Resolving Race Conditions in Asynchronous Partial Order Scenarios

Bill Mitchell

Department of Computing, University of Surrey

Guildford, Surrey GU2 7XH, UK

w.mitchell@surrey.ac.uk

This is a draft version, the final version appeared in IEEE Transactions on Software Engineering, Vol 31, No 9, pp 767- 784, TSE-0039-0205, 2005. ISSN 0098-5589 DOI: 10.1109/TSE.2005.104

Abstract

Scenario based requirements specifications are the industry norm for defining communication protocols. However, such scenarios often contain race conditions. A race condition occurs when events are specified to occur in a particular order, but in practice, this order cannot be guaranteed. The paper considers UML/MSC scenarios that can be described with standard partial order theoretic asynchronous behavioural semantics. We define these to be partial order scenarios. The paper proves there is a unique minimal generalization of a partial order scenario that is race free. The paper also proves there is a unique minimal race free refinement of the behavioural semantics of a partial order scenario. Unlike the generalization, the refinement cannot be realized in the form of a partial order scenario, although it can always be embedded in one. The paper also proves the results can be generalised to a sub-class of iterative scenarios.

I. INTRODUCTION

Practitioners commonly specify asynchronous communication protocols for distributed systems as a collection of scenarios. UML sequence diagrams [32], Message Sequence Charts (MSCs) [31], and Live Sequence Charts (LSCs) [15] are popular for defining such scenarios. Such languages are intuitive and simple to use. Unfortunately, such languages can also beguile the overworked practitioner struggling with product deadlines.

In general, requirements specifications have been shown to be a significant source of defects. Published case studies, [11], [19], [20], [24], [33], have shown that about a third of significant behavioural defects can be traced to requirements specifications. This situation is no different for distributed communication systems that use sequence diagrams to define behavioural aspects of the system.

Scenario specification languages have become quite sophisticated and expressive. Despite this sophistication, basic scenarios are still the mainstay of industrial specifications. A basic scenario diagram is one whose behavioural semantics can be defined in terms of a partial order (Definition 2.1) on the events in the scenario. This is the semantics defined in the MSC standard [31] and now adopted as the core semantics for UML sequence diagrams [32]. The partial order restricts the order in which events can occur in any system trace. This partial order is called the causal ordering, (Definition 2.4). The intuition is that the partial order is specifying the causality between events in a scenario. We refer to basic scenario diagrams as partial order scenarios to

emphasise this point, (Definition 2.3). It is possible to define other behavioural semantics for sequence diagrams, for example by constraining behaviour with UML architecture diagrams. This is beyond the scope of this paper and we only consider partial order scenarios.

A causal order not only specifies that certain events must be ordered in a particular way, but also that certain events are independent of one another. This can be the case even for events within the same process. Therefore, a partial order scenario specifies separate concurrent threads of activity and at what points these threads must be synchronized.

Causal orders can specify an almost arbitrary partial ordering on events within a process. However, they may only order events from different processes as a result of message passing between those processes. Hence, an arbitrary partial order over the set of events in a scenario does not necessarily itself correspond to a partial order scenario.

There are varieties of errors that can creep into scenario based specifications. The purpose of a protocol scenario is to define message exchanges between processes in order for them to achieve some common goal. Messages can easily be added between the wrong processes, accidentally added in the wrong direction, or just missed altogether. This type of static error is generally picked up by routine document inspections or through automated checks by software tools, such as with Telelogic's TAU G2 tool [27], and as such do not require significant effort to resolve. However, behavioural inconsistencies are more difficult to detect and resolve. In this paper we focus on one common behavioural inconsistency known as a race condition. Essentially a race condition asserts a particular order of events will occur because of the causal ordering, when in practice this order cannot be guaranteed to occur. Because the standard partial order semantics places almost no constraint on how the causal order is constructed, it is very easy to inadvertently introduce race conditions into a scenario. [12], [13] give the original formal description of race conditions within the MSC context.

Scenarios are really best suited for describing exemplary behaviour rather than specifying all possible concurrent behaviour in a distributed system. However, practitioners find them more intuitive and 'easier' to understand than say state machines, [29]. The standard partial order semantics does not attempt to distinguish between exemplary and mandatory behaviour, or what can be implemented in practice. LSCs were introduced in an attempt to address these issues, and UML 2.0 now includes many constructs that permit greater expressivity in relation to message flows. Despite these additional constructs the fundamental issue still remains: what consistent

asynchronous behaviour can be extrapolated from a set of scenarios. Many protocols start life as partial order scenarios. It is therefore useful to address the issue at this level. Consistent scenarios that are subsequently constructed can then be enhanced with appropriate constructs to describe how they should be used in the specification as a whole.

It is possible to directly analyse the causal ordering to automatically detect race conditions [13]. It is also possible to automatically construct a system trace that describes how a race occurs. This still leaves the onerous task of deciding what behaviour is possible in practice and actually correcting the specifications. Complications arise when a scenario contains multiple interrelated races. Attempting to fix races piecemeal by examining individual error traces can be frustrating when the races have a common cause that should be resolved directly. Section X describes two such examples from a proprietary industrial case study.

Main Results of the Paper

In the paper, we prove there is a unique minimal generalization of a partial order scenario that is race free up to simulation equivalence (Theorem 6.2). We refer to this scenario as the inherent causal scenario. Note that although the inherent causal scenario is unique, its graphical depiction is not. Within UML and MSC, it is possible to depict the same behaviour in several ways due to the many constructs that now exist in these languages. We also prove there is a unique minimal race free refinement of the behavioural semantics for a partial order scenario (Theorem 8.6). We refer to this as the inherent refinement ordering. This refinement cannot be directly realized as a partial order scenario. However, messages can be added to the original scenario to realize the behavioural refinement as a partial order scenario (Lemma 8.3), whereas the generalization can be achieved by introducing greater concurrency, which does not require any additional messages. Race conditions are studied within a partial order theoretic framework. Uniqueness results are first derived purely in terms of partial orders. We then construct a partial order scenario realisation of each partial order that characterises a uniqueness result. This is necessary since an event partial order alone does not necessarily correspond to a partial order scenario.

The paper also considers how race conditions can occur in iterative scenarios. These extend partial order scenarios by adding a loop construct. Loops can be nested, and we allow iterative scenarios to be weakly composed. Race conditions can still occur within a partial order scenario

as before (which may or may not be part of an iterative scenario). However, they can also occur because of loop constructs, which only become evident when the loops are unwound. We will refer to the later type of race as an iterative race.

The paper defines a particular property of partial order scenarios that we refer to as the convergence property (Definition 9.8). The paper proves that as long as an iterative scenario is constructed from race free partial order scenarios by means of the loop construct, then it is free of iterative races if and only if each of its composite partial order scenarios satisfies the convergence property (Theorem 9.14). From this it is possible to give a constructive algorithm for adding finitely many messages to an iterative scenario to remove all iterative races and in such a way that the message flows within loops are not significantly changed. Hence, the problem of resolving races in iterative scenarios can be reduced to that of resolving them in partial order scenarios.

Related Work

The study of partial order scenarios has been an active topic of research for many years. [28] contains an excellent survey of related work in this area, which we strongly recommend to the reader. [25] is another excellent survey covering verification of protocols specified with MSC scenarios. In light of these, we will not attempt to give a complete list of all related work here.

The problem addressed in this paper is focused on one particular aspect of message flows within a sequence diagram. There is of course a wide variety of verification issues connected with protocol design. Verification of logical properties from MSCs and MSC-Graphs has been considered by [1], [14], [30], amongst others. This work addresses issues such as deadlock, livelock and ensuring correct responses to requests. Much of this work has been based on synthesising finite state automata for the purposes of model checking. [7], [8], [18], [23], [30] consider how to construct state machines directly from a collection of message sequence charts via different kinds of compositional semantics. Other work has considered how to interpolate missing requirements from scenario based specifications [2], [3], [4], [21], [28]. This work is useful both in verifying a system and in synthesising a more complete specification. [3] is the seminal work that first considered the realizability of collections of MSCs. In their work, they consider when MSCs composed via an MSC-Graph can collectively be implemented. The issue they address is orthogonal to that considered here, as they implicitly regard race conditions as

benign in their examples. Research into automatic test generation from partial order scenarios is another active research area [5], [6], [9], [26].

An interesting variation on mainstream MSC/UML scenarios is given by live sequence charts (LSCs) [15]. As mentioned in the introduction, these permit more expressitivity by permitting exemplary and mandatory behaviour to be annotated directly within a scenario. It is also possible to synthesise state machines from LSCs [16]. Although both UML and MSC scenarios are widespread in industry, LSCs have not yet achieved a large following outside of the academic community.

From our search of the literature it appears that our paper is novel in that it proves there exists a unique optimal solution to a specific class of semantic inconsistencies for sequence diagrams, namely race conditions. It also appears to be novel in characterising the solution purely in partial order theoretic terms.

