Phase Semantics of MSC Traces

Paul Bristow, Clive Jervis, Bill Mitchell, Robert Thomson,
b.mitchell@motorola.com
Motorola UK Research Lab

March 28, 2003

Abstract

Specifications for wireless telecommunications systems are often only partially defined. It is also common for the specification to consist of a set of normative scenarios together with scenarios for some of the more important exceptional behaviours. A major challenge is to find effective means of detecting feature interaction conflicts contained in such specifications. Moreover the detected conflicts should be of value in debugging the specifications and should not be cluttered with inconsequential errors that are due to the incompleteness of the specifications.

The paper describes a technique for constructing a phase automaton that can be used to statically analyse the specification scenarios in order to detect certain types of interactions between them, known as phase transition conflicts. There is anecdotal evidence to suggest that these kinds of inconsistency can account for a significant proportion of feature interactions.

The phase automaton is intended primarily for the purpose of detecting these phase transition errors. This means the state space in the automaton only need include that part of the feature behaviour that is significant for those conflicts. This results in a small automaton that tends to make the conflict detection problem tractable.

1 Introduction

The pressure to bring wireless telecommunication products to market rapidly results in very lightweight requirements specifications, which are frequenctly imcomplete and often consist mainly of a collection of 'sunny day' scenarios. For convenience we will assume such scenarios are defined as MSCs [**16**], however the results apply to any notation that defines the externally observable concurrent traces of the system components.

These scenarios tend to concentrate on the behavior of individual features or components, and do not consider how the various features will interact in any great detail. This naturally leads to conflicts when the various features are allowed to act concurrently. It is also possible for scenarios for the same feature to contain inconsistencies with respect to each other. The main problems addressed in this paper are how to analyse partial specifications

- to give useful conflict reports for debugging the specification
- that avoids generating many inconsequential errors
- and applies to complex specifications

Engineering groups will normally only focus on key issues during requirements specification. A secondary issue is to define a technique that users can easily apply to their existing specifications and which requires no additional input from the users. Hence some means must be found to reason about the requirements specifications in the form that engineering group are comfortable with, and which reveal errors that are significant for those groups.

Motorola UK Research Lab has been collaborating with testing, architecture and design groups throughout Motorola to investigate this area over the last few years. We have found that standard analysis techniques are inappropriate for scenario based incomplete specifications. For example, an internal study of TETRA [**18**] MSC [**16**] engineering scenarios used Spin and algorithms similar to those found in [**1**], [**2**] and [**12**] to analyse the specifications in a standard manner. The final model suffered from the usual state explosion problem and generated many error reports none of which were genuine. Two other Motorola Research Labs used independent methods to construct a complete model of a POTS system incorporating a switch, written in SDL [**17**], which were analysed with technology built out of the FDR model checking system. It also suffered state explosion problems, and was only able to find conflicts when the exploration algorithm was directed towards the error states. Hence we have found it necessary to develop novel approaches to the problem that are reported here.

The paper assumes that a protocol specifies message exchanges whose purpose is to define the transitions between different phases of operation in the system. For example in a system such as TETRA [**18**] there are phases such as *call setup*, *call active*, *call roaming*, *call queued*, and *ruthless resource pre-emption*. The specifications for a TETRA system define transitions such as from *call setup* to *call active*. Some of these transitions may involve other phases. For example, the transition from *call setup* to *call active* may include intermediate transitions to *call queued* or *ruthless resource pre-emption*. By establishing how the phases of operation used in the specification restrict potential system implementations it is possible to analyse some of the concurrent behaviour of the features. This is behaviour not explicitly defined by the original specification, but is a consequence of the phase semantics of the combined features. Within this set of concurrent behaviours it is possible to search for feature conflicts.

Given a scenario defining message exchanges that causes a phase transition, which may include other intermediate phase transitions, we can study the event traces defined by the scenario. Each event in a trace can be annotated by the phase that was active when that event occurred. These define the phase traces of a scenario (see Section 2 for the formal definition for MSC phase traces).

The paper defines an abstract implementation of the phase traces for each process in the specification. This implementation is in the form of an automaton, called the phase automaton. This describes the implicit phase traces defined by the specification. These will often include phase traces describing implicit phase transitions of the specification. Static analysis of this automaton can detect errors and ambiguities in the specification with respect to the phase transitions. Two main types of such error are described here, these are called phase transition errors. They are defined in Section 5.

An internal software tool, FATCAT, has been developed by Motorola UK Research Labs that constructs phase automaton implementations of each instance in a set of MSCs, analyses the automata for phase transition errors and then outputs MSC conflict scenarios. FATCAT is built on patented technology ([**4**], [**5**]) already developed by Motorola UK Labs for the *ptk* software tool [**3**]. The *ptk* tool generates a set of conformance tests from a collection of MSCs. *ptk* has been developed over a number

of years and is now used by engineering groups throughout Motorola.

The problem of composing separate MSCs into a single implementation has been studied for some time. The work reported in [**1**], [**2**] and [**12**] describes how to synthesize automata from a set of MSCs and consider related decidability issues. That work does not consider phase traces, which is the central focus here, instead they look at the MSC event traces. The difficulties considered in [**1**], [**2**] and [**12**] arise since race conditions may occur in the MSCs. Asynchronous wireless telecommunication protocols are intended to be race free and when that is the case it is straightforward to construct automata describing their event traces. Mauw ([**13**], [**14**]) considers a form of interleaving to compose MSCs. This approach also works with event traces rather than with phase traces. The idea of static analysis of specifications through overlapping techniques has also been studied by Calder and Miller [**6**]. They successfully consider how to statically examine Promela encodings of state machines as one technique for avoiding the state explosion problem.

2 Phase Traces and Phase Automaton

This section defines phase traces and phase automaton. Later sections will define a semantics for a set of phase traces in terms of a phase automaton. This will allow us to define a model of a set of MSC specifications in the form of a phase automaton.

Throughout this document we will adopt the following notation. Where a message m occurs in an MSC, we use $!m$ to denote the send event generated by m, and $?m$ to denote the receive event generated by m in accordance with MSC semantics [**16**].

Let the set of events that can occur in the specifications be taken from the set of symbols E, let P be the set of phases that can occur in the specifications. Let S be a set of states, and let ϕ be a function $\phi : S \longrightarrow P$. Define a set of deterministic transitions to be a partial function $\partial : S \times E \longrightarrow S$. The tuple $\mathcal{A} = (S, E, \partial, \phi)$ is defined to be a *phase automaton*. Note there is no start state or accepting states defined for $\mathcal{A}$. These will not be required for the type of conflict analysis discussed in the paper, which are defined in terms of phase transitions. It is also often the case in practise that such states are only defined after the fact, so making assumptions about them during requirements analysis is not appropriate.

