Cutting and Pasting with Requirements Specifications Scenarios

Bill Mitchell [a], Robert Thomson [b], Clive Jervis [b], Paul Bristow [b],

[a] *Department of Computing, University of Surrey Guilford, Surrey GU2 7XH, UK*

[b] *Motorola UK Research Lab, RG22 4PD, UK*

Abstract

Wireless Telecommunications requirements specifications tend to be defined as sets of normative scenarios. These frequently only provide partial coverage of the scenarios that are necessary to give a comprehensive specification. Standard model checking techniques have not been successful in analysing such protocol specifications, first because of the high degree of concurrency that is inherent in such systems, and secondly because the very partial nature of the specifications tends to cause many inconsequential defects to be reported that frustrate the process of identifying key issues that have to be addressed immediately. Typically the inconsequential defects are addressed by adding new scenarios to the requirements, whereas significant defects require structural changes to the design itself and are not solved by adding new scenarios.

This paper describes a technique for synthesising tractable *phase automata* from Message Sequence Chart scenarios that describe major phase transitions in the specifications. These can be automatically analysed to detect certain types of significant interactions between the scenarios that will be invariant under the addition of new scenarios to the requirements specifications.

Key words: Requirements Scenarios, Tractable Model Synthesis, Feature Interaction, Message Sequence Charts

Email addresses: w.mitchell@surrey.ac.uk (Bill Mitchell), brt007@motorola.com (Robert Thomson), clive.jervis@motorola.com (Clive Jervis), paul.bristow@motorola.com (Paul Bristow).

1 Introduction

The International Telecommunications Union (ITU) is a specialised agency of the United Nations. It is responsible for coordinating global telecom networks and services for private companies and government bodies. The European Telecommunications Standards Institute (ETSI) defines the protocol standards for fifty six countries within the EU and beyond. Together ETSI and the ITU also define the standards for languages and notations that are used for requirements and architecture specifications in the telecommunications industry throughout the world.

Message Sequence Charts (MSC) [17] are an international standard defined by the ITU for capturing requirements specification scenarios. ETSI recommend their use in defining requirements specification scenarios for their standards documents. Due to the extreme pressure to rapidly prototype designs, and bring them to market, it is often the case that MSC scenarios make up virtually the whole of the requirements specifications from which architecture designs are derived and code developed. Often the MSC scenarios only partially cover all the possible scenarios that are needed for a comprehensive description of the particular protocol. Often they only cover 'sunny day' scenarios for each individual feature. This can lead to architecture designs and final products that contain many defects. This makes it imperative to have a mechanism for analysing what interactions there are between the different MSC requirements, which frequently describe the behaviours of features in isolation without taking into account other features that may be concurrent with them.

Much work has been done on synthesising system models from MSC scenarios. These techniques have tended to result in intractable models for industrial sized specifications. They also reflect the simple message exchange aspects of the scenarios rather than the central purpose of the protocols they are specifying. In this paper we construct phase automata to represent the overall system behaviour of each process in the scenarios. These reflect the transitions between the major operational phases of the scenarios, rather than the entire possible set of concurrent behaviour. This leads to tractable models that can be statically analysed for certain types of interactions that are significant to the system development.

Intuitively these automata describe a simple mechanism for overlapping different scenarios when they meet certain semantic criteria that makes the overlap valid. This overlapping provides a form of abstract cutting and pasting that permits new scenarios to be automatically generated that describe interactions between the different requirements. The difficulty arises in defining the correct semantics so that useful overlaps are permitted, and inconsequential, or incorrect overlaps are not. The paper defines one particular semantics that

has proved to be useful for Motorola in analysing requirements specifications for 3G products they are currently developing. The paper illustrates the semantics with a realistic example of requirements specification scenarios for the high level functionality of a WAP browser application.

A terse discussion of the work reported in this paper was first presented in [14], which discussed an elementary example of a WAP browser requirements specification. This paper provides a more detailed discussion of the phase semantics, including more detailed examples, and contains a discussion of the difference between explicit state semantics and phase semantics, which was not discussed in the preliminary report.

Phase Traces

This paper looks at MSC scenarios that define 'phase transitions'. These are MSC scenarios that define the transitions between major operational phases of a protocol in conjunction with the events that cause the transitions between these phases. It is typical for such phase information to be encoded within the MSC condition symbols. We will adopt the same convention here. Examples of major phases for a telecommunications protocol such as TETRA [19] are *call setup*, *call active*, *call roaming*, *call queued*, and *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*.

From an MSC defining phase transitions we can derive the set of 'phase traces'. These are the event traces defined by the MSC where we annotate each event with the phases that were active just before and just after the event. Usually the MSC phases correspond to significant operational phases of the system, and so do not normally change after an event. When they do this is an important point to focus analysis for interaction detection.

A phase transition can be triggered by different phase traces described by different MSC scenarios. If we assume that each process in an MSC can be represented as a deterministic finite state automata (which is not always theoretically possible for MSC-s that include iteration), it is a non trivial problem to decide how to combine the different phase transitions of each phase within the automata. For this paper it is important to define the automaton in such a way that it is tractable and does not reflect concurrent behaviour that is not directly relevant to the phase transitions. This is because the interaction analysis that is defined for these automata searches for implicit phase transitions that were not explicitly contained in the original requirements scenarios. Anec-

dotal evidence from Motorola case studies has shown that these interactions are significant.

Related Work

Alur and Yannakakis in [1] have proved that the general property checking problem for a restricted set of iterative message sequence charts is undecidable. They also define criteria for when the problem becomes decidable, but also NP complete. In [1] they define an algorithm for constructing a finite state automaton that describes all the interleaving execution traces defined by a set of MSC scenarios when their criteria is satisfied. Madhusudan [10] defines a second order monadic logic that is decidable for a larger class of MSC scenarios, which of course must also lead to NP complete algorithms. This work focuses on synthesising models of the event traces defined by MSC requirements specifications, and is not concerned with the phase transitions that such scenarios usually describe.

In ([11], [12]) they consider a form of interleaving to compose interworkings, which are similar in spirit to MSCs. This approach also works with event traces rather than with phase traces. Superficially this is very similar to the work reported here. Indeed their intention is to allow a collection of scenarios described in terms of interworking diagrams to be composed together to describe the system as a whole. They also restrict the form of interleaving they model so that their overall system view is a subset of all possible interleavings, which helps to restrict the complexity that normally accompanies such composition. Within the context of modeling the phase transition behavior of a system, their semantics are very different from those given here. Some of the compositional operators they consider would not be regarded as valid within the context of phase overlapping as defined here.

In [16], [15] they address the problem of synthesising statecharts from MSC scenarios. In particular in [16] they look at how to use global state names that are incorporated in the MSC scenarios using condition symbols to compose MSC-s into statecharts for each process in the specification. The results in [16] are defined for synchronous message passing, whereas wireless telecommunication protocols tend to be asynchronous. For exactly that reason the ITU standard for MSC specifies message passing to be asynchronous. Phases do not generally correspond to explicit states within an implementation. They are more intuitively associated with predicates that guard when a particular region of a process becomes active. Section 3 describes an example based on the TETRA standard to illustrate this point.