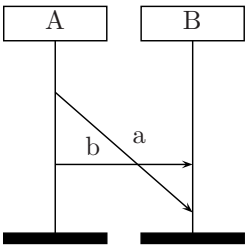
Graphical Notation

Although our results apply to any partial order scenarios, we will use MSC as the graphical language for the paper. The MSC standard [31] is stable and MSCs are common in industry, also their semantics are targeted at asynchronous protocols. For the type of elementary scenario we describe here, the notation is almost identical for UML sequence diagrams, LSCs and MSCs. In this section, we give a terse and intuitive explanation of the constructs we will use. For formal details the interested reader is referred to the standards [31] and [32].

Consider the MSC depicted graphically in Figure 2. Each vertical line describes the time-line for a process, where time increases down the page. Messages are depicted by arrows. Each message m defines a pair of events $(!m, ?m)$, where $!m$ is the send event for m, and $?m$ is the receive event for m. Messages in MSCs are always asynchronous since we are dealing only with asynchronous protocols.

The distance between two events on a time-line does not represent any literal measurement of time; only that non-zero time has passed. Events on the same time-line are ordered linearly down the page, except where they occur within a coregion. Within a coregion, events are not locally ordered unless that is directly imposed by a general order construct (described below). Coregions are depicted with a dashed line. For process B in Figure 2 events $?c$ and $?d$ are unordered as they occur within a coregion.

Fig. 1. Message Overtaking



It is possible with MSC and UML notation to describe message overtaking, as shown in Figure 1. As is the norm, we rule out message overtaking in specifications. That does not mean we attempt to forbid message overtaking in practice, only that it may not be specified as having to occur as part of the protocol definition.

The MSC and UML standards include a general ordering construct, which is a simple graphical notation that explicitly forces one event to occur before another event. A general order construct is depicted as a dashed arrow between the events to be ordered, with arrowhead placed in the middle of the arrow. For example, the MSC in Figure 12 contains a general ordering construct between the $?b$ and $!c$ events. Where general ordering constructs span processes in an MSC, as in Figure 12, the MSC is not strictly a partial order scenario. Events in different processes may only be ordered by virtue of inter-process communication, as explained in definition 2.4. We will use the general order construct in simple MSCs to illustrate scenarios, but we point out when the MSC is not a partial order scenario.

A parallel construct in a MSC/UML sequence diagram, denoted by keyword `PAR`, describes a set of concurrent threads that occur in the diagram. Dotted lines delineate the different threads. Hence, events from one thread are not causally ordered with respect to events from any other thread. Figure 20 contains two parallel constructs. The first parallel construct contains messages a, b and c in separate threads, which can therefore occur in any order. The bounding box of a parallel construct has no effect on the ordering of events, it solely delineates the scope of the concurrent threads. For example, receive event $?a0$ is ordered before each of $?a$, $?b$ and $?c$, even though these events are not ordered with respect of one another since they are in separate threads. Events within a particular thread are ordered in the usual way, so for example, $!c1$ must be before $!c2$ in the second parallel construct.

An inline reference, denoted by keyword `REF`, is a placeholder for another sequence diagram. The reference can be replaced by the contents of the other sequence diagram if desired. The

reference is weakly composed with the referring diagram when inlined. Figure 20 contains an inline reference spanning processes A through D.

MSC and UML notations allow a message to be split into lost and found events. This allows a message to be sent in one scenario and received in another. The send part of the message is represented by a lost event, and the receive part by a found event. In this paper, we will only use this construct to simplify the visual layout of scenarios. Hence, the found event corresponding to a lost event always occurs in the same scenario. Figure 11 gives an example where messages a and c have been split into lost and found events. The lost event for $!a$ is depicted by the solid circle terminating the arrow mid-process. The found event for $?a$ is depicted by the empty circle that initiates an arrow into process B. In this paper, we follow the convention that the found event given by a message must always occur after the lost event for the message. The MSC and UML standards do not make any formal link between a lost event and a found event; that is they do not identify them as component parts of a single message. However, our convention is only used in simplifying the visual layout of our examples and does not affect the theoretical results in any way.

II. Basic Partial Order Specifications

In this section, we define the causal ordering semantics for partial order scenarios. We use the same message semantics as the MSC 2000 standard [31]. Hence, a partial order scenario defines a set of message exchanges between processes with asynchronous communication channels.

Definition 2.1:

- *A partial order over a set E is a binary relation $<$ such that*
  - *$<$ is irreflexive, $\neg(x < x)$ for any $x \in E$*
  - *$<$ is transitive, $x < y$ and $y < z$ implies $x < z$*
  - *$<$ is asymmetric, there are no elements $x, y \in E$ such that $x < y$ and $y < x$*
- *We write $\neg(x < y)$ to denote that it is not the case that $x < y$.*
- *Two elements x and y of E are unordered if $\neg(x < y)$ and $\neg(y < x)$.*
  When this is the case we write $x\ [<]_{\mathsf{Un}}\ y$. We define a set to be unordered if every pair of distinct elements from that set are unordered.

Often in the literature, this type of partial order is referred to as a strict partial order. Virtually all the partial orders considered in this paper are strict so we adopt the convention of taking

partial orders as strict unless otherwise stated. A non-strict partial order in the usual sense is antisymmetric rather than asymmetric. The antisymmetric condition states that if $x < y$ and $y < x$ then $x = y$.

Definition 2.2:

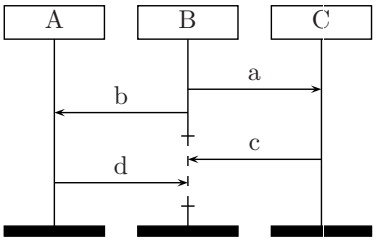
- *A total order over the set E is a partial order on E where for any two distinct elements a and b, either $a < b$ or $b < a$.*
- *A total extension of a partial order $<$ is any total order $<_1$ such that $x < y$ implies $x <_1 y$.*
- *The trace of a total order $<_1$ on the set E is the unique sequence of the form*

$$e_0 \cdot e_1 \cdot \cdots \cdot e_n$$

where $E = \{e_i \mid 1 \leq i \leq n\}$, and $e_i <_1 e_{i+1}$ for each i.

- *A trace of the partial order $<$ is the trace of any total extension of $<$.*

Fig. 2. Race Hidden by Coregion



Let $\mathcal{P}$ be a set of processes. A message m between processes is a pair $(!m, ?m)$ where $!m$ is the send event for m, and $?m$ is the receive event for m. We regard $!m$ as belonging to the sending process, and $?m$ as belonging to the receiving process. Let E be the set of all send and receive events between all processes. Each event has a label, let $l : E \longrightarrow L$ be the labelling function. For a message m, $l(!m) = l(?m)$. Within the MSC standard, there are many other kinds of events such as action boxes and condition symbols, but here we only consider message events to simplify proofs as much as possible. It is possible to generalize the results to include these other events.

Definition 2.3: A partial order scenario M on processes $\mathcal{P}$ is

- *a collection of disjoint sets $E(P) \subseteq E$, for each $P \in \mathcal{P}$ that defines the message events belonging to P,*

- *and a set of partial orders $<_P$, where $<_P$ is a partial order on $E(P)$ that defines the local ordering of events for process P.*

These local partial orders must be subject to the constraint that for each send event $!m$ in a set $E(P)$ the corresponding receive event $?m$ occurs in some set $E(Q)$. Note messages are allowed to be sent from a process to itself, so we allow $P = Q$. We treat a partial order as a binary relation that can be represented as the set of pairs that are ordered by the relation. Hence, we can take the union of partial orders, which is just the set theoretic union of the sets that represent the relevant order relations. It is important to note the local orders are not necessarily total, but can be any partial order. In the literature, it is sometimes assumed basic scenario diagrams have total local orderings, so it is worth emphasizing this does not have to be the case.

Let Msg be the set of messages defined as the set of send and receive event pairs:

$$\{(!e,\, ?e) \quad | \quad !e \in E(P) \text{ and } ?e \in E(Q) \text{ for some } P,\, Q \in \mathcal{P}\}$$

Definition 2.4: The causal ordering $<_{\mathcal{C}}$ on a partial order scenario is the transitive closure of the relation given by

$$\Big(\bigcup_{P \in \mathcal{P}} (<_P) \Big) \cup \mathsf{Msg}$$

Note all causal orderings are irreflexive, so that messages must be received after they are sent. The causal ordering defines the set of all possible system traces that are given by the partial order scenario. A system trace is any total order extension of $<_{\mathcal{C}}$. Recall a total order on a set S is a partial order $<$ on S where for any distinct elements $x,\, y \in S$, either $x < y$ or $y < x$.

Definition 2.5: The set of system traces defined by a causal ordering $<_{\mathcal{C}}$ is the set of traces for the causal order $<_{\mathcal{C}}$.