A *phase trace* is a sequence of the form

$$S_0,\, a_0,\, S_1,\, a_1,\, \ldots S_n,\, a_n,\, S_{n+1}$$

where each $a_i \in E$ and each $S_i \in P$.

A phase trace is a *specification phase trace* if the sequence

$$a_0,\, a_1,\, \ldots a_n$$

is a complete trace of events from one of the specification scenarios, and in that scenario each event a_i occurs during phase S_i, and after the event the phase becomes S_{i+1}.

A phase trace is an *execution phase trace* of phase automaton $\mathcal{A}$ if there are states x_i for $0 \leq i \leq n+1$ such that $\phi(x_i) = S_i$ and $\partial(x_i, a_i) = x_{i+1}$. When a specification phase trace t is also an execution phase trace, we say the phase automaton generates t.

From now on we will refer to both specification phase traces and execution phase traces as simply phase traces whenever it is clear from the context which prefix should apply.

Phases are defined in an MSC with conditions. For example consider the MSC in figure 1, which has conditions $S0$ and $S1$. The phase traces for an instance in an MSC are the specification phase traces that are defined for events that occur on that instance.

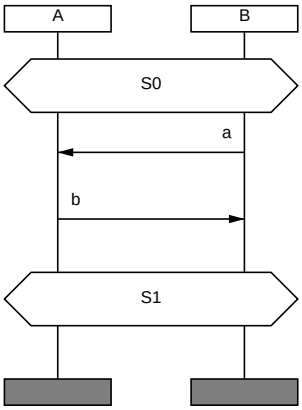

Figure 1: Partial Specification MSC

There is a single phase trace for instance A in Figure 1:

$$S0,\ ?a,\ S0,\ !b,\ S1$$

That is the current phase immediately before event $?a$ is $S0$, which is also the current phase immediately after $?a$ and before $!b$. Once $!b$ occurs the current phase becomes $S1$. Hence $!b$ causes a phase transition in instance A.

3 Phase semantics discussion

Adding phase information to scenarios provides extra information about the intended structure of the state space of an implementation. For a phase automaton to implement a set of phase traces it will not be enough for it to simply generate each phase trace in the set. It will also be necessary for the phase automaton to combine phase transitions contained in the phase traces in the correct way.

In an ideal world we could suppose that phases are nothing more than explicit state names in the implementation. This is unworkable if the specifications do not explicitly name each state change after every event that occurs on each instance of an MSC. It is not the purpose of protocol specifications to provide this minute level of implementation detail, and it is unlikely that any engineering group would accept such an overhead.

Common practise treats MSC conditions as a convenient mechanism for incorporating important composite states into the scenario specifications. Effectively this makes the phase traces the focus of the specification scenarios, rather than the event traces.

The problem now becomes one of how to combine a set of phase traces within a single phase automaton. Consider the two MSCs contained in Figure 2.

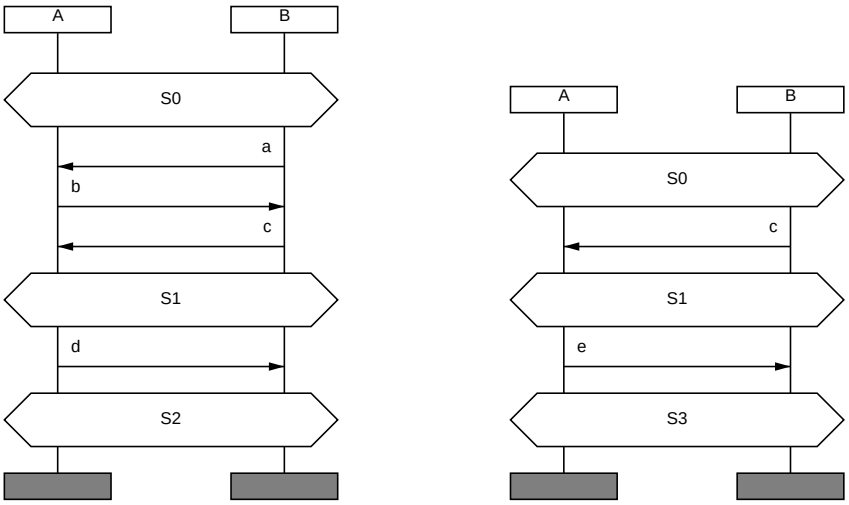

Figure 2: Two MSC scenarios for instances A and B

Instance A has a single phase trace in each MSC of Figure 2:

$$S0,\ ?a,\ S0,\ !b,\ S0,\ ?c,\ S1,\ !d,\ S2$$
$$S0,\ ?c,\ S1,\ !e,\ S3$$

How are these to be combined within a single phase automaton? If we have no further information about how the different phase traces are related to each other, the only way we could incorporate these phase traces within an implementation would be as shown in Figure 3.

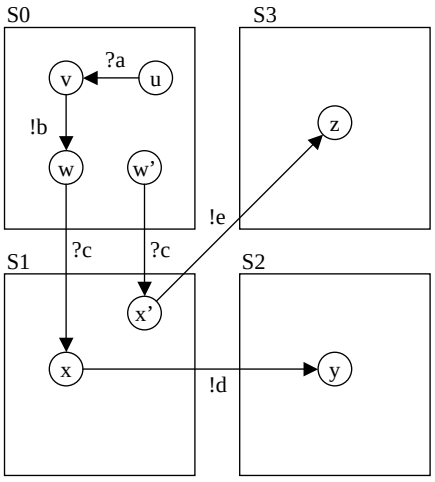

Figure 3: Phase automaton with disjoint states

Here we have depicted a phase as a box that surrounds those states that are part of the phase according to the phase function ϕ. This automaton does not combine the phase transitions it merely enumerates the phase traces as if they are completely unconnected.

From a number of studies we have formalised phase automata as the way engineers intuitively combine multiple MSC scenarios. That is:

> A phase automaton is an implementation of a set of phase traces if during any execution, if it has generated the first part of a specification phase

trace, and it has reached a point where a phase transition is possible then the implementation can always generate the rest of the phase trace from that point.

For the phase traces of Figure 2 applying these informal semantics would lead to the phase automaton implementation of Figure 4.

