such that there is a linearisation $s_1, s_2, ..., s_n, ...$ of a maximal bMSC $b_1, b_2, ..., b_{n'}, ...$with $b_1, b_2, ..., b_{n'}$ a maximal non-looping sequence of the composed hMSC and s_n is the first event appearing in w of bMSCs $b_{n'+1}, b_{n'+2}, ...$or $s_1, s_2, ..., s_n = w_1w_2$

- If w_1 is a proper prefix of $s_1, s_2, ..., s_n$ then $s_1, s_2, ..., s_n$ includes the first action in w_2. But as $s_1, s_2, ..., s_n$ is the prefix of a word in *L(Spec)* then w_1 is not the maximal prefix of w_1w_2 that can be extended to belong to *L(Spec)*. This is a contradiction.

- If w_1 is not a proper prefix of $s_1, s_2, ..., s_n$, then $s_1, s_2, ..., s_n$ is a prefix of w_1. We have two possibilities:

  - o If $s_1, s_2, ..., s_n$ is a prefix of $\beta_1, \beta_2, ...\beta_m$, as $b_1, b_2, ...b_m$ does not have loops and s_n is the first event in the looping part we have a contradiction.

  - o If $\beta_1, \beta_2, ...\beta_m$ is a proper prefix of $s_1, s_2, ..., s_n$ then s_n is in w_4. As s_n is in one of the following $b_{n'+1}, b_{n'+2}, ...,$ all the events of $b_1, b_2, ...b_{n'}$ that are missing in w_4 are independent of s_n. Then we could move these missing events before s_n and still reach the same global state after s_n. But now we would have a sequence of $\beta_1, \beta_2, ...\beta_{n'}$ that loops and we had repeated this procedure as far as possible to generate w_1w_2. This is a contradiction.

Demonstration (Part B): Let *Spec* be a MSC specification, *P* be the synthesised LTS of *Spec* and w a trace detected by our safety property. We shall show that w is an implied scenario. In other words that $w \not\in L(Spec)$. We shall suppose that $w \in L(Spec)$ and reach a contradiction.

First of all, we know that if w is detected by our safety property, then there is no prefix $s_1, s_2, ..., s_n$ of w such that $s_1, s_2, ..., s_n, ...$ is a linearisation of a maximal bMSC $b_1, b_2, ..., b_m, ...$with $b_1, b_2, ..., b_m$ a maximal non-looping sequence of the composed hMSC and s_n is the first event appearing in w of bMSCs $b_{m+1}, b_{m+2}, ...$or $w = s_1, s_2, ..., s_n$

However, if $w \in L(Spec)$ then there is a maximal bMSC $b_1, b_2, ...$ such that w is a linearisation of it. Let $b_1, b_2, ..., b_m,$ be the maximal non-looping sequence of bMSCs. If we consider the minimal prefix of w in that ends with an event in $b_{m+1}, ...$ this prefix contradicts the previous paragraph. If such prefix does not exist, then b_m is the last bMSC of the bMSC sequence, thus it also contradicts the previous paragraph.

6. LTSA – MSC TOOL

The algorithms presented above have been implemented in Java and integrated into the Labelled Transition System Analyser (LTSA) tool. MSC specifications are inputted in textual format [7] and output is a FSP specification, which can immediately processed by LTSA. The implementation, together with some examples (including the one used throughout this paper), is available at [11]. In Table 1 we show some execution times and sizes of synthesised LTSs for the example used in this paper, a slightly bigger version of it and a version of the ATM system (see e.g. [10]). All examples were run on a Pentium III, 300Mhz, 256Mb with Windows NT 4.0 and Java 1.3.

7. RELATED WORK

This work uses several of the concepts presented by Alur et al. in [1]. In particular we have used their notions of implied scenario and realisability, which we call implementability. The fundamental difference with this work is the scenario language being studied. In [1] only bMSCs are allowed, thus the issue of constructing an implementation and finding implied scenarios is limited to a finite set of finite executions. We extend their work as high-level MSCs allow specifying an infinite number of (possibly infinite) systems traces. Another difference is that in [1] communication between components is considered to be asynchronous. In other words message passing is not considered to be hand shaking.

Van Lamsweerde et al. [13] present a different approach, a set of examples and counterexamples expressed as scenarios is used to infer a temporal logic specification. Thus, generating explicit declarative requirements from an operational description. Combining these requirements with LTS models may be an interesting possibility for future work.

Harel et al. [4] use a complex scenario language that uses live sequence charts (LSC) to describe universal and existential scenarios. This approach departs from the idea of using simple graphical scenario languages for requirement elicitation. We believe that much benefit can and should be gained from the kind of scenario languages being used today, thus prefer the simpler approach to scenarios. The synthesis method presented in [4] differs from our approach significantly in that firstly a system model is constructed and then decomposed into a set of components.