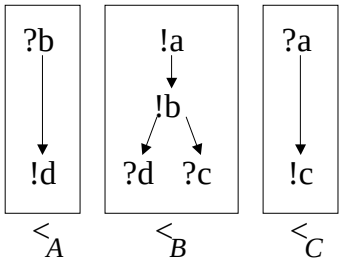
Definition 2.6: Let M be a partial order on events E_M, processes $\mathcal{P}_M$ and process orders $\{<_P^M |\ P \in \mathcal{P}_M\}$. Let N be a partial order on events E_N, processes $\mathcal{P}_N$ and process orders $\{<_P^N |\ P \in \mathcal{P}_N\}$. Scenario M is embedded in N if

- $\mathcal{P}_M \subseteq \mathcal{P}_N$
- $E_M \subseteq E_N$
- *for all $x,\, y \in E_M$ and processes $P \in \mathcal{P}_M$*

$$x <_P^M y \Leftrightarrow x <_P^N y$$

Consider the MSC depicted graphically in Figure 2. The local partial orders defined by this MSC are given in Figure 3 where we draw the ordering downwards, so that $!a <_B !b$ for example. In this case the causal ordering $<_{\mathcal{C}}$ is given in Figure 4.

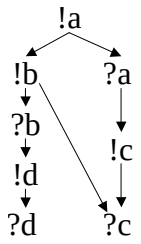
Fig. 3. Process Partial Orders



III. Race Conditions

Figure 2 illustrates a race condition. The causal ordering asserts that $!b <_{\mathcal{C}} ?c$. If this MSC is taken as a specification it asserts that after C receives a it must send c so that it arrives after b is sent. It is not possible for C to know for sure when $!b$ occurs without querying B. Hence it is quite possible if this scenario is implemented naively that c will arrive before b is sent, contradicting the specification. This error can occur even though each of the processes locally implements the specification correctly.

Fig. 4. Causal Ordering



Definition 3.1: A partial order $<$ on E is defined to be race free when for every event x and message e:

$$x <?e \Rightarrow (x <!e \text{ or } x =!e)$$

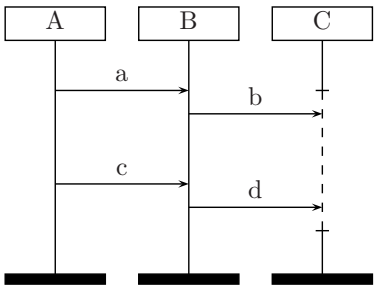
An MSC is defined to be race free when its causal ordering is race free.

That is $<$ is race free if the following holds. When $<$ orders an event x before the receive event of some message e, then it also orders x to be before the send event of e. Note that Figure 2 is not race free since $!b <_{\mathcal{C}} ?c$, but $!b \not<_{\mathcal{C}} !c$.

Figure 5 shows another typical kind of race condition. This is a greatly simplified version of a scenario from a Motorola case study. Two other examples from this case study are considered in detail in Section X. In this scenario A is transmitting messages to C via process B. The only function for B is to forward the messages. A race condition occurs here between $!b$ and $?c$. The race occurs since $!b <_{\mathcal{C}} ?c$, but $!b \not<_{\mathcal{C}} !c$.

In practice it may be that the latency of messages from A to B is large so that B has plenty of time to forward a before c is received. However, the semantics do not reflect this. As specified there is no mechanism for B to guarantee that b can be sent before c is received. Such implicit assumptions based on an existing system can cause problems later when networks are upgraded and the assumptions are no longer valid. Scenario based specifications are intended for protocol designs that are independent of low-level properties of the underlying network. One of the useful side effects of correcting race conditions is that it tends to focus the design at a higher level of abstraction.

Fig. 5. Message Forwarding Race Condition



IV. Partial Order Processes

In this section, we define a process algebra semantics of system traces. This is a standard result for partial orders, but we present it in a slightly non-standard format for ease of use later in the paper. The process algebra term characterizes the system behaviour up to simulation equivalence (this follows from Lemma 7.3). In Section V we describe the system behaviour as a finite state automaton whose states are the unordered subsets of E. This allows us to link the behavioural description here with earlier work of [1].

First we set up some notation for defining sets of events that are important in generating system and process traces.

Definition 4.1: Let $<$ be a partial order on a set of events E. For a set $S \subseteq E$ define

$$\begin{aligned}
\mathsf{n}(S,<) &= \{x \in E \mid \exists\, y \in S : y < x, \\
&\qquad \text{and } \neg\exists\, z \in E : y < z < x\} \\
\mathsf{min}(S,<) &= \{x \in S \mid \neg\exists\, y \in S : y < x\} \\
\mathsf{max}(S,<) &= \{x \in S \mid \neg\exists\, y \in S : x < y\} \\
\mathsf{cns}(a,S,<) &= \mathsf{min}((S - \{a\}) \cup \mathsf{n}(\{a\},<),<)
\end{aligned}$$

The set $\mathsf{n}(S,<)$ are those events that are a least upper bound for some element in S. The set $\mathsf{min}(S,<)$ is just the set of minimal elements of S.

Note that $\mathsf{cns}(a,S,<)$ is always an unordered set since the minimal elements of a set are themselves always unordered. The set $\mathsf{cns}(a,S,<)$ defines what events may be consecutive to a in a system trace, when S describes a set of events that are eligible to occur concurrently with a at a given point of the system execution. Suppose we have a system trace t that is a total extension of $<$. Let a be some event in t, so that t is of the form $t_0 \cdot a \cdot t_1$ (where $\cdot$ denotes concatenation). Let S be the set of minimal events from the set of all events not in $t_0 \cdot a$. Then t_1 must be of the form $b \cdot t_2$ where $b \in \mathsf{cns}(a,S,<)$. We can formally prove this result in the following proposition.

Lemma 4.2: Let $<$ be a partial order on E, $a \in E$ and $S = \{x \mid x\ [<]_{\mathsf{Un}}\ a\}$.

a) If there is a trace for $<$ of the form $t_0 \cdot a \cdot b \cdot t_1$ then b is an element of $\mathsf{cns}(a, S - B_a, <)$, where B_a is the set of events in t_0.

b) For any $b \in S$ there is some trace of the form $t_0 \cdot a \cdot b \cdot t_1$.

Proof

Proof of a)

Let t be a trace of the form $t_0 \cdot a \cdot b \cdot t_1$. We need to prove that $b \in \mathsf{cns}(a, S - B_a, <)$. Since a occurs before b in a trace of $<$ it follows that $\neg(b < a)$. Therefore either $a < b$ or $a\ [<]_{\mathsf{Un}}\ b$.

i) Consider the case where $a < b$.

Since there is a trace where b is consecutive to a there can be no event c where $a < c < b$.

Hence

$$b \in \mathsf{n}(\{a\}, <)$$

If it is not the case that

$$b \in \mathsf{cns}(a, S - B_a, <)$$

then there must be some e where $e < b$ and $e \; [<]_{\mathsf{Un}} \; a$. However, if $e < b$ then e must have occurred at some point in t_0 and hence $e \in B_a$. This leads to a contradiction, so we must have that $b \in \mathsf{cns}(a, S - B_a, <)$.

ii) Consider the case where $a \; [<]_{\mathsf{Un}} \; b$.

Just as in the previous case if there is any $e \in E$ where $e < b$ and $e \; [<]_{\mathsf{Un}} \; a$, then e must occur in t_0. Hence b must be an element of $\mathsf{min}(S - B_a, <)$. So by definition $b \in \mathsf{cns}(a, S - B_a, <)$. This completes the proof of a).

Proof of b)

This case is very straightforward and is included for completeness. Choose any $b \in S$. Let

$$X = \{x \in E \mid x < a \;\; \text{or} \;\; x < b\}$$

Let

$$Y = E - \{a, b\} - X$$

Let t_0 be any trace of $<$ restricted to X. Let t_1 be any trace of $<$ restricted to Y. Clearly $t_0 \cdot a \cdot b \cdot t_1$ is a trace of $<$.

That completes the proof of the Lemma. □

The first element in a trace of $<$ has to come from $\mathsf{min}(E, <)$. Hence we can define the system behaviour for a causal ordering as follows.

Definition 4.3: For a set $S \subseteq E$ define a recursive process algebra term by

$$P(S, <) = \sum_{\{a \in S\}} a \cdot P(\mathsf{cns}(a, S, <), <)$$

and $P(\emptyset, <) = 0$.

Where $a \cdot P$ denotes the usual sequential composition of action and process, and the summation is nondeterministic choice (as standard in both CCS and CSP [22], [17]).

Definition 4.4: For a partial order $<$ on events E define the observable behaviour of $<$ to be the process:

$$P_< = P(\mathsf{min}(E, <), <)$$

Define the observable behaviour for partial order scenario M with causal ordering $<_{\mathcal{C}}$ to be the process:

$$P(M) = P_{<_{\mathcal{C}}}$$

Lemma 4.5: Let $<$ be a partial order on events E. The traces of process $P_<$ are exactly the traces of $<$.

Proof

The proof of Lemma 4.5 is immediate from Lemma 4.2. For processes P and Q we use the notation $P \xrightarrow{a} Q$ to denote P is strong bisimulation equivalent to $a \cdot Q + P'$ for some process P'. Let

$$a_0 \cdot a_1 \cdots a_n$$

be a trace for the partial order $<$.