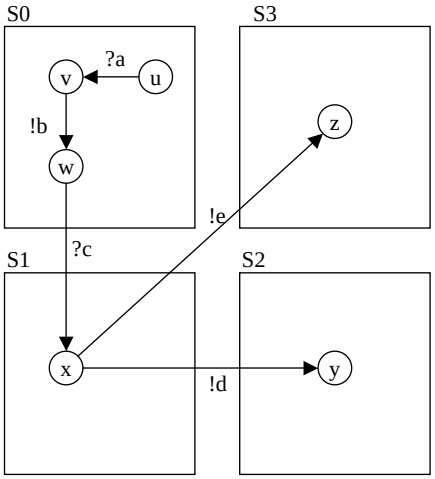

Figure 4: Phase automaton without disjoint states

Consider the initial phase automaton of Figure 3. The transition $w \xrightarrow{?c} x$ generates the first part of the phase trace $S0$, $?c$, $S1$, $!e$, $S3$. Also event $!e$ in this trace causes a phase transition. Hence the informal semantics dictate it must be possible to generate the rest of the phase trace from state x. Hence there must be a transition of the form $x \xrightarrow{!e} z$. Once this transition has been added the resulting phase automaton satisfies the informal semantics for these phase traces. Finally we can discard states w' and x' as they are now redundant. That is the phase automaton of Figure 4 is an implementation of the phase traces for instance A in Figure 2.

Note, had we combined the phase traces in the opposite order we would not have defined exactly the same automaton with exactly the same states as Figure 4. However there is an equivalence between any phase automaton that is an implementation for a set of phase traces, as described in section 4.2.

4 Formal semantics of phase automaton implementation

Suppose we are given a phase trace t of events for one instance A from an MSC which partially specifies some system. Let the phases in t be S_i, for $0 \leq i \leq m+1$. For $0 \leq i \leq m$ let the trace events of t in S_i be a^i_j, for $1 \leq j \leq n_i$. Hence t is of this form

$$S_0,\, a^0_0,\, S_0,\, a^0_1,\, S_0,\, \cdots,\, S_0,\, a^0_{n_0},\, S_1,\, a^1_0,\, S_1,\, \cdots,\, S_m,\, a^m_{n_m},\, S_{m+1}$$

That is a^0_0 is the first event in t, which occurs in phase S_0. For this to be true it must be that the MSC has an initial condition S_0 connected to instance A. After that all the events a^0_i, for $1 \leq i \leq n_0$, occur in phase S_0. Event a^1_0 is the first event to occur in phase S_1. This means that the next condition to be connected to instance A after S_0 is S_1 which occurs between the $a^0_{n_0}$ event and the a^1_0 event. The final event is $a^m_{n_m}$ which occurs in phase S_m, after which the current phase becomes S_{m+1}.

Let $\mathcal{A} = (S, E, \partial, \phi)$ be a phase automaton. We say state s is in phase S_i when $\phi(s) = S_i$. Define a state s to be an exit state when there is a transition to a state s' where $\phi(s') \neq \phi(s)$. The state s' is known as an entry state. Phase automaton $\mathcal{A}$ is defined to satisfy the phase semantics of t if the following conditions are true.

1. Given states $s(i,j)$ in phase S_i, for each $0 \leq i \leq m$, and $0 \leq j \leq n_i$, and *any* sequence of transitions in $\mathcal{A}$ of the form

$$s(0,0) \xrightarrow{a^0_0} s(0,1) \xrightarrow{a^0_1} \cdots s(0,n_0) \xrightarrow{a^0_{n_0}} s(1,0) \cdots s(k,n_k-1) \xrightarrow{a^k_{n_k-1}} s(k,n_k)$$

   where $0 \leq k \leq m$ and $s(k,n_k)$ is an exit state of S_k, then there is a transition sequence of the form

$$s(k,n_k) \xrightarrow{a^k_{n_k}} s(k+1,0) \cdots \xrightarrow{a^m_{n_m}} s(m,n_m+1)$$

   Note by definition $s(i+1,0)$ is an entry state of phase S_{i+1}

2. $\mathcal{A}$ generates t.

Without forcing there to be at least one sequence of transitions in $\mathcal{A}$ which generate t it could be that the first condition is vacuously true.

Phase automaton $\mathcal{A}$ satisfies the phase semantics of a set of phase traces if it satisfies the phase semantics of each of the traces as above.

Phase automaton $\mathcal{A}$ satisfies the phase semantics of an instance in a set of MSCs if it satisfies the phase semantics of all the phase traces belonging to that instance constructed from the MSCs. When $\mathcal{A}$ is a minimal[1] such automaton it is defined to implement the instance.

4.1 Combining phase traces

This section outlines how finite phase traces can be combined into a phase automaton that satisfies the formal specification of section 8. The suggested technique is very inefficient, it is intended for reference only. It is straightforward to define a phase automaton that generates the phase traces for a single instance in a single MSC (when it is race free or finite). The FATCAT tool uses an optimal patented algorithm (designed by Robert Thomson) that directly combines these phase automata into the phase automaton implementation.

For each phase trace τ proceed as follows. Suppose τ is of the form:

$$S_0,\ a_0,\ S_1,\ a_1,\ \cdots,\ a_n,\ S_{n+1}$$

where each S_i is not necessarily distinct from the other phases. Define new states x_i in phase S_i and transitions $x_i \xrightarrow{a_i} x_{i+1}$. This ensures the phase automaton can generate τ.

Define a phase precursor to be an initial section s of τ, where the following event in τ after s causes a phase change. Let s' be the tail of τ occurring after s. Enumerate the phase precursors of τ.

For each precursor s and each state x, test if there is a state x' such that there is a path from x to x' that generates s. If so add states and transitions (if necessary) to ensure that s' can be generated from x'.

[1] i.e there is no subautomaton that also satisfies these phase traces

Next force the resultant automaton to be deterministic whilst preserving the phase structure. Finally minimise the phase automaton with a standard state reduction algorithm whilst again preserving the phase structure of the automaton. Continue the above steps until there are no more phase traces to be considered.

4.2 Phase automaton implementation equivalence

We may regard any finite state automaton as a process that can be described by a process algebra such as CCS [**15**]or CSP [**11**]. Recall the simulation relation $\sqsubset$ between processes can be defined as:

$$P \sqsubset Q \text{ iff for each } P' \text{ such that } P \xrightarrow{a} P' \text{ there is some } Q' \text{ such that } Q \xrightarrow{a} Q' \text{ and } P' \sqsubset Q'$$

From a phase automaton $\mathcal{A} = (S, E, \partial, \phi)$ we can define another automaton $X(\mathcal{A})$ where the states of $X(\mathcal{A})$ are the same as $\mathcal{A}$. A start state of $X(\mathcal{A})$ is any state in $\mathcal{A}$ from which it is possible to generate some semantic phase trace. (We should really only have a single start state in an automaton. This can be achieved by introducing a single new state, *the* start state, which has an ϵ transition to each of the former start states as defined above.) All the states are accepting states in $X(\mathcal{A})$. The events of $X(\mathcal{A})$ are $P \times E \times P$, and the transitions are

$$\partial(x_i,\ (S_i, a_i, S_{i+1})) = x_{i+1}$$

where $x_i \xrightarrow{a_i} x_{i+1}$ is a transition in $\mathcal{A}$, $\phi(x_i) = S_i$, and $\phi(x_{i+1}) = S_{i+1}$.