There is much work on scenario on synthesis techniques for building models from a scenario description. However, 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

Table 1 - Synthesis algorithm execution times.

	Nodes in hMSC	Transitions in hMSC	Model synthesis time	Model size (# states)	Property synthesis time	Property size (# states)	Safety check time
Sensor v1.0 (this paper)	4	7	20ms	11	190ms	17	31ms
Sensor v2.0	6	8	40ms	21	200ms	42	92ms
ATM	10	15	90ms	32	211ms	71	82ms

uniquely determines an implementation up to a certain level of abstraction. This is a valid approach but differs from the one used in this paper where we consider that a specification can be implemented by many models. In this view of scenarios as a design language many approaches provide algorithms for generating statechart models from MSCs [2, 10, 14]. Another approach is to provide a formal semantics for MSCs based on state machines such as the one provided by Cobens et al. [3], which is part of the Z.120 recommendations for MSCs. In [12] we have present an MSC language that integrates these kind of approaches by providing a simple mechanism for tailoring MSC specifications to specific interpretations by use of state labels.

8. CONCLUSIONS

We have presented a framework for synthesising implementation models for scenario-based specifications and for the existence and providing feedback on implied behaviours. This framework, which has been entirely implemented and integrated in the LTSA tool, allows building a model of a system that implements a MSC specification. The resulting model is guaranteed to be the model that implements the least unwanted behaviours. The framework also provides a method for assessing if a scenario specification has implied behaviours. Furthermore, an example of implied behaviour is given if the specification is not safely implementable. Finally as our approach integrates with LTSA, the synthesised implementation can be more thoroughly analysed by model checking of safety and liveness properties. There is also the potential for model animation [9] as a means of including further domain constraints and of making the models more comprehensible to stakeholders and developers.

An important observation on implied scenarios is that they are the result of an inconsistency between system decomposition and system behaviour. Providing an implementation for a set of components that can send and receive messages as in the MSC specification such that the overall system behaviour is the language determined by the MSC specification is not always possible. Implied scenarios are not an artefact of a particular MSC language, they are the result of specifying the global behaviours of a system that will be implemented component-wise.

Scenarios have proved to be a good tool for bridging the gap between stakeholders and developers. However, up to now, this is mainly a one-way bridge in which developers gain more insight of stakeholders' domain knowledge. Future work will be focused on building a bridge in the other direction, i.e. building mechanisms to provide feedback of the developer's world to stakeholders. Preliminary work in this direction is promising. We are automating the construction of alternative system views from synthesised LTS models.

9. ACKNOWLEDGMENTS

We would like to thank Victor Braberman for his helpful comments and discussion. We are grateful to the EPSRC for funding part of the work under grant BR/M24493 (BEADS).

10. REFERENCES

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

2. Broy, M., Krüger, I., Grosu, R. and Scholz, P., From MSCs to Statecharts. in *Distributed and Parallel Embedded Systems*, (, 1999), Kluwer Academic Publishers, 61-71.

3. Cobens, J.M.H., Engels, A., Mauw, S. and Reniers, M.A. Formal Semantics of Message Sequence Charts, Eindenhoven University of Technology, Eindhoven, The Netherlands, 1998.

4. Harel, D. and Damm, W., LSCs: Breathing Life into Message Sequence Charts. in *3rd IFIP Int. Cond. of Formal Methods for Open Object-Based Distributed Systems*, (New York, 1999), Kluwer Academic, 293-312.

5. Helouet, L. and LeMaigat, P., Decomposition of Message Sequence Charts. in *2nd Workshop on SDL and MSC*, (Grenoble, France, 2000).

6. Holzmann, G.J. and Peled, D., The State of Spin. in *CAV'96*, (, 1996), Springer.

7. ITU. ITU-T Recommendation Z.120. Message Sequence Charts (MSC'96), ITU Telecommunication Standardisation Sector, Geneva, 1996.

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

9. Magee, J., Kramer, J., Giannakopoulou, D. and Pryce, N., Graphical Animation of Behavior Models. in *22nd International Conference on Software Engineering (ICSE'00)*, (Limerick, Ireland, 2000), 499-508.

10. Systä, T. Static and Dynamic Reverse Engineering Techniques for Java Software Systems *Dept. of Computer and Information Sciences*, University of Tampere, Tampere, 2000.

11. Uchitel, S. LTSA-MSC Tool., Available at http://www-dse.doc.ic.ac.uk/~su2/Synthesis/, Department of Computing, Imperial College., 2001.

12. Uchitel, S. and Kramer, J., A Workbench for Synthesising Behaviour Models from Scenarios. in *ICSE 2001*, (Toronto, Canada, 2001).

13. Van Lamsweerde, A. and Willemet, L. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering*, *24* (12). 1089-1114.

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