To prove the Lemma it is enough to prove that there are processes P_i where $P_i \xrightarrow{a_i} P_{i+1}$, $P(M) = P_0$ and $P_{n+1} = 0$, where 0 is the empty process. Define

$$U(a_i) = \{e \in E \mid e\ [<]_{\mathsf{Un}}\ a_i\}$$
$$A_j = \{a_i \mid 0 \leq i \leq j\}$$
$$S_{i+1} = \mathsf{cns}(a_i, U(a_i) - A_j, <)$$

It follows from Lemma 4.5 that we may define $P_{i+1} = P(S_{i+1}, <)$. Note $S_{n+1} = \emptyset$ so that $P_{n+1} = 0$. This completes the proof.

□

In [10], a process algebra semantics is defined for basic MSCs (which we refer to as partial order scenarios) that characterises system traces. In that work, they provide an algebraic semantics for each of the various constructs for basic MSCs. They also define a particular form of concurrent composition that reflects the asynchronous communication between processes defined by the MSC standard [31]. This is unnecessary for our purposes. We only require a process that captures the system behaviour directly from the causal order. This allows us to use the more elementary definitions given in this paper.

V. Automata of Unordered Sets

From the definition for $P(M)$ we can construct a finite state automaton $\mathcal{A}(M)$ that also defines the traces of the system. The states of the automaton are unordered sets $S \subseteq E$. The initial state is $S_0 = \mathsf{min}(E, <_{\mathcal{C}})$, and $\emptyset$ is the accepting state. For each $a \in S$ there is a transition

$$S \xrightarrow{a} \mathsf{cns}(a, S, <_{\mathcal{C}})$$

From the recursive definition of $P(M)$ we can also give a recursive construction for $\mathcal{A}(M)$. Define States_i and Trans_i recursively as follows.

- $\mathsf{States}_0 = \{S_0\}$, $\mathsf{Trans}_0 = \emptyset$
- Define States_{n+1} to be the sets $\mathsf{cns}(a, S, <_{\mathcal{C}})$ where $S \in \mathsf{States}_n$ and $a \in S$. Define Trans_{n+1} to be the transitions $S \xrightarrow{a} \mathsf{cns}(a, S, <_{\mathcal{C}})$ for $S \in \mathsf{States}_n$ and $a \in S$.

The states of $\mathcal{A}(M)$ are the union of all the States_i. The transitions of $\mathcal{A}(M)$ are the union of all the Trans_i. The language accepted by $\mathcal{A}(M)$ is exactly the set of traces given by $P(M)$. The proof of Lemma 4.5 essentially defines the correspondence between the traces of the automaton and the traces of the process.

Fig. 6. Automaton Representation of Partial Order Behaviour

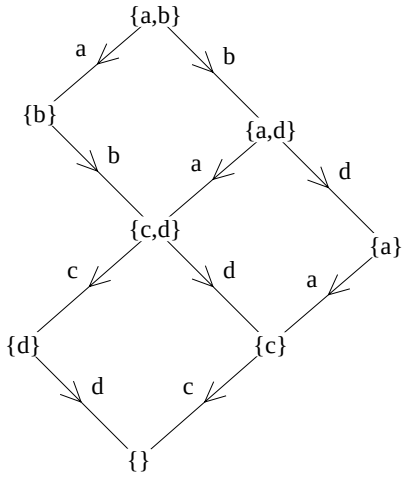


Figure 6 gives an example of the automaton for events $\{a, b, c, d\}$ with causal order $a < c$, $b < c$, $b < d$.

The start state in Figure 6 is $\{a, b\}$.

The automaton $\mathcal{A}(M)$ also reflects a lattice structure on the unordered sets of $<_{\mathcal{C}}$. Define a (non-strict) partial order $\prec$ on unordered sets as follows. Let

$$\overline{S} = S \cup \{x \mid \exists y \in S : x >_{\mathcal{C}} y\}$$

Define $S_1 \prec S_2$ when

$$\overline{S_1} \subseteq \overline{S_2}$$

The order $\prec$ defines a lattice structure over the unordered sets. It turns out that there is a transition $S \xrightarrow{a} S'$ in $\mathcal{A}(M)$ if and only if $S \prec S'$ and there is no unordered set U where $S \prec U \prec S'$. Hence we can construct the Hasse diagram for $\prec$ directly from $\mathcal{A}(M)$.

For $U \subseteq E$, U is referred to as an upper section if $U = \overline{W}$ for some $W \subseteq E$. In [1], an automaton $\mathcal{A}'(M)$ is defined with states given by the upper sections of E. Transitions are given by $U \xrightarrow{a} (\overline{U - \{a\}})$ for each $a \in \mathsf{min}(U, <_{\mathcal{C}})$. It turns out that $\mathcal{A}'(M)$ is isomorphic to $\mathcal{A}(M)$. The isomorphism works as follows.

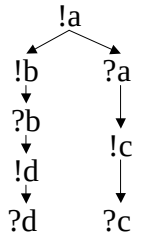
Given a state U in $\mathcal{A}'(M)$, this maps to the state $\mathsf{min}(U, <_{\mathcal{C}})$ in $\mathcal{A}(M)$. A state S in $\mathcal{A}(M)$ maps to $\overline{S}$ in $\mathcal{A}'(M)$. A transition $S \xrightarrow{a} S'$ maps to $\overline{S} \xrightarrow{a} \overline{S'}$. This gives an isomorphism since for any upper section U, $U = \overline{\mathsf{min}(U, <_{\mathcal{C}})}$. Also for any unordered set S, $S = \mathsf{min}(\overline{S}, <_{\mathcal{C}})$.

In this paper we use unordered sets to construct the system behaviour, rather than upper sections, since they give a clearer description of possible consecutive events at each point of a system trace. That gives us greater insight into how races can occur, which is what we require for the proofs of the main results.

VI. Inherent Causal Behaviour

A partial order $<$ on events preserves the message ordering when $!e <?e$ for every message e. Let $<_{\mathcal{C}}$ be the causal ordering for a partial order scenario.

Fig. 7. Inherent Ordering of Figure 2



Definition 6.1: The inherent causal ordering $<_I$ of $<_{\mathcal{C}}$ is defined to be the transitive closure of the following binary relation $<$. For every event x and message e define:

1) $x <\, !e \iff x <_{\mathcal{C}} !e$

2) $!e <\, ?e$

Note that when regarding a partial order as a set of pairs, we have

$$(<_I) \subseteq (<_{\mathcal{C}})$$

Figure 7 gives a graphical depiction of the inherent ordering for Figure 2.

The inherent ordering is the causal order of some partial order scenario. This follows from the next Theorem.

Theorem 6.2: The inherent causal ordering $<_I$ of a partial order scenario with processes $\mathcal{P}$ is the transitive closure of the following binary relation $<_0$. For every event x and message e define:

1) $x <_0 !e \iff \exists\, P \in \mathcal{P}$ such that $x <_P !e$

2) $!e <_0 ?e$

Proof

Recall that $<_I$ is the transitive closure of the binary relation $<_1$ defined by:

1) $x <_1 !e \iff x <_C !e$

2) $!e <_1 ?e$

Clearly $<_I$ is an extension of $<_0$ so it only remains to prove that $<_I$ is contained in the transitive closure of $<_0$. Let $<_0^*$ be the transitive closure of $<_0$. For a partial order $<$ on E let $a <^+ b$ denote that $a < b$ and there is no event w where $a < w < b$.

Suppose we have x, $y \in E$ where $x <_I y$. The proof now splits depending on whether y is a send or receive event.

i) Consider the case where $y = !e$. In this case $x <_I !e$ if and only if $x <_{\mathcal{C}} !e$.

Let u_i be events where

$$x <_{\mathcal{C}}^+ u_1 <_{\mathcal{C}}^+ u_2 \cdots <_{\mathcal{C}}^+ u_n <_{\mathcal{C}}^+ !e$$

We prove this case by induction on n.

If x and $!e$ are part of the same process we are done. So we may assume this is not the case. Let $!e$ belong to process P. Let i be the minimal value such that u_i also belongs to P.

Suppose that $u_i = !f$ for some message f. By definition u_{i-1} does not belong to P. We therefore have $u_{i-1} <_{\mathcal{C}}^{+} !f$, but u_{i+1} and $!f$ are on different processes. However, if for any events g and h on different processes we have $g <_{\mathcal{C}}^{+} h$ then $h = ?k$ and $g = !k$ for some k. Hence u_i can not be a send event.

We may then suppose that $u_i = ?f$ for some message f. It follows from what we have just said that $u_{i-1} = !f$. We have thus constructed a sequence:

$$x <_{\mathcal{C}}^{+} u_1 <_{\mathcal{C}}^{+} u_2 \cdots <_{\mathcal{C}}^{+} u_{i-1} = !e$$

We may now apply the induction hypothesis to prove that $x <_0^* u_{i-1}$. By definition we also have $u_{i-1} <_0 u_i$ and $u_i <_0 !e$. Hence by transitivity $x <_0^* !e$. This completes the induction step.

The base case is when $x <_{\mathcal{C}}^{+} u_1 <_{\mathcal{C}}^{+} !e$ and u_1 belongs to P, but x does not. This can only be true if $u_1 = ?f$ and $x = !f$ for some f. In which case it follows that $x <_0^* u_1 <_0 !e$ by definition. That completes the proof by induction.

ii) Consider the case where $y = ?e$.