Given two phase automaton implementations $\mathcal{A}_1$ and $\mathcal{A}_2$ for the same set of phase traces, they are *simulation equivalent* in the sense that:

$$X(\mathcal{A}_1) \sqsubset X(\mathcal{A}_2) \text{ and } X(\mathcal{A}_2) \sqsubset X(\mathcal{A}_1)$$

The main motivation for considering phase automaton is to permit the detection of feature interactions that can be found by static analysis of the phase automaton. The types of error that are defined later are invariant with respect to simulation equivalence.

5 Phase transition errors

It is possible to statically analyse phase automaton to detect certain simple types of conflict without user input. More sophisticated conflict analysis requires additional properties of the specifications to be defined that describe the purpose of the features in a form that can be verified against the automaton. A phase automaton can be verified with standard model checking techniques against any modal property, which is ongoing research. Care must be exercised since, for example, searching for unreachable states is not appropriate for partial requirements.

We describe two types of errors that can be statically detected during the construction of the phase automaton. There is anecdotal evidence to suggest that these kinds of inconsistency can account for a significant proportion of feature interactions.

Phase inconsistencies

A significant static error that can occur is where two phase traces define the same events initially, but disagree with the phase transition that later occurs. Here is an example of two such phase traces.

$$\begin{array}{cccccc} S0, & ?u, & S0, & !a, & S1 \\ & & S0, & !a, & S2 \end{array}$$

The phase semantics force there to be two distinct transitions labeled with $!a$ leading to different phases from the same state. In general this conflict leads to a nondeterministic automaton where the next composite state is not uniquely defined. A detailed browser example is given in Section 6.

Structural inconsistencies

The simplest form of static analysis is to detect certain kinds of nondeterministic behaviour taken by the instance implemented by the automaton. For example, when the instance has to make a choice between sending one of two messages. Consider a state x and transitions

$$\begin{array}{ccc} x & \xrightarrow{!a} & x_1 \\ x & \xrightarrow{!b} & x_2 \end{array}$$

where a and b are distinct. A system component will not know whether it should send a or send b.

6 A realistic example for mobile handset browser requirements

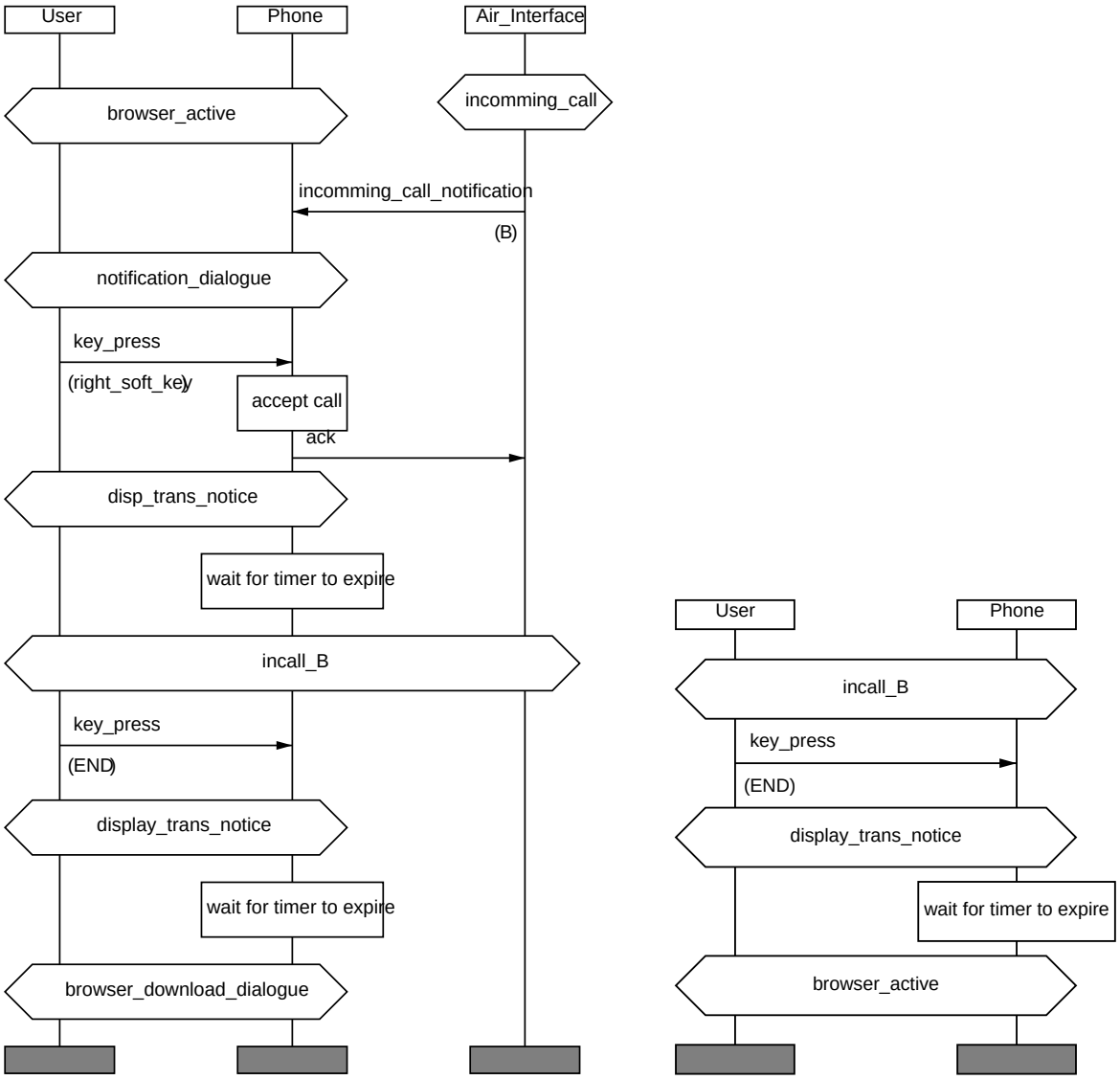

Figure 5: Browser download suspend example, MSC M_1 and MSC M_2

The MSCs of Figure 5 describes two hypothetical MSC requirements for a browser like feature for a mobile handset. Let these be MSC M_1 and M_2 as numbered from left to right. M_1 describes a high level scenario where the handset browser is active downloading a file when a call is received from another device. The download is suspended while the call is dealt with. After the call is terminated the handset presents a dialogue box to enable the user to resume the file download if they wish.