2 Cutting and Pasting

This section will describe the intuition behind the abstract cutting and pasting concept that underlies the formal semantics of phase trace composition that will be defined later.

The MSC in figure 1 describes a scenario involving three processes, 'Phone', 'Browser' and 'Air Interface'. Each vertical line is the time line describing the events that occur for each process. Time increase downwards, but nonlinearly. Hence it is not possible to deduce the relative occurrence of events of different processes from their specific locations in the time line. The messages between processes enforce a causal relationship that determines what order events can occur in. However this only defines a partial order between events in general. Messages are asynchronous in MSC-s, so that although by convention they are drawn horizontally, suggesting synchronous transmission, they have arbitrary latency. Throughout this document we will use $!m$ to denote the send event for a message m and $?m$ for the receive event of message m in accordance with the ITU standard for MSC-s.

The MSC notation includes several complex concepts apart from simple message exchange. These include process creation and destruction, inline references to other MSC scenarios, iteration, timing constraints, and branching. Interested readers are referred to the standard [17] for complete details. We will not describe the full semantics of MSC-s here, but will use figure 1 to illustrate some of the main constructs. Note the solid rectangles at the end of each time line do not denote the termination of the process, but the conclusion of the scenario. Similarly the process name boxes at the start of the time lines denote the start of the scenario, not the creation of the processes.

The shortened hexagonal symbols in figure 1 are MSC condition symbols. In this paper we will adopt a particular semantics for these which agrees with common industrial practise, but which is an extension of the MSC standard. Our semantics do not change the standard semantics with relation to any single MSC, or higher order MSC, but permit us to restrict the type of composition we use to synthesis a model of the phase transitions described by the MSC-s. In this paper we assume that MSC condition symbols denote which operational phase is active within each process. The phase remains active until the point in the process time line where the next condition symbol occurs. From now on we will refer to these as phase symbols. For example the 'Air Interface' process remains in the 'Data' phase until the 'EOF' message is transmitted, when it transitions to the 'Channel' phase. It is very tempting to assume these represent states in an implementation, but we will see in section 3 that is not a suitable interpretation in our context.

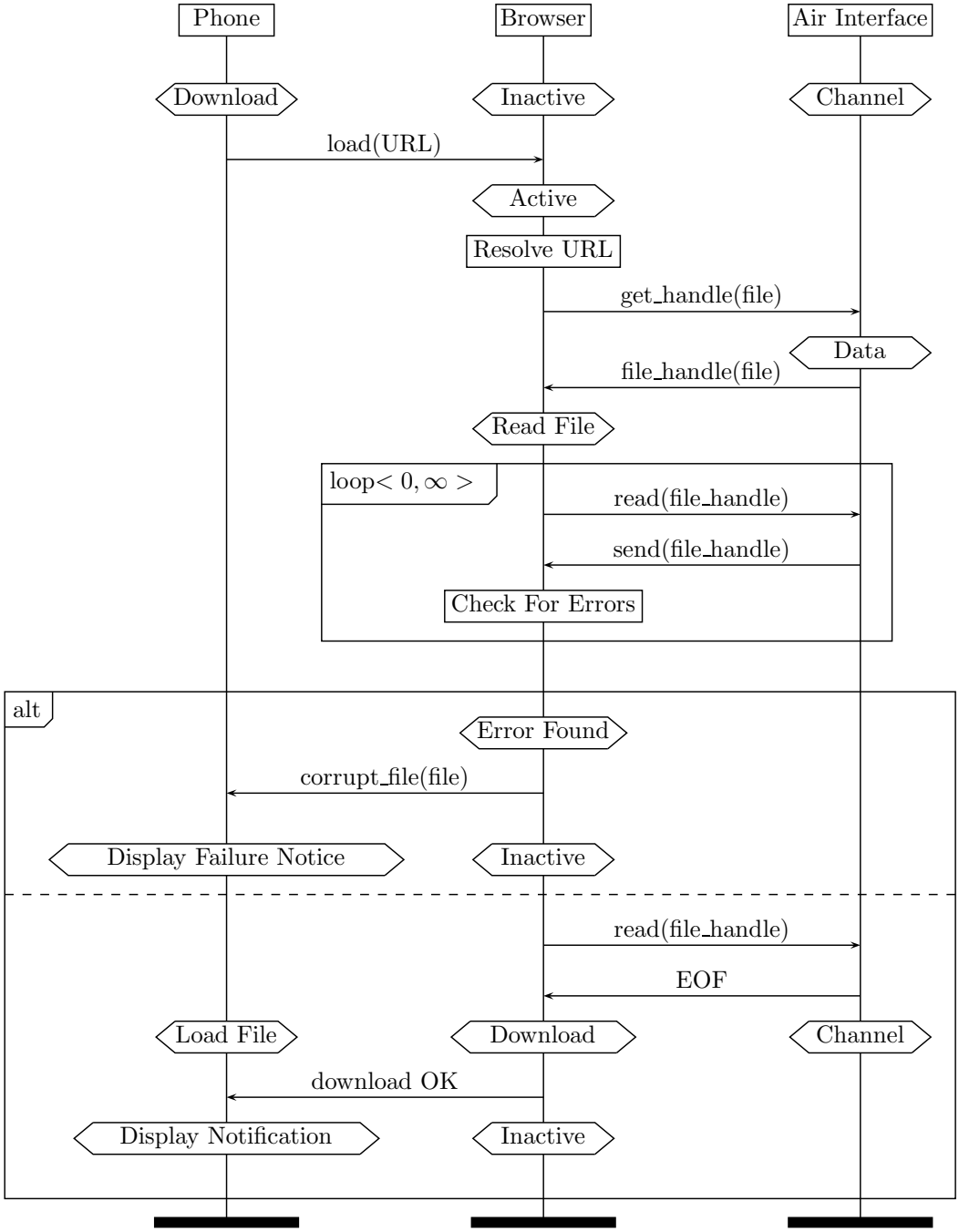

Fig. 1. Error Checking File Download with Iterative MSC

The box labelled with 'loop$< 0, \infty >$' is an iterative loop. It represents that the events within the loop can be unfolded any finite number of times between 0 and ∞. Note this means that the iterated events are inlined into the scenario, which has subtle consequences for the event traces of the MSC. It is because of this that the general property checking problem is undecidable for general MSC scenarios. In figure 1 the loop describes how a file will be iteratively downloaded from the 'Air Interface' by the 'Browser' process, ter-

minated either when the 'EOF' message is received or internal error checking detects a corrupt file.

The box labelled 'alt' (for alternative) describes possible branches that can occur in the scenario. The dotted line across the box delineates the two different possibilities that can occur in this example. In general there can be any finite number of alternatives, including none. The alternatives are mutually exclusive, and the choice of alternative is nondeterministic. MSC scenarios are meant to describe externally observable events of a system, and not describe the internal actions that cause them. Internal actions of processes can be defined by action boxes, which represent internal atomic events which can not be externally witnessed. Process 'Browser' in figure 1 has an action box 'Resolve URL' immediately after the 'Active' phase becomes valid. In practise engineering groups tend to use action boxes to store code fragments during the architecture design stage, though they were not intended for this use.