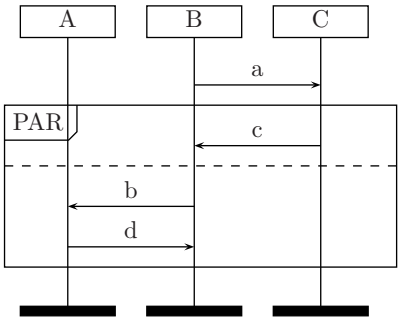
From the definition for $<_I$ this can only be if

$$x <_I !e <_I ?e$$

Hence by case i) we are done since $!e <_0 ?e$. □

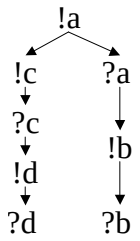
In Figure 8 we have illustrated the inherent causal scenario of Figure 2 in the form of an MSC using a parallel construct. With the parallel construct we have separated message c into one concurrent thread and messages b and d into another. The dotted line delineates the two threads within the PAR construct. Note that since $!a$ is outside the parallel construct it must still occur before both $?c$ and $!b$. The illustrations used in the paper for inherent causal scenarios can be automatically generated from their inherent causal orders, but we will not describe that process here, as it is a distraction from the central theme of the paper.

Fig. 8. Inherent Causal Scenario of Figure 2 as MSC



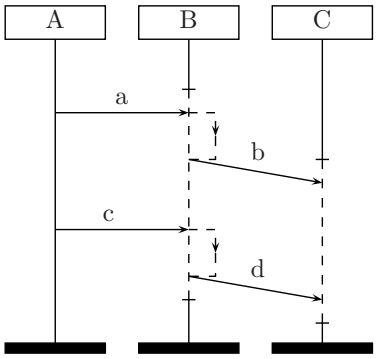
The inherent causal order for Figure 5 is shown in Figure 9. This is isomorphic as a partial order to the inherent causal order in Figure 7. However because of the partitioning of events between processes the inherent causal scenario for Figure 5 is very different to that of Figure 2.

Fig. 9. Inherent Ordering of Figure 5



One graphical depiction for the inherent causal scenario for Figure 5 is given in Figure 10. In this depiction all the events of process B are placed in a single coregion. This decouples $?b$ and $?c$ which removes the race. By placing all the events in a coregion, we must reintroduce desired orderings between events on that process by use of general ordering constructs. Hence, there is a general order construct to ensure $?a <_I !b$ and another to force $?c <_I !d$.

Fig. 10. Inherent Causal Scenario of Figure 5



In this particular case, it is not possible to use a parallel construct in the same way we did earlier to depict the inherent causal scenario. It is not as straightforward in this example to separate the concurrent behaviour into separate linear threads.

Each message forwarded by B is essentially treated independently. Hence, the behaviour of B is independent of the order that messages are sent to it by A. We can emphasis this more clearly, and simplify the inherent causal scenario at the same time, by using lost and found messages.

Fig. 11. Alternative Depiction for Inherent Causal Scenario of Figure 5

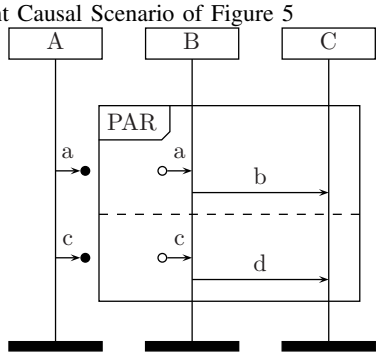


Figure 11 gives an alternative depiction of the inherent causal scenario for Figure 5. It specifies exactly the same causal order as Figure 10. With our convention, outlined in Section I, of only using lost and found events to provide a graphical means of splitting a message we have not lost any information by depicting a and c in this way. However, by splitting a and c into lost and found events we can parcel up the rest of the scenario into two concurrent linearly ordered threads within a parallel construct. Visually this more clearly describes the concurrent structure of the inherent causal scenario than Figure 10.

VII. Canonical Inherent Processes

Recall in Section IV we defined the observable process behaviour $P(M)$ of a partial order scenario M.

Definition 7.1: The inherent process behaviour of a partial order scenario M is defined to be

$$P_I(M) = P(\mathsf{min}(E, <_I), <_I)$$

Let $\sqsupset$ denote the standard simulation relation for process algebras. That is $P \sqsupset Q$ iff

for every transition $Q \xrightarrow{a} Q'$, there exists a transition $P \xrightarrow{a} P'$ where $P' \sqsupset Q'$

Theorem 7.2:

- $(<_I) \subseteq (<_{\mathcal{C}})$, and $P_I(M) \sqsupset P(M)$
- For any race free partial order $<$ that preserves message ordering, let $P_< = P(\mathsf{min}(E, <), <)$. Then $P_< \sqsupset P(M)$ iff $(<) \subseteq (<_I) \subseteq (<_{\mathcal{C}})$ and $P_< \sqsupset P_I(M)$

That is $P_I(M)$ is the canonical process that simulates $P(M)$ and is race free. To say that $(<_1) \subseteq (<_2)$ means that for every x and y in E, when $x <_1 y$ then $x <_2 y$.

This theorem proves that the order $<_I$ describes the maximal ordering with respect to simulation equivalence that is a race free weakening of $<_{\mathcal{C}}$. Hence constructing an MSC that has partial order semantics given by $<_I$ defines a new MSC that corrects any race conditions in M, and weakens the causal ordering of M as little as possible. It is straightforward to construct such an MSC.

The theorem is a consequence of the following Lemmas together with Lemma 4.5. For any partial order $<$ (which is not necessarily race free) let $T(<)$ be the set of total extensions of $<$.

Lemma 7.3: For partial orders $<_1$ and $<_2$ where

$$P_{<_1} \sqsupset P_{<_2}$$

then $(<_1) \subseteq (<_2)$

Proof

Note that $x < y$ iff for every trace in $T(<)$, x occurs before y in the trace. When $P_{<_1} \sqsupset P_{<_2}$ then the set of traces for $P_{<_2}$ is contained in the set of traces for $P_{<_1}$, that is $T(<_2) \subseteq T(<_1)$.

$x <_1 y \Rightarrow x$ occurs before y in every trace of $T(<_1)$

$\Rightarrow x$ occurs before y in every trace of $T(<_2)$

$\Rightarrow x <_2 y$

Hence $(<_1) \subseteq (<_2)$, which concludes the proof. □

Lemma 7.4: Given two partial orders $<_1$ and $<_2$,
$T(<_1) \subseteq T(<_2)$ iff $P_{<_1} \sqsubset P_{<_2}$

Proof

Note that $T(<_1) \subseteq T(<_2)$ iff $(<_2) \subseteq (<_1)$.

Given $(<_2) \subseteq (<_1)$, to prove $P_{<_1} \sqsubset P_{<_2}$, it is enough to prove that for any $S \subseteq E$ and $a \in E$,

$$\mathsf{cns}(a, S, <_1) \subseteq \mathsf{cns}(a, S, <_2) \tag{1}$$

Let

$$\begin{aligned} m_1 &= \mathsf{min}((S - \{a\}) \cup \mathsf{n}(\{a\}, <_1), <_1) \\ m_2 &= \mathsf{min}((S - \{a\}) \cup \mathsf{n}(\{a\}, <_2), <_2) \end{aligned}$$

We write $U \leq V$ for sets U, $V \subseteq E$, when for each $u \in U$, there is some $v \in V$ such that $u \leq v$. Note that since $(<_2) \subseteq (<_1)$ then $\mathsf{n}(\{a\}, <_2) \leq \mathsf{n}(\{a\}, <_1)$.

For a contradiction suppose that $x \in m_1$ and $x \notin m_2$. This implies there is some $y \in m_2$ such that $x <_2 y$. First consider if $y \in S - \{a\}$. Then $x <_1 y \in S - \{a\}$, hence $x \notin m_1$. This is a contradiction, hence we must have $y \in \mathsf{n}(\{a\}, <_2)$.

Since $\mathsf{n}(\{a\}, <_2) \leq \mathsf{n}(\{a\}, <_1)$, there is some $y' \in \mathsf{n}(\{a\}, <_1)$ such that $x <_2 y <_1 y'$. Therefore $x <_1 y' \in \mathsf{n}(\{a\}, <_1)$, and so $x \notin m_1$. Again a contradiction as required to complete the proof of equation 1. The proof that $T(<_1) \subseteq T(<_2)$ implies $P_{<_1} \sqsubset P_{<_2}$, is completed once we note that $\mathsf{min}(E, <_1) \subseteq \mathsf{min}(E, <_2)$.

The converse implication for the Lemma is straightforward. It is true for any processes P and Q that if $P \sqsupset Q$ then the set of traces of Q is contained in the set of traces for P. Since the traces of $P_{<_i}$ are exactly $T(<_i)$, the result is then immediate. That completes the proof of the Lemma. □

Lemma 7.5: For a partial order $<$ that preserves message ordering and is race free,

$$\Big((<) \subseteq (<_C)\Big) \Rightarrow \Big((<) \subseteq (<_I) \subseteq (<_C)\Big)$$

Proof

For this it is enough to prove that whenever $x < y$ then $x <_I y$. The proof splits into cases depending on whether y is a receive or send event.