M_2 describes a different scenario relating to the general operation of the browser like device. This scenario describes how the handset should behave when the browser has been suspended during an active call, and then the END key is pressed. The scenario states that the phone should always return to the `browser_active` composite state.

Notice that if both of these scenarios are applied to the browser feature then they cause a conflict. When a file download is suspended in order to take a call, and then the

END key is pressed, the handset is trapped between the `browser_download_dialogue` and `browser_active` composite states. That is the next composite state after the timer expires is not uniquely defined, which is an error.

The phase automaton of Figure 6 is the FATCAT implementation for the phone instances of M_1 and M_2. The states are labelled with integers from 0 to 10. The phases are depicted by the grey boxes surrounding the states contained by the relevant phase. Diamond shapes represent an exit state for a phase. Square states are entry states, that is a state that can be reached by a transition from a different phase. States that are a square superimposed over a diamond are both an entry and exit state.

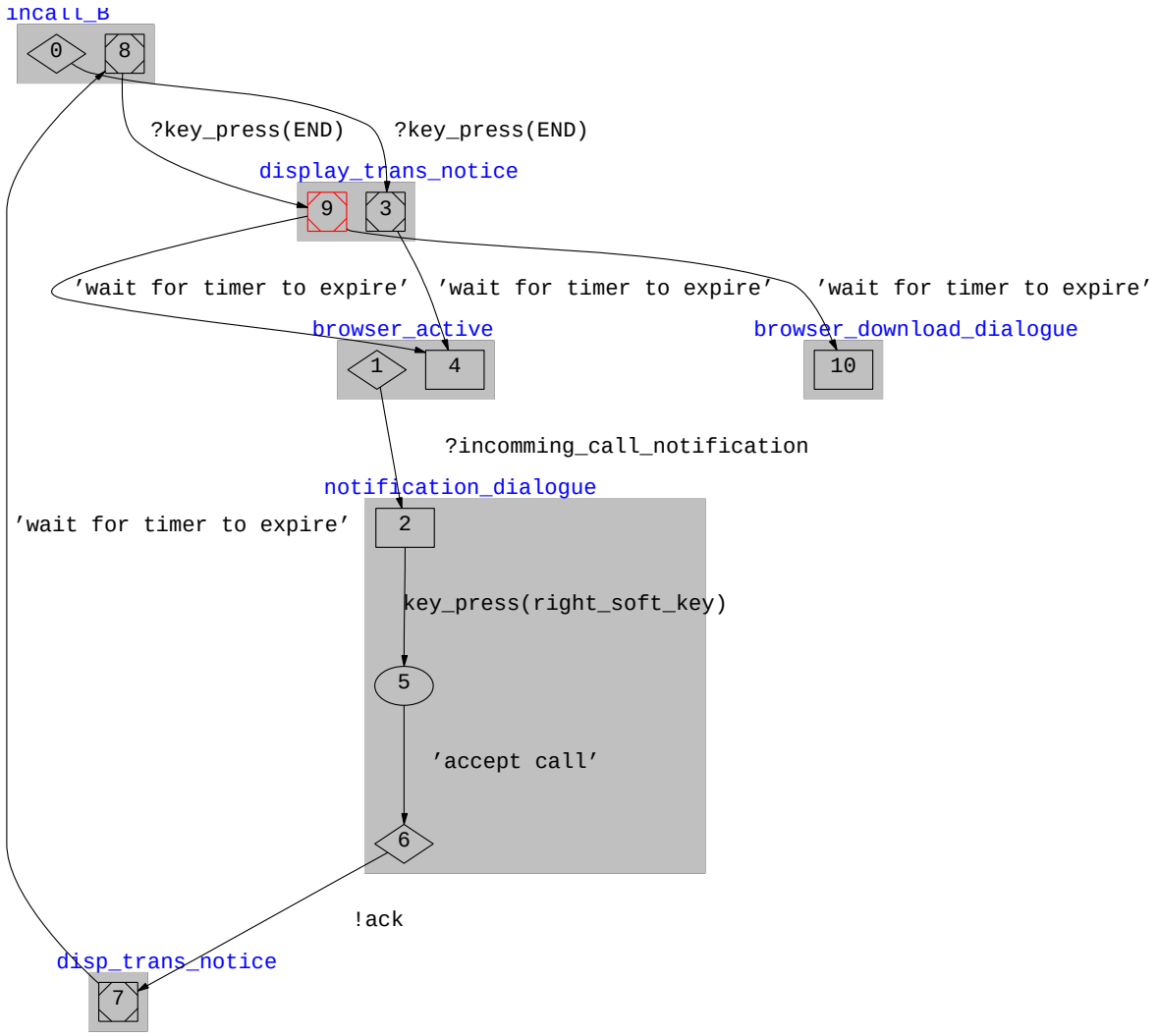

Figure 6: FATCAT phase automaton implementing phone instance

State 9 represents an error state, since from there the next state is not uniquely defined after the timer expires. Note there is a trace from state 1 to state 9. This illustrates how the error state can be reached assuming that state 1 can be reached from some initial configuration. This seems reasonable since state 1 belongs to phase `browser_active` from Figure5, which is the initial phase for that scenario. Hence the trace from state 1 to state 9 represents an example of how the conflict can occur in practise and is the one chosen by FATCAT for output in MSC format.