In summary, figure 1 describes a scenario where the central process controlling a mobile handset, the 'phone' process, delegates the task of downloading a file to the 'Browser' process. This instigates a data channel connection with the 'Air Interface' process. Once this channel is allocated the file is iteratively downloaded until either the download is complete, or an error is detected. The 'Browser' reports back to the 'Phone' the final outcome, and the 'Air Interface' returns to the 'Channel' phase, which is meant to represent the teardown of the data channel, and the switch back to the control signaling channel.

We interpret the initial phase symbols of the scenarios as acting as a kind of universal guard. Thus whenever we observe 'Phone' in phase 'Download', 'Browser' in phase 'Inactive' and the 'Air Interface' in phase 'Channel' it must be possible to witness the events described by the scenario in the order it defines for them. Semantics for the phase symbols that are not initial, but secondary is more complex.

Take the 'Active' phase for the 'Browser' process as an example. It is implicit that this has the following semantics. Whenever the 'Browser' process reaches the 'Active' phase from the 'Inactive' phase by receiving the 'load(URL)' message, then it is always possible to observe the subsequent events described in the MSC scenario.

More generally we can informally define the semantics for each phase symbol thus. If a process transitions to a particular operational phase in accordance with all the events and phase changes that are described in a scenario, then it is always possible from that point to witness the subsequent events that are described by that scenario.

Note these semantics would be correct if each phase is a global state name

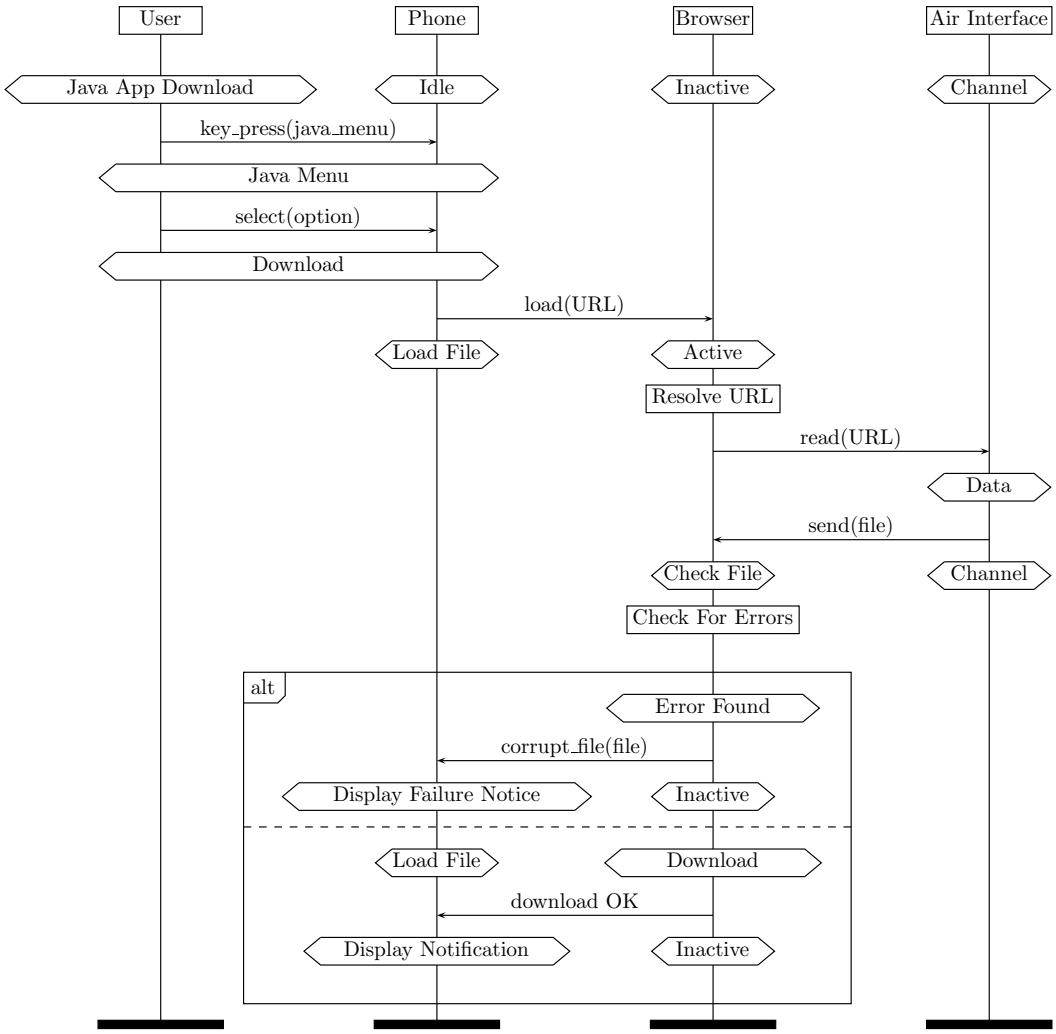

Fig. 2. Java App Download with Dedicated Key

within a statechart, for example. However, here we interpret a phase as having the above dynamic semantics and also identify a phase with a collection of states within an automata representing the behaviour of the relevant scenario process. When a process transitions to a phase, that simply means that a state transition has occurred to a state belonging to the set of states defined by that phase. Now consider a second MSC example, figure 2 which represents a requirement for downloading java applications via a specific menu that is activated by a dedicated key on the handset. In this example the message exchanges to download the file once the 'Browser' has transitioned to the 'Active' phase differ significantly from figure 1. With our informal semantics we can identify how these scenarios can be combined to give an interaction between the two. In figure 2 the 'Browser' process reaches the 'Active' phase from the 'Inactive' phase by receiving the 'load(URL)' message. From our informal semantics from figure 1 that means we can always witness the subsequent events from figure 1 after this point. In other words we can *cut* from figure 2 the events after the 'Active' phase, and *paste* in the corresponding events from