– First suppose that $y = !e$ for some message e. Then $x < !e$ implies $x <_{\mathcal{C}} !e$ since $(<) \subseteq (<_{\mathcal{C}})$. By definition of $<_I$, $x <_{\mathcal{C}} !e$ implies $x <_I !e$.
– The other case is where $y = ?e$ for some message e. Since $<$ is race free, $x < ?e$ implies that $x < !e$. As above this implies $x <_I !e$. The ordering $<_I$ preserves message ordering, and hence $x <_I ?e$.

This completes the proof of the Lemma. □

VIII. Inherent Refinement Behaviour

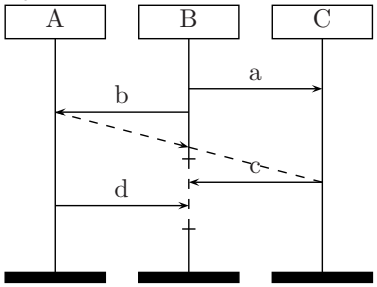
In this section, we define the inherent refinement ordering. This is the dual of the inherent causal order in that it characterises the minimal refinement of a causal order that is race free. This is proved by theorem 8.6 which is the dual result of theorem 7.2. However, it is not the case that we can prove the dual result to Theorem 6.2. That is the inherent refinement ordering can not necessarily be defined as the causal order for some partial order scenario. In Lemma 8.2 we give a counter example and prove that the refinement order of this example can never be the causal order of a partial order scenario. In Lemma 8.3 we prove that every refinement order can at least always be embedded in a race free partial order scenario.

Definition 8.1: The inherent refinement ordering $<_R$ of a causal ordering $<_{\mathcal{C}}$ is defined to be the transitive closure of the following binary relation $<$. For every event x and message e define:

- $x < !e \iff x <_{\mathcal{C}} ?e$
- $!e < ?e$

First note that $<_R$ is race free. Since it is clear from the definition that $x <_R ?e$ implies that $x <_R !e$ or $x = !e$. Also notice that the refinement order only extends $<_{\mathcal{C}}$ by forcing particular send events to be delayed so that other events may occur first, and hence is implementable.

Fig. 12. Inherent Refinement Ordering of Figure 2 as MSC



If the partial order scenario is represented as an MSC M then the inherent refinement ordering can be constructed by adding suitable general orderings to M. These general order constructs cause appropriate send events to wait until the relevant receive events have occurred. For example the MSC in Figure 12 describes the inherent refinement order for the partial order scenario in Figure 2. Note that Figure 12 is *not* a partial order scenario. The general order construct here acts across processes and not within a single process. The causal order for a partial order scenario can impose an arbitrary ordering within a process, but events in separate processes can only be ordered because of inter-process communication. The general ordering in Figure 12 is clearly not the result of any inter-process communication. In fact we prove in Lemma 8.2 that the refinement order for Figure 2 can never be the causal order for a partial order scenario.

Lemma 8.2: Let $<_R^0$ be the refinement order of the partial order scenario given by the MSC in Figure 2. Then there is no partial order scenario whose causal order is an extension of $<_R^0$.

Proof

Recall from definition 2.4 that the causal order for a partial order scenario is defined as the transitive closure of the relation given by

$$\Big(\bigcup_{P \in \mathcal{P}} (<_P) \Big) \cup \mathsf{Msg}$$

Where $<_P$ are the various process orders for the scenario. The process orders for A, B and C inferred by $<_R^0$ are just those defined by Figure 3. Any partial order scenario that extends $<_R^0$ has to be the result of extending the process partial orders $<_A$, $<_B$ and $<_C$. Note that $<_A$ and $<_C$ are total orders and so can not be extended any further. Hence any causal order that is an extension of $<_R^0$ must extend $<_B$.

However, it is clear that no extension of $<_B$ can cause $!b$ to be ordered before $!c$ in the resulting causal order. That is there are no extensions of $<_A$, $<_B$ and $<_C$ that result in a causal order that extends $<_R^0$. As required to complete the proof. □

Although the inherent refinement order is not a true causal order, we can embed it within a race free partial order scenario. That is, it is possible to add messages to a partial order scenario so that the resultant scenario is race free. In addition, when restricted to the events of the original scenario the new scenario defines exactly the same process orders and its causal order is exactly the inherent refinement ordering of the original scenario. This is precisely stated in Lemma 8.3.

Lemma 8.3: Let M be a partial order scenario on events E, with processes $\mathcal{P}$ and process ordering $<_P$ for each $P \in \mathcal{P}$. Let $<_R$ be the inherent refinement order for M.

There is a race free partial order scenario M^m on events E^m, with processes $\mathcal{P}$, process orderings $<_P^m$ for each $P \in \mathcal{P}$ and causal order $<_{\mathcal{C}}^m$ such that

1) For each $P \in \mathcal{P}$, $E(P) \subseteq E^m(P)$
2) For each $P \in \mathcal{P}$ and each $x, y \in E(P)$

$$x <_P y \iff x <_P^m y$$

3) For each $x, y \in E$

$$x <_R y \iff x <_{\mathcal{C}}^m y$$

Proof

We initialise M^m to be M and add new messages and modify the process ordering as follows.

Recall that the refinement order is defined by adding $x <_R !e$ whenever $x <_{\mathcal{C}} ?e$.

For each such pair where $x \in P_1$ and $!e \in P_2$ we define a new message m_e^x where we add $!m_e^x$ to $E^m(P_1)$ and $?m_e^x$ to $E^m(P_2)$.

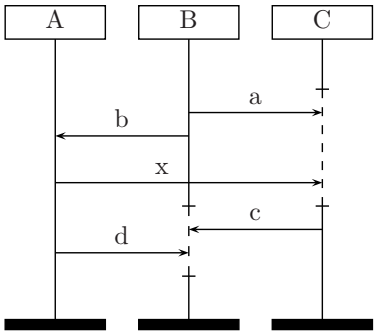
The set E^m consists of E together with all such new events.

For each such pair add $x <_{P_1}^m !m_e^x$, and $?m_e^x <_{P_2}^m !e$.

The causal order for M^m restricted to E is exactly $<_R$ and the process orders for M^m restricted to E are exactly the same as for M, as required. □

Figure 13 shows one way in which the refinement order for Figure 2 can be embedded in a partial order scenario. This scenario is given by the construction outlined in the proof of Lemma 8.3.

Fig. 13. Inherent Refinement Ordering Embedded in Partial Order Scenario



The refinement order itself is the unique minimal race free refinement of a causal order, as proved below in theorem 8.6. However, the choice of embedding for the refinement order is not unique. The UML and MSC standards contain other constructs that could be used to embed the refinement ordering in a scenario. In this paper, we do not further address the problem of choosing how to embed a refinement order within a scenario. Such a choice is best left to system designers who will want to use constructs suited to their particular circumstances.

Lemma 8.4:

$$(<_{\mathcal{C}}) \subseteq (<_R)$$

Proof

To prove this suppose $x <_{\mathcal{C}} y$. The proof is split into two cases depending on whether y is a send or receive event.

- If $y = !e$ for some e, then $y <_{\mathcal{C}} ?e$.
  Hence from the definition $x <_{\mathcal{C}} ?e$ and hence $x <_R !e$.
  That is $x <_R y$.
- When $y = ?e$, then $x <_R !e$.
  Also $!e <_R ?e$, hence by transitive closure, $x <_R ?e = y$.
  This completes the proof of the Lemma. □

Lemma 8.5: For any race free transitive partial order $<$ that preserves messages and where $(<_{\mathcal{C}}) \subseteq (<)$, then

$$(<_{\mathcal{C}}) \subseteq (<_R) \subseteq (<)$$

Proof

To prove this first consider an event x and message e where $x \neq ?e$ and $x <_{\mathcal{C}} ?e$.

Therefore $x <_R !e$.

Since $(<_{\mathcal{C}}) \subseteq (<)$, we have $x < ?e$.

Since $<$ is race free we have $x < !e$. Hence, if $x <_R !e$ then $x < !e$.

Since $<$ preserves messages it trivially follows that $!e <_R ?e$ implies $!e < ?e$.

Hence as $<$ is transitive we have proved that $(<_R) \subseteq (<)$. □

Given the Lemmas already proved in Section VII we have thus proved the following theorem, which is the dual to theorem 7.2.

Theorem 8.6: Let $P_R(M) = P(\mathsf{min}(E, <_R), <_R)$, then

- $(<_{\mathcal{C}}) \subseteq (<_R)$, and $P(M) \sqsupset P_R(M)$
- For any race free partial order $<$ that preserves message ordering, let $P_< = P(\mathsf{min}(E, <), <)$. Then $P(M) \sqsupset P_<$ iff $(<_{\mathcal{C}}) \subseteq (<_R) \subseteq (<)$ and $P_R(M) \sqsupset P_<$

Hence $<_R$ is the canonical refinement of the causal order that corrects all race conditions in the specification.