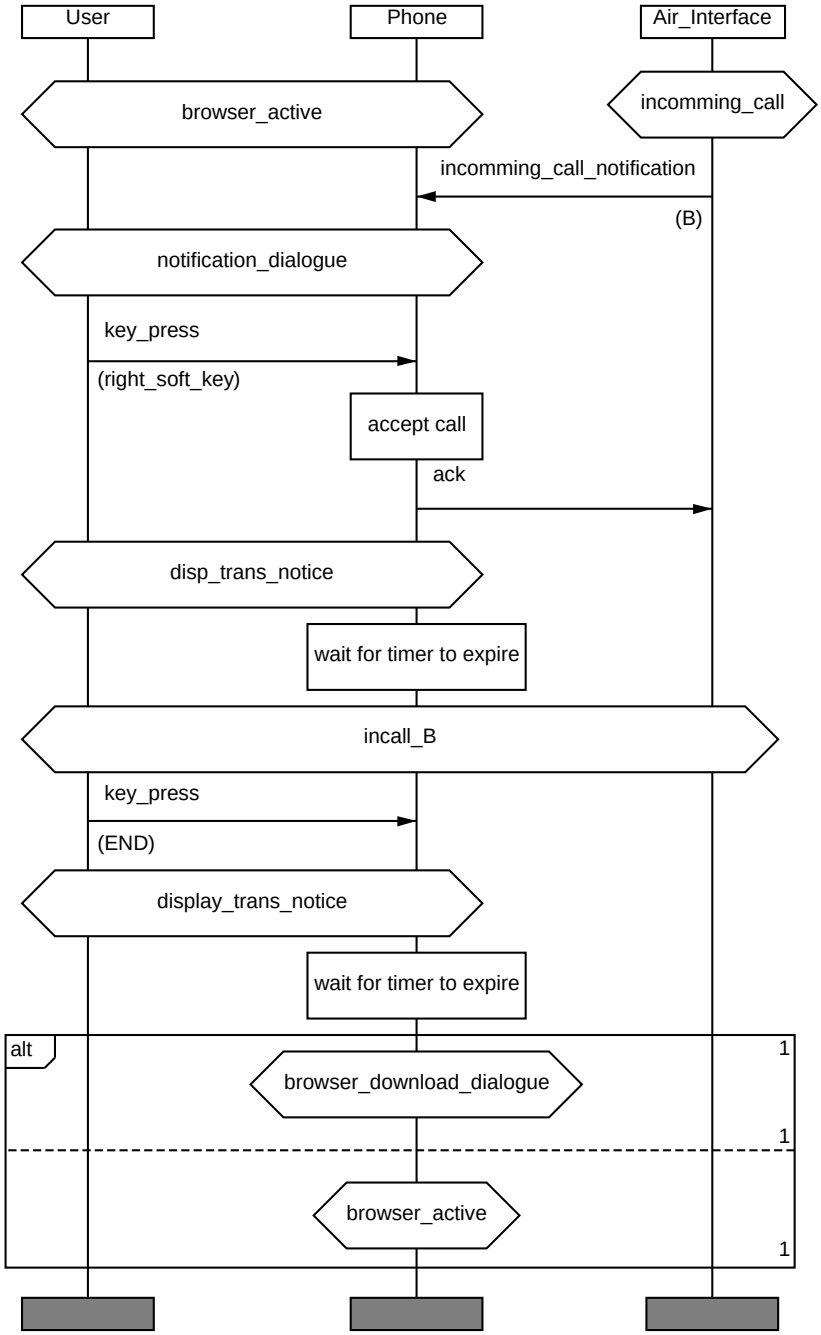

Figure 7: FATCAT example of conflict

This trace can be represented with an MSC as in Figure 7. The conflict in this MSC is represented by the alternative construct at the end of the MSC. This illustrates that after the timer expires the next composite state for the phone instance is not uniquely defined. Recall an MSC alternative construct describes mutually exclusive possibilities. The dashed line delineates these possibilities.

7 Three way interaction

This section gives an example of how three MSC scenarios may interact to cause a conflict, and where no two of those scenario are in conflict when considered separately.

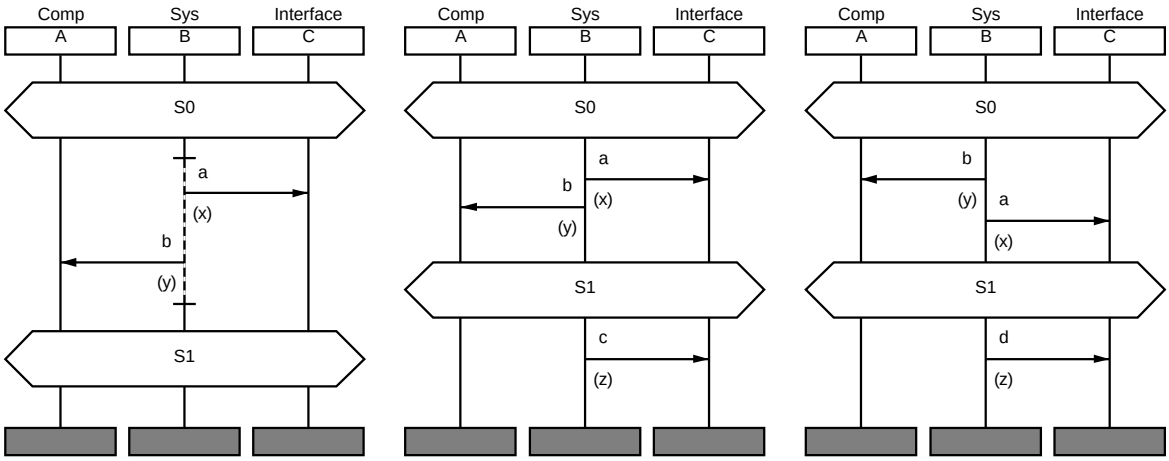

Figure 8: Three MSCs that conflict only when all three are included

The dashed line in the left most MSC of Figure 8 defines a co-region. Events within such a co-region can occur in any order with respect to one another. The FATCAT tool detects a conflict between the MSCs of Figure 8 and outputs the MSC in Figure 10 to describe how it occurs. As before the error is captured in the form of the alternative construct. Figure 9 describes the phase automaton implementation of instance B output by FATCAT for the three MSC scenarios in Figure 8.