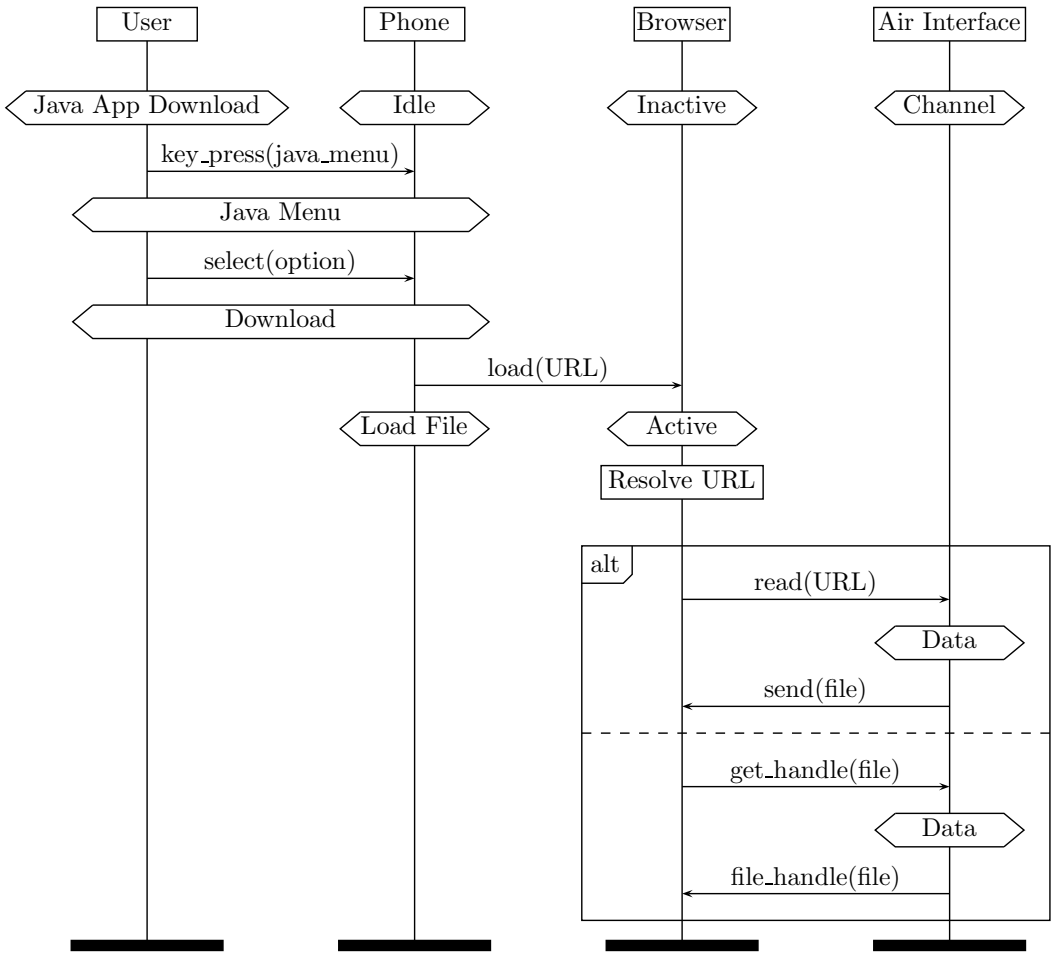

Fig. 3. Scenario difference as an alternative construct

figure 1. We now have two scenarios defining how a Java application is to be downloaded via the 'Browser' process that can not both be valid. We can use the MSC notation with the alternative construct to illustrate at what point these two scenario differ. Figure 3 illustrates how the two scenarios contradict each other with respect to downloading the application. It describes the first place where the new cut and paste scenario will differ from figure 2. Note it does not include all the events that can occur after the two scenarios then diverge. This illustrates a requirements interaction that could result in a defect if not resolved. Clearly the 'Browser' application must be behaving incorrectly in one of these two scenarios once it has resolved the URL it is meant to be downloading.

3 Phases are not States

This section will describe an example of two requirements scenarios that give a very high level view of some aspects of the preemptive priority call (PPC)

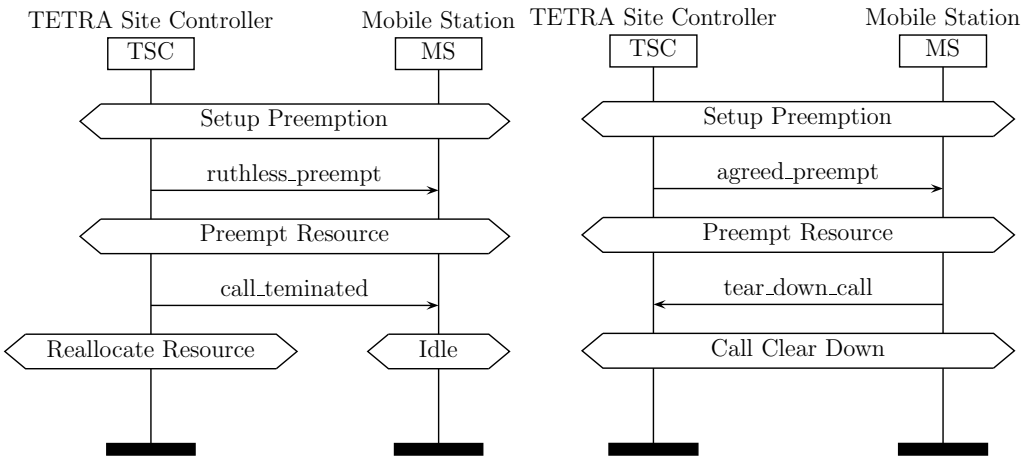

Fig. 4. Two Preempt Priority Call Scenarios

feature for the TETRA standard. This feature allows a high priority call to preempt allocated channel resource from a lower priority call when there is no other free resource. The channels allocated to the lower priority call can either be released through agreed call tear down or through ruthless resource preemption. Which occurs depends on the priority of the call. An emergency call will always result in ruthless preemption. Assume for the moment that the phases in the two scenario are explicit states in automata representations of the scenario processes. It is then a straightforward exercise to construct finite state automata from these scenarios describing the concurrent behaviour of the processes. For brevity we will replace phase names and message names by their initials. Hence, 'Setup Preemption' will become the state 'SP' in the automata, 'Call Clear Down' becomes 'CD', 'ruthless_preempt' becomes 'rp', 'tear_down_call' becomes 'tdc' and so on. Figure 5 gives the automata derived from the two MSC in figure 4. These two automata running concurrently can deadlock. This can occur when they have both reached state 'PR' and the TSC automaton sends '!ct' at the same time that the MS automaton sends '!tdc'.

Such a deadlock would not occur if the phases in the scenarios were not identified with explicit states in an automaton. Phase automata representations of the processes constructed with the phase semantics outlined in section 2 do not deadlock in this way.

Figure 6 defines the phase automata derived from figure 4 using the informal phase semantics defined in section 2. Each phase in the scenarios defines a set of states. In the illustration each phase is depicted as a box surrounding the states that are contained within that phase.