IX. Inline Iteration

In this section, we extend the notion of partial order scenarios with weak compositional iteration. Iterative scenarios cannot be characterized by a single partial order scenario and instead define a set of partial order scenarios. This set can be infinite if the iteration is unbounded. The semantics defined here is the same as the MSC/UML semantics for iterative scenarios.

First we define in-line composition of partial order scenarios. Let E_1 and E_2 be disjoint sets of events, and let $l : E_1 \cup E_2 \longrightarrow L$ be the labelling function. Let M_1 be a partial order scenario defined over E_1 consisting of processes $\mathcal{P}_1$, and M_2 be a partial order scenario over E_2 consisting of processes $\mathcal{P}_2$. Note there is no particular relationship between $\mathcal{P}_1$ and $\mathcal{P}_2$. I.e. they do not have to be identical and may even be distinct. Although the event sets are distinct the events can have the same labels. Let $<^i_P$ denote the local order for process P in scenario M_i.

Definition 9.1: For partial orders $<_1$ on $S_1 \subseteq E_1$, and $<_2$ on $S_2 \subseteq E_2$ define the ordering $(<_1; <_2)$ to be the transitive closure of:

- *For $a, b \in S_i$, $a(<_1; <_2)b$ iff $a <_i b$*
- *For all $a \in S_1$ and $b \in S_2$, $a(<_1; <_2)b$*

Definition 9.2: The in-line composition $M = M_1; M_2$ is defined to be the partial order scenario consisting of events $E_1 \cup E_2$ and processes $\mathcal{P}_1 \cup \mathcal{P}_2$.

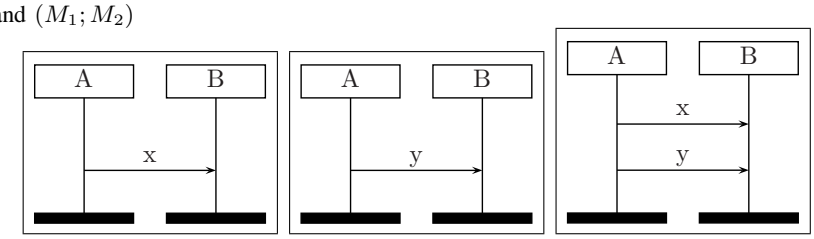
The local order $<_P$ for M splits into the following cases:

- *For $P \in \mathcal{P}_1 - \mathcal{P}_2$, $(<_P) = (<^1_P)$.*
- *For $P \in \mathcal{P}_2 - \mathcal{P}_1$, $(<_P) = (<^2_P)$.*
- *For $P \in \mathcal{P}_1 \cap \mathcal{P}_2$, $(<_P) = (<^1_P; <^2_P)$.*

We write $M; M$ to denote the in-line composition of two copies of M where we have renamed all the events in the second copy to be distinct from the first copy, whilst preserving the local orderings and labels. The expression M^n is defined to be n copies of M composed in-line: $M; M; \ldots; M$.

It is important to realize that the causal order for $M_1;\ M_2$ in general is *not* $(<_{\mathcal{C}_1}; <_{\mathcal{C}_2})$. The in-line composition of two partial order scenarios represents the graphical intuition of concatenating two scenarios by visually attaching one after the other, as can be seen by examining the MSC example shown in Figure 14.

Fig. 14. M_1, M_2 and $(M_1; M_2)$



Iteration will be specified in terms of sets of partial order scenarios. Hence, we need to extend the definition of composition to cater for this.

Definition 9.3: An extended scenario $\mathbb{S}$ is defined to be a finite set of partial order scenarios. Define the traces of $\mathbb{S}$ to be the union of the traces for the partial order scenarios contained in $\mathbb{S}$.

Definition 9.4: Let $\mathbb{S}_1$ and $\mathbb{S}_2$ be extended scenarios. Define $\mathbb{S}_1; \mathbb{S}_2$ to be

$$\{(M_1;\ M_2) \mid M_1 \in \mathbb{S}_1,\ M_2 \in \mathbb{S}_2\}$$

We can now define in-line iteration for extended scenarios as follows.

Definition 9.5: The n-th iteration for extended scenario $\mathbb{S}$ is defined to be the set

$$\mathbb{S}^n = \{M_{i_1}; \ldots; M_{i_n} \mid M_{i_j} \in \mathbb{S}\}$$

Define the in-line iterative loop construct $\mathsf{loop}\langle m, n\rangle(\mathbb{S})$ to be the union

$$\bigcup_{i=m}^{i=n} \mathbb{S}^i$$

$\mathsf{loop}\langle 0, \infty\rangle(\mathbb{S})$ denotes the union of all finite iterations of $\mathbb{S}$.

Note this is an infinite set of partial order scenarios and so not an extended scenario.

For a partial order scenario M we will adopt the convention of writing $\mathsf{loop}\langle m, n\rangle(M)$ as short hand for $\mathsf{loop}\langle m, n\rangle(\{M\})$. When m and n are finite $\mathsf{loop}\langle m, n\rangle(M)$ is also finite. When resolving races for $\mathsf{loop}\langle m, n\rangle(M)$ we could simply resolve the races in each separate iteration M^k. The result may no longer be a simple iteration, depending on how the races

are resolved. However, the result is still an extended scenario. Therefore from a theoretical perspective resolving races in bounded loops does not pose a serious problem since it can be reduced to resolving races in a finite set of partial order scenarios. An unbounded loop $\mathsf{loop}\langle 0, \infty\rangle(M)$ denotes an iteration that can occur any finite number of times, but where it is not known in advance how many times. For example, unbounded loops can be used to represent 'do until' iterations. This type of iteration could occur for example where a base station is attempting to re-establish a dropped connection and is repeatedly sending a connect request. It will continue to send the request until it gets an acknowledgement. The question that remains for the rest of the section is how to resolve races in unbounded loops.

Fig. 15. Unbounded iterative loop

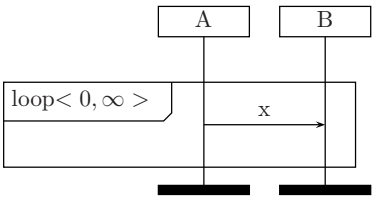


Figure 15 gives the graphical depiction for $\mathsf{loop}\langle 0, \infty\rangle(M)$ when M contains only a single message x between process A and B. In this example note that M^2 will contain a race. As we now have distinct messages that can have the same label, we are no longer free to identify a message with its label as we did in earlier examples. In this example let there be events $?e_1$ and $?e_2$ with $l(e_1) = l(e_2) = x$. Then $?e_1 <_{\mathcal{C}} ?e_2$ but $\neg(?e_1 <_{\mathcal{C}} !e_2)$, which defines a race. For M^n there will be n consecutive copies of the message x from A to B, resulting in $n(n-1)/2$ race conditions. This example illustrates that even when a basic scenario is race free the iteration of the scenario may well not be.

Definition 9.6: An iterative scenario is defined recursively as

- *Any partial order scenario.*
- *Any scenario of the form* $\mathbb{S}_1; \mathsf{loop}\langle 1, \infty\rangle(\mathbb{S}_2); \mathbb{S}_3$ *where* $\mathbb{S}_1$*,* $\mathbb{S}_2$ *and* $\mathbb{S}_3$ *are iterative scenarios.*

For brevity we will write $\mathbb{S}^\infty$ *as shorthand for* $\mathsf{loop}\langle 1, \infty\rangle(\mathbb{S})$ *from now on.*

Notice that since $\mathbb{S}_1$, $\mathbb{S}_2$ or $\mathbb{S}_3$ could be the empty set in this definition, we are free to combine extended scenarios by adding infinite loops whenever we wish. This definition restricts those infinite sets of partial order scenarios we wish to consider so that they must be the result of

the loop construct. Note we constrain unbounded loops so that they iterate at least once. This is done to avoid unnecessary detail in the proofs, and can be done without loss of generality.

Lemma 9.7: There is no generalisation of the iterative scenario in Figure 15 that is race free.

Proof

In this case it is only possible to generalise the iteration by generalizing the body of the loop. Clearly the loop body can not be any further generalized. The only partial order weaker than $!x <?x$ is the unordered set $\{!x, ?x\}$. This however is not a partial order scenario. Such scenarios must preserve message ordering, hence there is no generalisation possible for the iteration. □

Fig. 16. Bounded and not Convergent Scenario

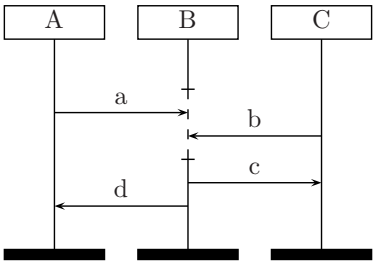
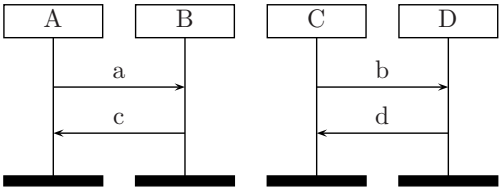


Fig. 17. Convergent and not Bounded Scenario



This lemma proves that the idea of weakening an iterative scenario to remove races is not possible in general. The problem stems from the fact that iteration semantics are defined via weak composition of the iterations of the loop body. Therefore the events for a process P in iteration n are by definition always ordered before the events of P in iteration $n+1$. However, to resolve races between events from different iterations by generalisation requires some way to specify a weakening of the local process orders across these different iterations. In order to proceed further we need to characterise race conditions in iterative scenarios in a manner that can be easily checked.