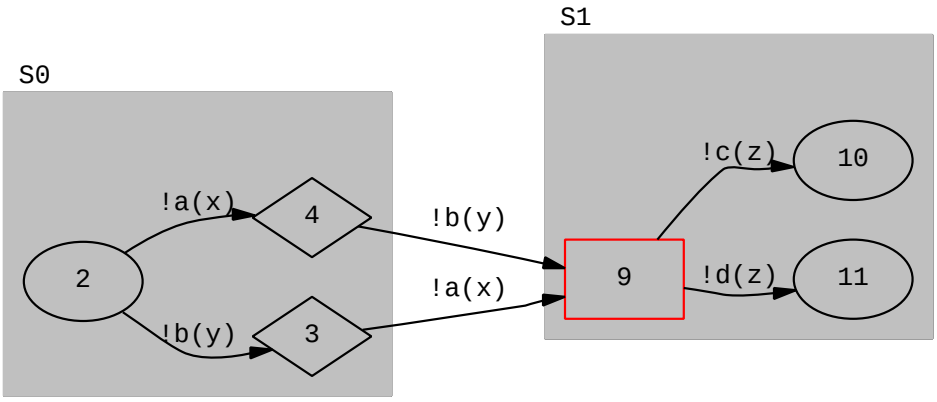

Figure 9: Phase automaton implementation for instance B

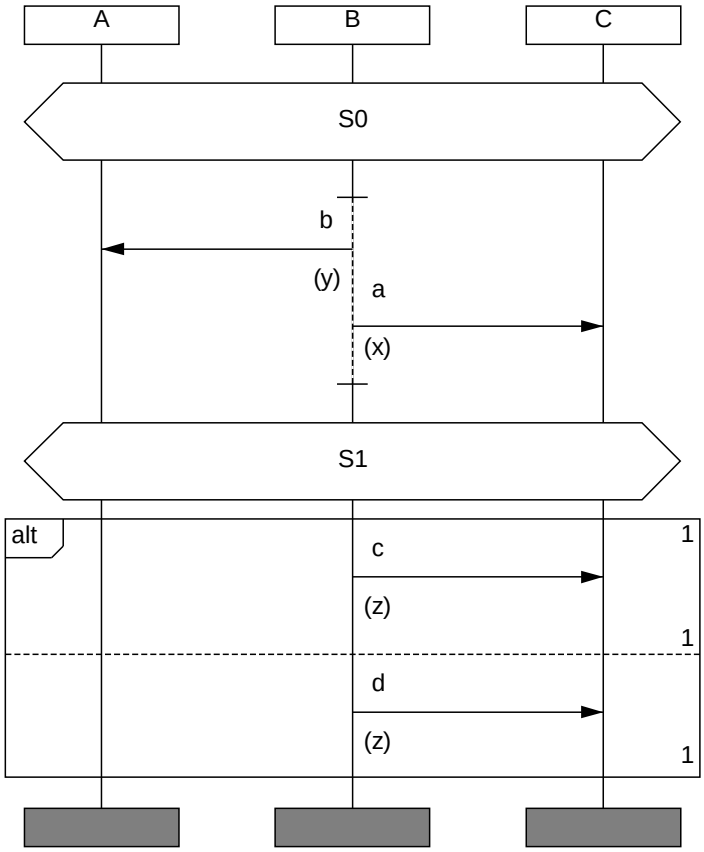

Figure 10: MSC conflict output from FATCAT

8 Process Algebra Products for Phase Automaton

This section defines the semantics of composing MSC scenarios in the form of a process algebra. MSC scenarios can be mapped to terms in the process algebra. The delayed product of the scenarios generates processes that are simulation equivalent to phase automaton implementations. The discussion so far has been implicitly restricted to finite phase traces, whereas the process algebra description here generalises the idea to recursive process specifications.

The process algebra semantics is divided into two parts. The algebra composition operators are defined as a set of axioms over an arbitrary set of process terms. Then a particular set of atomic actions is chosen to represent the phase traces of a set of MSCs. This allows us to hide some of the semantic complexity in the atomic actions and represent the semantics of phase trace composition in a clean abstract setting.

Let E be a set of atomic actions. Let $+$ be the usual choice operator over process terms. For a set U define $\sum U$ to denote $\sum_{u \in U} u$. Also let $\cdot$ be the usual composition operator of atomic actions and process terms. The set of terms defined by $+$ and $\cdot$ over E is the set of process terms. A process term is a representation of all the possible traces of actions that the process can define. Each branch of the term represents one of the traces. Let $\sqsupset$ be a binary reflexive relation over E and let $\eta : E \longrightarrow \mathbb{B}$ be a

boolean valued function. Define composition of process terms with these axioms.

par	$P \mid Q$	$=$	$P \lhd Q + P \rhd Q$
skew1	$a \cdot P \lhd b \cdot Q$	$=$	$a \cdot P \lhd\mid b \cdot Q$ if $a \sqsupset b$
skew2	$a \cdot P \lhd b \cdot Q$	$=$	$a \cdot (P \lhd b \cdot Q)$ if $a \not\sqsupset b$
skew3	$P \rhd Q$	$=$	$Q \lhd P$
zero1	$0 \lhd Q$	$=$	0
left-synch1	$a \cdot P \lhd\mid b \cdot Q$	$=$	$a \cdot (P \lhd\mid Q)$ if $a \sqsupset b$ and $\neg\eta(a)$
left-synch2	$a \cdot P \lhd\mid b \cdot Q$	$=$	$a \cdot (P \parallel Q)$ if $a \sqsupset b$ and $\eta(a)$
branch1	$a \cdot P \lhd\mid b \cdot Q$	$=$	$a \cdot P + b \cdot Q$ if $a \not\sqsupset b$ and $\eta(a)$
prune	$a \cdot P \lhd\mid b \cdot Q$	$=$	$a \cdot P$ if $a \not\sqsupset b$ and $\neg\eta(a)$
zero2	$0 \lhd\mid Q$	$=$	0
synch1	$a \cdot P \parallel b \cdot Q$	$=$	$a \cdot (P \parallel Q)$ if $a \sqsupset b$
synch2	$P \parallel Q$	$=$	$Q \parallel P$
zero3	$0 \parallel Q$	$=$	Q
branch2	$a \cdot P \parallel b \cdot Q$	$=$	$a \cdot P + b \cdot Q$ if $a \not\sqsupset b$ and $b \not\sqsupset a$

$P \mid Q$ is defined to be the delayed composition of process algebra terms.

Define the stack automaton $\mathsf{Sk}(I)$ for I in M as follows. Let $\mathcal{A}$ be any phase automaton that generates the phase traces of I in M. Note $\mathcal{A}$ is not a phase automaton implementation of I, which has to correctly combine all the phase traces for I from all the MSCs that define the specification. Constructing $\mathcal{A}$ is straightforward when M does not contain any race conditions. It can be derived from any automaton that generates the event traces of I by annotating the states with suitable phase information. Note when M contains infinite traces and there are race events it is quite possible that $\mathcal{A}$ does not exist.

A state in $\mathsf{Sk}(I)$ is a pair (s, Sk), where s is a state of $X(\mathcal{A})$, and Sk is a stack containing events from I (see section 4.2 for the definition of $X(\mathcal{A})$). A transition in $\mathsf{Sk}(I)$ is defined as follows. Let $\partial(s,\, (c, a, c')) = s'$ be a a transition in $X(\mathcal{A})$. Let Sk' be the result of pushing a onto Sk whenever $c = c'$, and set $Sk' = [\,]$ when $c \neq c'$. Then

$$\partial((s, Sk),\, (c, a, c')) = (s', Sk')$$

is a transition in $\mathsf{Sk}(I)$. A start state of $\mathsf{Sk}(I)$ is a pair $(s_0, [\,])$, where s_0 is a start state of $X(\mathcal{A})$.

The traces of $\mathsf{Sk}(I)$ are really the phase traces of I in M. The stacks simply record what events have happened on the instance since the phase last changed. The stacks record what has happened on an instance during the current phase.