Since the two scenarios of figure 4 start with the same phase transitions, each scenario must apply whenever the other applies. Thus there must be a state within the 'SP' (Setup Preempt) set of states from which we can observe both

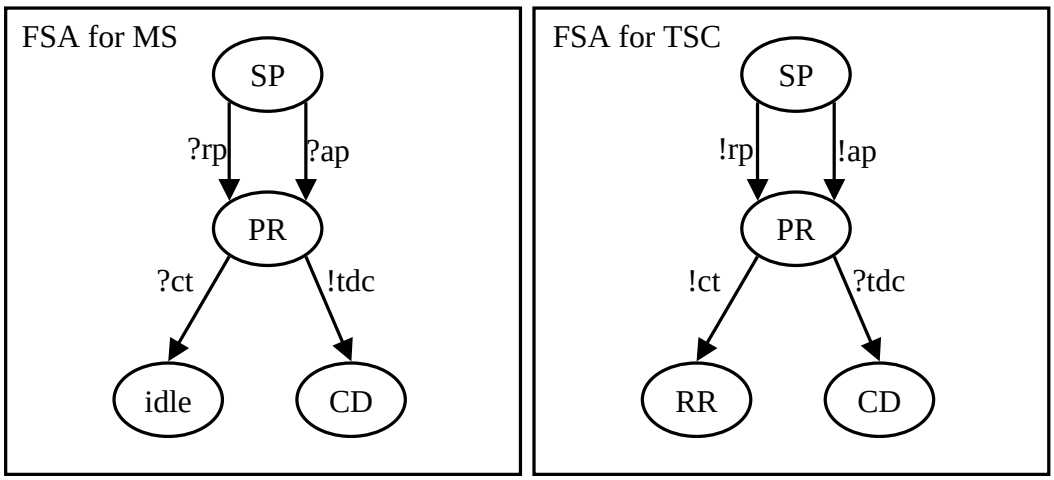

Fig. 5. FSA Automata

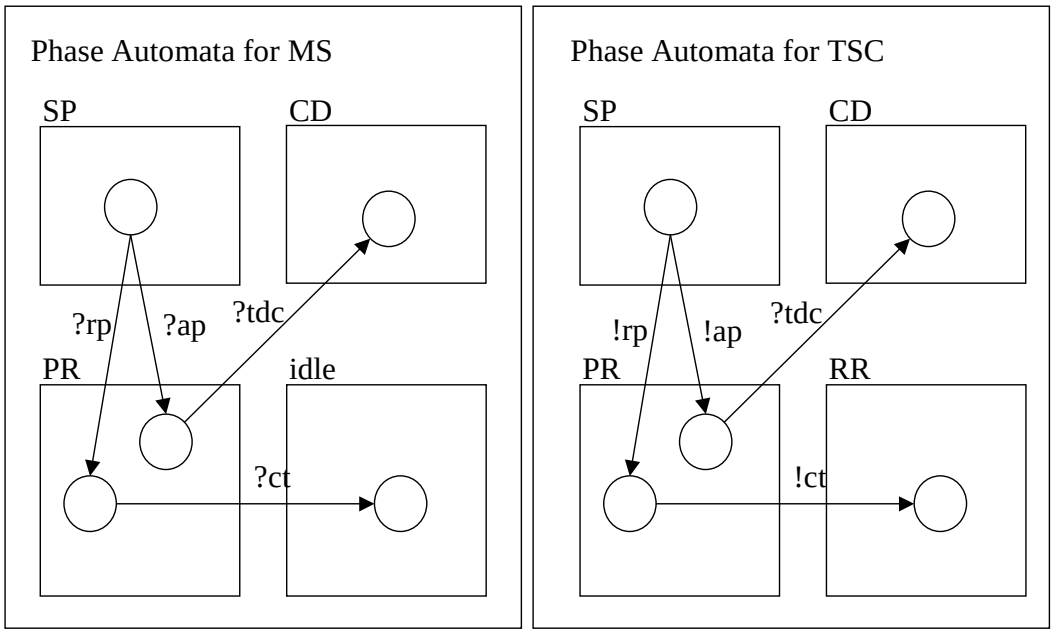

Fig. 6. Phase Automata for MS and TSC

scenarios unfolding. However there are no overlaps between the phase traces of the two scenarios in figure 4, unlike the examples of section 2. Hence no other states within the phase automata can be joined. This leads to the traces within the phase automata being disjoint after the initial state. Because of this the two automata do not deadlock when run concurrently. Either explicit state or phase semantics can be used within the appropriate context. Phase semantics are appropriate when the scenarios are more informal, even though they are defined within a formal notation. Phase semantics allow scenarios to define weaker constraints than explicit state semantics, whilst still constraining the composition of phase transitions. This provides more flexibility for developers during the early stages of requirements capture when they do not want to

unduly constrain design decisions for the software development teams. Note that in general the set of traces from the phase automata are a subset of the traces defined by the explicit state automata. So that if, at a later date, the phases are used to define states in a more detailed model, the traces from the phase automaton will still be valid in the new model.

4 Phase Semantics

This section will define the formal phase semantics for MSC scenarios, and how to synthesise a phase automata from a set of phase traces.

Let the set of symbols E denote the set of events that can occur in the specifications. Let P be the set of phases that can occur in the specifications. Let S be a set of states that will be used to construct phase automata, and let $\phi : S \longrightarrow P$ be the phase function. This defines which phase a state belongs to. Define a set of deterministic transitions to be a partial function $\partial : S \times E \longrightarrow S$. The tuple $\mathsf{A} = (S, E, \partial, \phi)$ is defined to be a *phase automaton*. A *phase trace* is an alternating sequence of phases and events S_i, e_i, terminated at both ends by a phase. An execution trace of A is a phase trace

$$S_0 \cdot e_0 \cdot S_1 \cdots S_{n-1} \cdot e_{n-1} \cdot S_n$$

where there are states x_i for $0 \leq i \leq n+1$ such that $\phi(x_i) = S_i$ and $\partial(x_i, e_i) = x_{i+1}$.

Each MSC defines a set of event traces as defined in the standard. These represent all possible observable interleavings that can occur for the events in the scenario. The possible interleavings are given by the total order extensions of the various partial order causal relationships that the message exchanges define. We can extract the event traces for each process from the MSC event traces by simply deleting those events from processes we are not concerned with. For the examples of this paper the time lines of each process are always continuous, so that the event traces for each process are very simple. When the time lines are broken up with co-regions or references to the concurrent composition of other scenarios it becomes more complex to generate the process event traces [3].

Each MSC scenario therefore defines a set of event traces for each process in the scenario. A specification phase trace of process P is derived from an event trace of P from one of the requirement MSC scenarios, by replacing event e in the trace with a triple (S_0, e, S_1). Where S_0 is the phase that is active immediately prior to e in P, and S_1 is the phase that is active immediately after e. Recall that the MSC condition symbols define the current phase in

a scenario. An event e is active within phase S if that is the closest phase symbol that precedes e on the same time line. The current phase within an MSC trace changes after an event e if and only if the following symbol on the time line after e is a condition symbol denoting a different phase. The triple (S_0, e, S_1) is called an *annotated event*. When $S_0 \neq S_1$, (S_0, e, S_1) is called a *phase transition event*.

Define $\underline{(S, e, S')} = S \cdot e$, and $\overline{(S, e, S')} = S'$. Given a specification phase trace $t = \alpha_0 \cdots \alpha_n$, where each $\alpha_i = (S_i, e_i, S_{i+1})$, define the *intrinsic phase trace* of t to be

$$t^* = \underline{\alpha_0} \cdot \underline{\alpha_1} \cdots \underline{\alpha_n} \cdot \overline{\alpha_n}$$

This notation simply defines a compact form for representing a specification phase trace. Note that consecutive events in t are (S_i, e_i, S_{i+1}), $(S_{i+1}, e_{i+1}, S_{i+2})$. So that t can always be reconstructed from t^*.

A specification phase trace t is an *execution phase trace* of A when t^* is an execution trace of A. If there are states x_i for $0 \leq i \leq n{+}1$ such that $\phi(x_i) = S_i$ and $\partial(x_i, e_i) = x_{i+1}$, x_0 is defined to be a *start state* of t in A, and x_{n+1} is the *accepting state* for t. The states x_i are called the *execution states* for t.

Given a set of specification phase traces for a process we want to define a phase automata that generates these phase traces, and also generate those phase traces that are given by the permissible overlapping of the MCS scenarios. The informal semantics defined in section 2 can now be formally stated.

Suppose we can write t as a concatenation $t_1 \cdot t_2$ where the final annotated event of t_1 is a phase transition. Then t_1 is defined to be a *phase precursor* of t.

Let A be a phase automata and t a specification phase trace as above. A *models* the phase transitions of t if for every phase precursor t_1 of t as above, whenever t_1 is an execution trace of A with accepting state x, then t_2 is also an execution trace of A with start state x.

A phase automata A is the *phase semantic representation* of a set of specification phase traces if, firstly all of these traces are execution traces of the automaton, and secondly the automaton models the phase transitions of each specification phase trace.

This section outlines how to construct a phase semantic representation A for a set of specification phase traces. The suggested technique is very inefficient, it is intended for reference only. In this section we assume that the representation is intended to be a deterministic finite state automaton.

- For each phase trace t proceed as follows.
  - For each phase precursor t_1 of t we add states and transitions to A as follows. Let $t = t_1 \cdot t_2$. For each state x, test if there is a state x' such that x is a start state for t_1, and x' is an accepting state for t_1. If so add states and transitions (if necessary) to ensure that t_2 can be generated from x'.
  - If t is not an execution traces of A then add states and transitions to A as follows. Suppose t^* is of the form:

    $$S_0 \cdot e_0 \cdot S_1 \cdot e_1 \cdots e_n \cdot S_{n+1}$$

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

    Note when this case occurs it must be that none of the phase precursors of t were execution traces of A.
  - 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.

Motorola UK Research Labs have built a prototype tool for constructing a phase semantic representation of a set of specification phase traces. This tool incorporates various patented technologies that allows the tool to construct phase automata in an optimal manner [4], [5]. The tool complements an existing Motorola UK Research tool [3] that automatically generates conformance test suits from MSC specifications. The tool can statically analyse phase automata to detect certain elementary types of feature interactions. This has been validated against a selection of randomly chosen examples from the interaction benchmark paper [7], plus a number of examples for 3G requirements scenarios. It is currently being used as part of a pilot study to analyse requirements specifications for a new Motorola product.

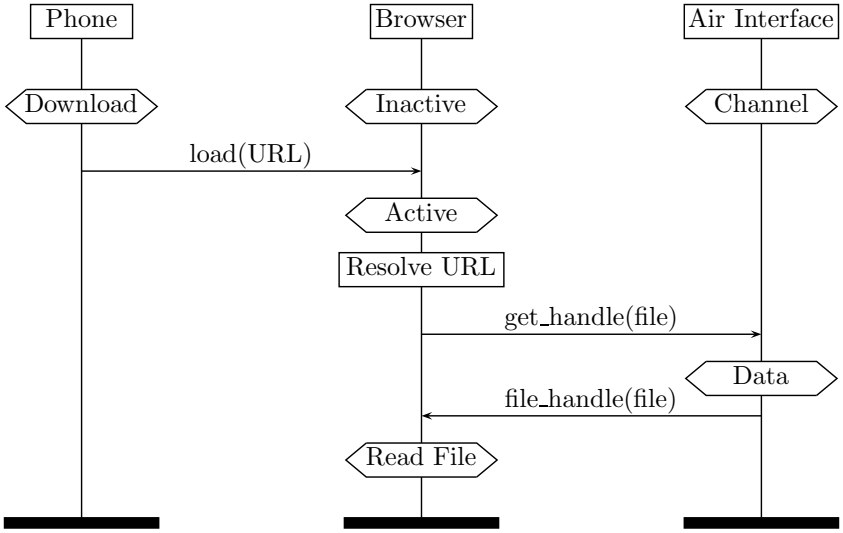

Fig. 7. Initial section of browser download scenario

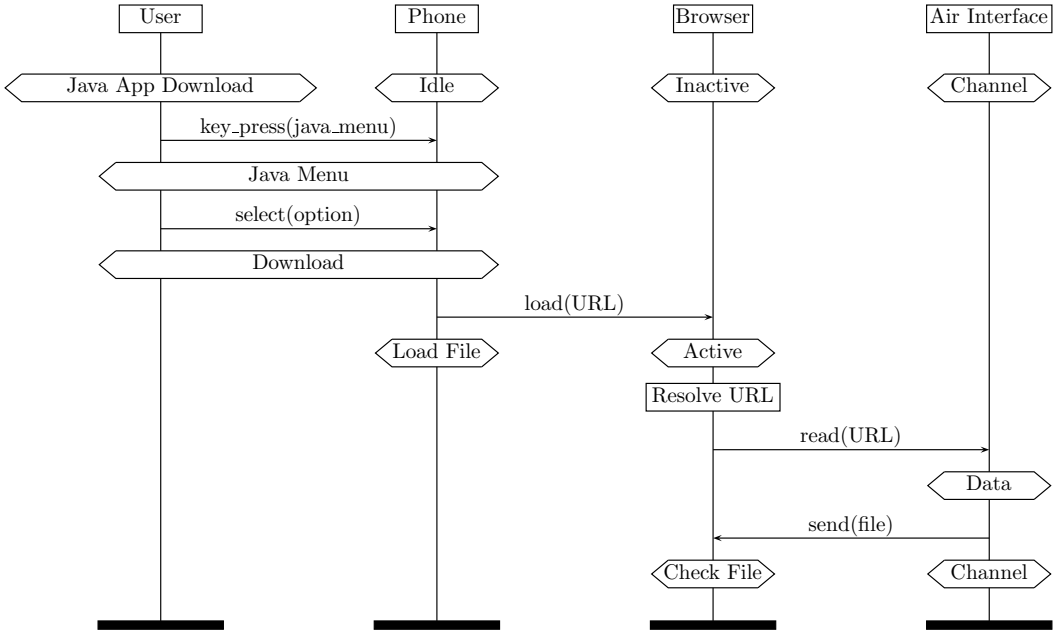

Fig. 8. Initial section of java application download scenario

5 Phase Semantic Representation for Browser Download Examples

This section gives an example of the semantic representation for two MSC scenarios. These are derived from figures 1, and 2. We consider just the initial parts of these two scenarios which are relevant to discovering an interesting overlap between them. These are given by figures 7 and 8.