For two stacks sk_1 and sk_2, we write $sk_1 \leq sk_2$ if stack sk_1 is the head of stack sk_2. In other words, if sk_1 has n elements, and if we pop each stack n times, we get exactly the same element from each stack each time. We will now define a process algebra term with the set of atomic actions equal to $P \times E \times P \times \mathsf{SK}$, where SK is the set of event stacks.

Define $(c, a, c', Sk) \sqsupset (c_1, a_1, c_1', Sk')$ when $c = c_1$, $a = a_1$, $c = c_1'$ and $Sk' \leq Sk$. Define $\eta(c_1, a_1, c_1', Sk')$ to be true exactly when $c_1 \neq c_1'$. So that η records when an event causes a phase transition. For a stack automaton $\mathsf{Sk}(I)$ define its process algebra term inductively. Let $P(s, Sk, \mathsf{Sk}(I))$ be

$$\sum\{(c, a, c', Sk) \cdot P(s', Sk', \mathsf{Sk}(I)) \mid (s, Sk) \xrightarrow{(c,a,c')} (s', Sk') \in \mathsf{Sk}(I)\}$$

Then define $P(I, M)$ to be the sum of terms $P(s_0, [\,], \mathsf{Sk}(I))$ where $(s_0, [\,])$ is a start state for $\mathsf{Sk}(I)$. The stack component in the atomic actions of $P(I, M)$ are used solely as a mechanism for controlling the delayed composition of these terms. The stacks are not relevant to the observed behaviour in the phase traces and once the composition has been evaluated they are of no more use.

For an instance I that occurs in MSCs M_i, where $0 \leq i \leq n$, let

$$P(I) = P(I, M_0) \mid P(I, M_1) \cdots \mid P(I, M_n)$$

Let $P'(I)$ be the normalised form of the process algebra term $P(I)$. This is the process algebra term defined only using the operators $+$, $\cdot$ and recursion that is strong bisimulation equivalent to $P(I)$. Let $P^*(I)$ be the result of replacing every atomic action in $P'(I)$ of the form (c, a, c', Sk) with (c, a, c'), that is delete the stack component from each atomic action. Then any phase automaton implementation $\mathcal{A}_1$ of instance I is simulation equivalent to $P^*(I)$ in that

$$X(\mathcal{A}_1) \sqsubset P^*(I) \text{ and } P^*(I) \sqsubset X(\mathcal{A}_1)$$

as defined in Section 4.2.

9 Conclusion

Where the purpose of a protocol specification is to define phase transitions it is possible to construct a phase automaton implementation for each process in the specification. These can then be statically analysed to detect phase transition errors. With the right semantic interpretation of phase transitions the phase automaton can be constructed efficiently, and is capable of detecting interesting interactions. Compared to automata constructed using naive compositional semantics, our phase automata approach yields highly compact representations enabling the tractable analysis techniques reported here.

Unlike most work in the area of feature interaction (see [**9**], [**10**] or [**7**] for a comprehensive set of examples) the phase automaton technique is not intended to detect all possible conflicts. Rather it detects phase transition interactions. This class of conflict seems to be of practical significance and simple to detect. Further, specifications are often deliberately of a highly partial nature. In such cases it is important to have some means of detecting errors that are inherent in the specification as it is meant to be implemented, and not highlight errors that are purely an artifact of the incomplete nature of the specification. For this reason the techniques described here are designed to detect persistent errors that will be present however the specifications are extended into a more complete form.

References

[1] R. Alur and M. Yannakakis, Model checking of message sequence charts, Proceedings of the Tenth International Conference on Concurrency Theory, Springer Verlag, 1999
`http://www.cis.upenn.edu/~alur/onlinepub.html`

[2] R. Alur, K. Etessami, M. Yannakakis, Inference of Message Sequence Charts, Proceedings 22nd International Conference on Software Engineering, pp 304-313, 2000.

[3] P. Baker, P. Bristow, C. Jervis, D. King, B. Mitchell, Automatic Generation of Conformance Tests From Message Sequence Charts, Proceedings of 3rd SAM (SDL And MSC) Workshop, Telecommunication and Beyond, Aberystwyth 24th-26th June 2002, to appear in LNCS 2003.

[4] P. Baker, C. Jervis, D. King, An optimised algorithm for test script generation, patent GB18137.0, 2000.

[5] P. Baker, C. Jervis, B. Mitchell, Method of Generating Coordinating Messages for Distributed Test Scripts, patent GB18138.8, 2000.

[6] M. Calder, A. Miller, Using Spin for Feature Interaction Analysis - a Case Study, Proceedings of SPIN 2001, Lecture Notes in Computer Science, Volume 2057, pp. 143–162, 2001.
`http://www.dcs.gla.ac.uk/~muffy/papers.html`

[7] M. Calder, E. Magil, Feature Interaction in Telecommunications and Software Systems VI, IOS, 2000.

[8] R. Hall, Feature Combination and Interaction Detection via Foreground/Background Models in [**10**] also found at
`ftp://ftp.research.att.com/dist/hall/papers/isat/feature-interactions-fiw98.ps`

[9] N.Griffeth, R. Blumenthal, J-C, Gregorie, T. Ohta, A feature Interaction Benchmark for the first feature interaction detection contest, in journal of Computer Networks, Vol 32, No 4,April 2000

[10] K. Kimbler, L. G. Bouma, Feature Interaction in Telecommunications and Software Systems V, IOS, 1998.

[11] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[12] P. Madhusudan, Reasoning about Sequential and Branching Behaviours of Message Sequence Graphs, proceedings of 28th International Colloquium on Automata, Languages and Programming, Crete, Greece 8-12 July 2001, LNCS 2076.

[13] S. Mauw, M. van Wijk, and T. Winter. A Formal Semantics of Synchronous Interworkings. In O. Faergemand and A. Sarma, editors, SDL'93 Using Objects, Proceedings of the Sixth SDL Forum, pages 167-178, Darmstadt, 1993. Elsevier Science Publishers, Amsterdam. ISBN 0-444-81486-8.
`http://citeseer.nj.nec.com/mauw93formal.html`

[14] S. Mauw, M.A. Reniers, A process algebra for Interworkings,
`http://citeseer.nj.nec.com/mauw00proces.html`

[15] R. Milner, Communication and Concurrency, Prentice Hall 1989.

[16] Z.120 (11/99)ITU-T Recommendation - Message Sequence Chart (MSC)
`http://www.itu.int/itudoc/itu-t/approved/z/index.html`

[17] Z.100 (11/99) ITU-T Recommendation - Languages for telecommunications applications - Specification and description language
`http://www.itu.int/itudoc/itu-t/approved/z/index.html`

[18] Annex C, Service Diagrams related to the model of Mobile user, Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Designers' guide; Part 2: Radio channels, network protocols and service performance, European Telecommunications Standards Institute 1997.