We will consider the construction for a semantic representation of the 'Browser' process. We have constructed these MSC so that each scenario generates only

a single phase trace for each process. In general of course this does not always hold. The intrinsic phase traces for the 'Browser' process are:

$$
\begin{aligned}
(1)\ \tau_1 = \ & \mathsf{Inactive} \cdot \mathsf{?load(URL)} \cdot \\
& \mathsf{Active} \cdot \mathsf{Resolve\ URL} \cdot \\
& \mathsf{Active} \cdot \mathsf{!get\_handle(file)} \cdot \\
& \mathsf{Active} \cdot \mathsf{?file\_handle(file)} \cdot \\
& \mathsf{Read\ File} \\
(2)\ \tau_2 = \ & \mathsf{Inactive} \cdot \mathsf{?load(URL)} \cdot \\
& \mathsf{Active} \cdot \mathsf{Resolve\ URL} \cdot \\
& \mathsf{Active} \cdot \mathsf{!read(URL)} \cdot \\
& \mathsf{Active} \cdot \mathsf{?send(file)} \cdot \\
& \mathsf{Check\ File}
\end{aligned}
$$

Notice that both τ_1 and τ_2 have only one phase precursor which is the same for each phase trace:

$$\tau = \mathsf{Inactive} \cdot \mathsf{!load(URL)} \cdot \mathsf{Active}$$

This is a slightly incorrect statement. Strictly speaking τ_1 is a precursor of itself since the last annotated event is a phase transition, which is also true for τ_2. However these precursors do not affect the semantic representation since we are forced to make each complete semantic trace an execution trace in any case.

Consider τ_1 first. A semantic representation of this trace on its own is given by figure 9. In this case the precursor τ can only be generated from state x in the Inactive phase, so the phase automata consists of a single phase trace, where v is the start state for τ_1 and z is the accepting state for τ_1. τ is also the precursor of τ_2. Let $\tau_2 = \tau \cdot \tau_3$. Since τ is an execution trace of A_0 with accepting state w, it must be the case the w is the start state for τ_3 for any phase automata that represents both τ_1 and τ_2. That implies we need to add other states and transitions as shown in figure 10 in order to extend A_0 into an automata A_1 that represents both phase traces.

A_1 generates both phase traces and models the phase transitions of both of the phase traces in the specification. It is not however deterministic. To make it so requires the identification of states x and x'. The result is phase automata A_2 shown in figure 11. Phase automaton A_2 is a phase semantic representation for phase traces τ_1 and $\tau 2$. It is also the minimal such deterministic automaton.

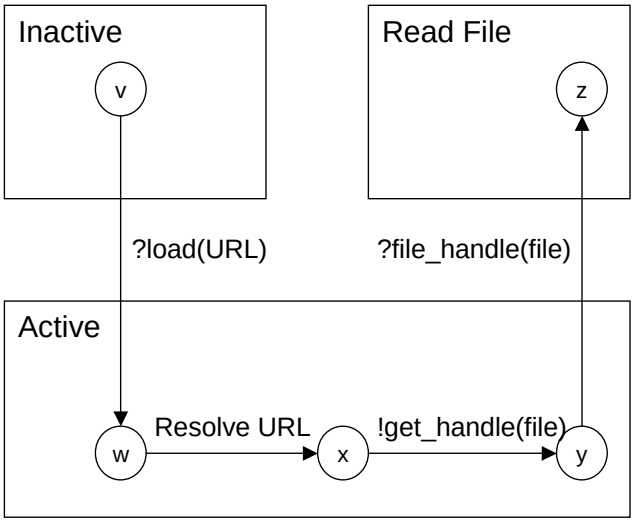

Fig. 9. Semantic representation A_0 for τ_1

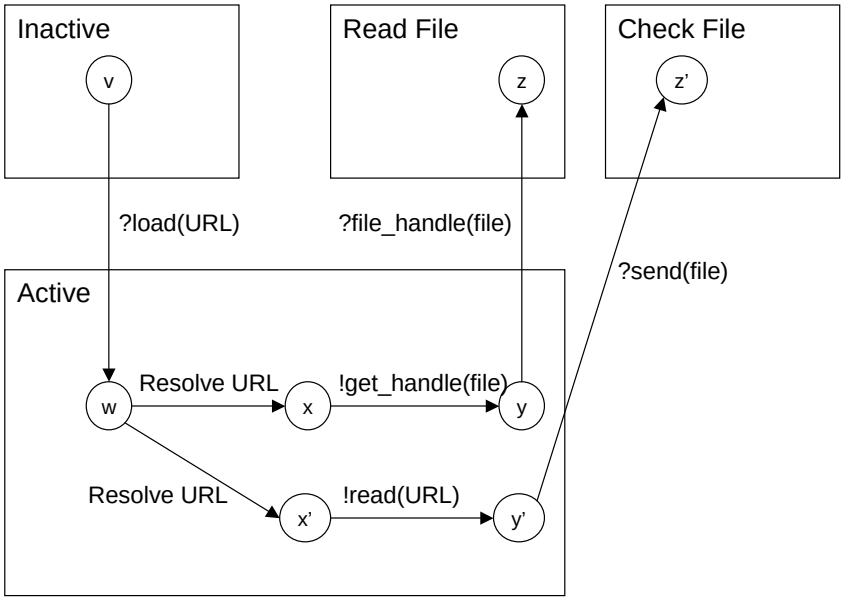

Fig. 10. Semantic representation A_1 for τ_1 and τ_2

6 Elementary Error Detection with Semantic Representations

It is possible to statically analyse phase automaton to detect certain simple types of elementary 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. Care must be exercised since, for example, searching for unreachable states is not appropriate for partial requirements.

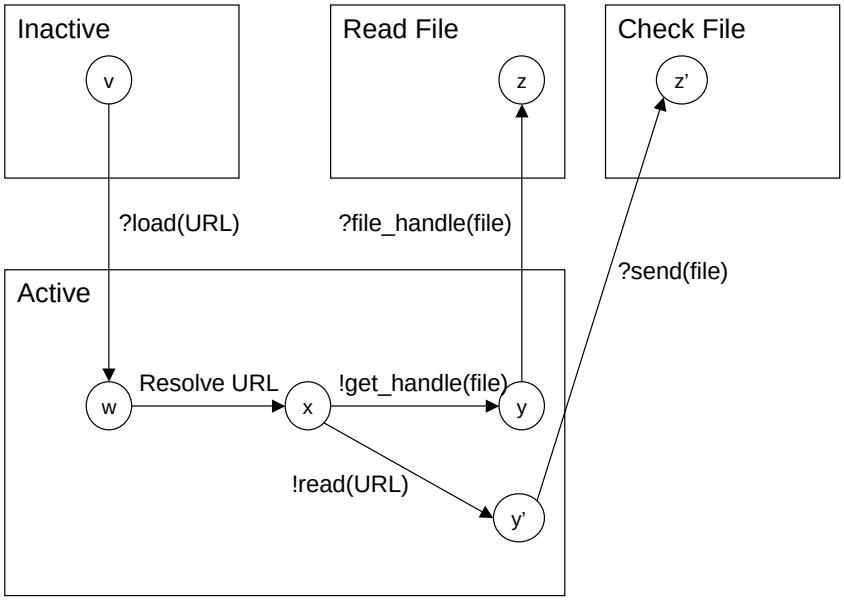

Fig. 11. Semantic representation A_2 for τ_1 and τ_2

There are two types of elementary errors that can be statically detected during the construction of the phase automaton, which avoids expensive dynamic detection. There is anecdotal evidence to suggest that these kinds of inconsistency can account for a significant proportion of feature interactions. These two types of error are the most fundamental that can be detected. Since they are caused by flaws in the phase automaton structure they can also be detected without any need for user input. These elementary structural errors can not be removed by adding new specification scenarios to the requirements.

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.

$$S0,\ ?u,\ S0,\ !a,\ S1$$
$$S0,\ !a,\ S2$$

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. This is generally an error, since it represents a particular phase trace that causes an ambiguous phase transition.

Structural inconsistencies

When a specification process is also a system component, it is often the case that it is constrained to always take the same action for a given sequence of events. That is for a given phase precursor that triggers a component to send a message event, that event should be unique. Consider a state x and transitions

$$x \xrightarrow{!a} x_1$$
$$x \xrightarrow{!b} x_2$$

where a and b are distinct. A system component will not know whether it should send a or send b. For example assuming 'Browser' is meant to chose a response deterministically for each set of inputs, then state x in A_2 should not have a nondeterministic choice of outputs. Hence this would represent an error.

The Motorola prototype performs exactly these two types of error detection at present.

6.1 Semantic Representation equivalence

The phase semantics of a set of specification phase traces (defined with MSCs for example) do not uniquely define the phase automaton that implements them. This section briefly describes the equivalence class of phase automata that are generated by a given set of specifications.

For a phase automata $\mathsf{A} = (S, E, \partial, \phi)$, that is the semantic representation of a set of specification phase traces T, we can define a standard finite state automaton $X(\mathsf{A})$ that define the specification phase traces generated by A. Define the states of $X(\mathsf{A})$ to be the same as those of A. The events of $X(\mathsf{A})$ are annotated events of A. The transitions of $X(\mathsf{A})$ are

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

where $x_i \xrightarrow{e_i} x_{i+1}$ is a transition in A, $\phi(x_i) = S_i$, and $\phi(x_{i+1}) = S_{i+1}$. A start state of $X(\mathsf{A})$ is any start state for a specification phase trace of A, and similarly, an accepting state of $X(\mathsf{A})$ is any accepting state for a specification phase trace of A. All that we have done to define $X(\mathsf{A})$ is to add the phase information from ϕ explicitly on to the transitions from A. We may regard any finite state automaton as a process that can be described by a process

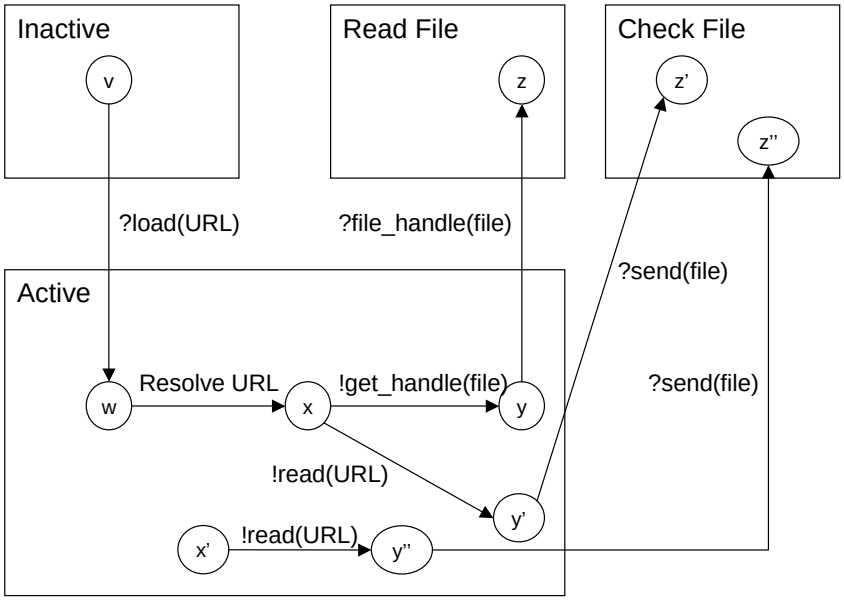

Fig. 12. Semantic representation A_3

algebra such as CCS [13] or CSP [9]. Recall the simulation relation $\sqsubset$ between processes can be defined as:

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

Given two semantic representations A_1 and A_2 for the same set of phase traces, they are *simulation equivalent* in the sense that:

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

The main motivation for considering phase automaton is to permit the detection of feature interactions that are caused by phase transitions. The types of elementary errors that are defined in section 6 are invariant with respect to simulation equivalence. Note this equivalence is not bisimulation equivalence (either weak or strong). To illustrate this point, add another specification phase trace τ_4 to the pair that define A_2. Let

$$\tau_4 = \mathsf{Active} \cdot \mathsf{!read(URL)} \cdot \mathsf{Active} \cdot \mathsf{?send(file)} \cdot \mathsf{Check\ File}$$

Phase automaton A_2 is a semantic representation of τ_1, τ_2, and τ_4. However so is phase automaton A_3 in figure 12. $X(\mathsf{A}_3)$ is simulation equivalent to $X(\mathsf{A}_2)$, but not bisimulation equivalent. Note that if we minimise $X(\mathsf{A}_3)$ we obtain $X(\mathsf{A}_2)$. It is conjectured that the minimal representation for $X(\mathsf{A})$ is unique up to strong bisimulation.

7 Conclusion

The MSC notation is an international standard defined by the ITU and recommended by ETSI as a notation for defining requirements specification scenarios. In practise requirements specifications in the telecommunications industry are often made up of MSC scenarios and little else.

MSC scenarios are a very effective way of describing major phase transitions within a protocol. The general property checking problem for recursive MSCs is undecidable, and even for restricted MSC notations the model checking problem is intractable. This paper has defined a formal semantics that permit tractable phase automata to be constructed that define the implicit phase transitions contained within the requirements specifications. These can be statically analysed without user input to detect certain elementary forms of undesirable interactions that are invariant under additions to the requirements specifications, and therefore represent significant interactions within the specification.

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

[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, E. Magil, Feature Interaction in Telecommunications and Software Systems VI, IOS, 2000.

[7] 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

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

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

[10] 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.

[11] 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`

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

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

[14] Bill Mitchell, Robert Thomson, Clive Jervis, Phase Automaton for Requirements Scenarios, Feature Interactions in Telecommunications and Software Systems VII, 77-84, 2003, IOS Press.

[15] Johann Schumann, Jon Whittle, Generating Statechart Designs From Scenarios, Proceedings of the 22nd international conference on on Software engineering, 2000.

[16] Sebastian Uchitel, Jeff Kramer, Jeff Magee, Synthesis of Behavioral Models from Scenarios, IEEE Transactions on Software Engineering, vol. 29, no. 2, February 2003

[17] Z.120 (11/99)ITU-T Recommendation - Message Sequence Chart (MSC)

[18] Z.100 (11/99) ITU-T Recommendation - Languages for telecommunications applications - Specification and description language

[19] 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.