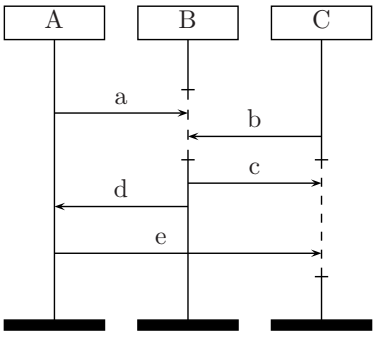
Definition 9.8: Let M be a partial order scenario over events E and causal order $<_{\mathcal{C}}$.

For $x \in E$, let $\overline{x} = \{x\} \cup \{y \in E \mid x <_{\mathcal{C}} y\}$ and let $\underline{x} = \{x\} \cup \{y \in E \mid y <_{\mathcal{C}} x\}$. For an event x let $P(x)$ be the process P where $x \in E(P)$. For a set of events $S \subseteq E$ let $P(S)$ be $\{P(x) \mid x \in S\}$.

Define M to be convergent if whenever there are events e and $?x$ where $P(?x) \in P(\overline{e})$, then

$$P(\underline{!x}) \cap P(\overline{e}) \neq \emptyset$$

Fig. 18. Convergent Embedding of Figure 16



Notice that the definition of convergence is constructive. Hence, it is straightforward to check if a scenario is convergent or not. The notion of convergence has similarities with the notion of boundedness defined in [1], [3], [21]. For a partial order scenario M define a graph G that has nodes given by the processes in M. Define an edge between nodes P and Q exactly when there is a message from P to Q. M is bounded if G is strongly path connected. That is there are paths between any two nodes in both directions. In [1], property checking for an arbitrary iterative message sequence chart (MSC) is proved to be undecidable. They prove that when the body of every iterative loop in an MSC is bounded then property checking becomes decidable.

Although the convergent and bounded definitions have similarities, they define distinct sets of partial order scenarios. Figure 16 shows a partial order scenario that is not convergent, but is bounded. It is not convergent since we have $P(\underline{!b}) \cap P(\overline{!d}) = \emptyset$ but $P(?b) \in P(\overline{!d})$. Figure 17 shows a partial order scenario that is convergent, but is not bounded.

Theorem 9.9: Let M be a race free partial order scenario. Then M is convergent if and only if M^{∞} is race free.

Proof

First we prove that if M is convergent then M^∞ is race free. For a contradiction suppose that M is convergent and there is a race in M^∞. For an event $e \in E$ we will write e^n to denote the occurrence of e in the n-th iteration of the loop. We use the notation

$$e^n <^\infty_{\mathcal{C}} h^{n+m}$$

to denote that event e in the n-th iteration is less than h in the $(n+m)$-th iteration according to the causal order for M^{n+m}.

If M^∞ has a race (since M is race free) there must be events e and $?x$ such that $e^n <^\infty_{\mathcal{C}} (?x^{n+m})$ and $\neg(e^n <^\infty_{\mathcal{C}} (!x^{n+m}))$ for some $m > 0$.

Consider any events u and v where $\neg(u <_{\mathcal{C}} v)$ and $u^n <^\infty_{\mathcal{C}} v^{n+1}$. Then there must be events u_1 and v_1 where $u <_{\mathcal{C}} u_1$, $v_1 <_{\mathcal{C}} v$ and $P(u_1) = P(v_1)$. This is exactly the condition that $P(\overline{u}) \cap P(\underline{v}) \neq \emptyset$.

Since $e^n <^\infty_{\mathcal{C}} (?x^{n+m})$ there must be some $g \in E$ where

$$e^n <^\infty_{\mathcal{C}} g^{n+m-1} <^\infty_{\mathcal{C}} (?x^{n+m})$$

Consider if $g^{n+m-1} <^\infty_{\mathcal{C}} (!x^{n+m})$. Then trivially there is no race between e and $?x$, which is a contradiction. Therefore $\neg(g^{n+m-1} <^\infty_{\mathcal{C}} (!x^{n+m}))$. That is g^{n+m-1} and $!x^{n+m}$ are in a race. In which case we must also have that g^1 and $!x^2$ are in a race. This last deduction is not vital for the proof, but it makes the notation much less cumbersome as we proceed.

Let $?x$ belong to process P. Consider if $P \notin P(\overline{g})$. Then there must be a process Q and events h_1, $h_2 \in E(Q)$ where $g <_{\mathcal{C}} h_1$ and $h_2 <_{\mathcal{C}} ?x$. Since there is a race between g^1 and $!x^2$ we have that $h_2 \not<_{\mathcal{C}} !x$. Therefore we have constructed a race between h_2 and $?x$. This is a contradiction as M is race free. Therefore $P \in P(\overline{g})$. Since M is convergent this implies $P(\underline{!x}) \cap P(\overline{g}) \neq \emptyset$. From our earlier observation this can only be true when $g^1 <^\infty_{\mathcal{C}} (!x^2)$. This contradicts that there is a race between g and $?x$, which in turn contradicts that there is a race between e and $?x$. Hence we have proved that if M is convergent then M^∞ is race free.

To prove the converse we prove that if M is not convergent then there is a race in M^∞. Suppose that we have events e and $?x$ where $P(?x) \in P(\overline{e})$ but $P(\underline{!x}) \cap P(\overline{e}) = \emptyset$.

Since $P(?x) \in P(\overline{e})$ we have that $e^1 <^\infty_{\mathcal{C}} (?x^2)$. As we observed earlier in the proof $u^n <^\infty_{\mathcal{C}} v^{n+1}$ if and only if $P(\overline{u}) \cap P(\underline{v}) \neq \emptyset$. Therefore $P(\underline{!x}) \cap P(\overline{e}) = \emptyset$ can only be true

if $e^1 \not<_{\mathcal{C}} (!x^2)$. Hence we have a race between e and $?x$ in M^∞. That concludes the proof. □

This result gives us a precise characterization of when a loop will generate a race purely in terms of the causal ordering for the body of the loop. Theorem 9.9 will enable us to resolve races that occur as a result of iteration solely by modifying the body of the loop. This can usually be done by adding suitable messages to the loop body to ensure the result is convergent, as we shall demonstrate in the remainder of the section. First, we extend the notion of convergence.

Definition 9.10: Let M and N be partial order scenarios. We define M to be convergent with respect to N when for every $e \in E(M)$ and $?x \in E(N)$ if $P(?x) \in P(\overline{e})$ then $P(\underline{!x}) \cap P(\overline{e}) \neq \emptyset$.

This version of convergence precisely captures when the composition of two race free partial order scenarios $M; N$ is also race free. When M is not convergent with respect to N then usually it is possible to embed M in some M' so that M' is then convergent with respect to N and so $M'; N$ is race free.

Lemma 9.11: When M and N are race free partial order scenarios, then $M; N$ is race free if and only if M is convergent with respect to N.

The proof of this lemma is immediate from the definition of the causal order for $M; N$.

Definition 9.12: Let M be a partial order scenario with events E and causal order $<_{\mathcal{C}}$. We define $\{!m_i \in E \mid 1 \leq i \leq n\}$ to be a virtual lower cycle when:

- $!m_i \in \mathsf{min}(E, <_{\mathcal{C}})$ *for* $1 \leq i \leq n$
- $?m_i \in \mathsf{min}(E(P(?m_i)), <_{\mathcal{C}})$ *for* $1 \leq i \leq n$
- $P(?m_i) = P(!m_{i+1})$ *for* $1 \leq i \leq n-1$
- $P(?m_n) = P(!m_1)$

We define $\{!m_i \in E \mid 1 \leq i \leq n\}$ to be a virtual upper cycle when:

- $!m_i \in \mathsf{max}(E, <_{\mathcal{C}})$ *for* $1 \leq i \leq n$
- $?m_i \in \mathsf{max}(E(P(?m_i)), <_{\mathcal{C}})$ *for* $1 \leq i \leq n$
- $P(?m_i) = P(!m_{i+1})$ *for* $1 \leq i \leq n-1$
- $P(?m_n) = P(!m_1)$

Scenario M contains no virtual cycles when it contains no upper or lower virtual cycles.

Fig. 19. Virtual Cycle

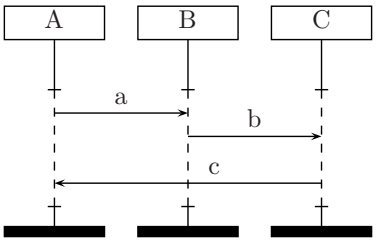


Figure 19 shows an example of a partial order scenario that contains a lower and upper virtual cycle, both given by $!a$, $!b$, and $!c$. Let U be the scenario in Figure 19. The virtual cycle means there is no race free partial order scenario that we can embed U into. To make progress we will need to avoid such scenarios.

Lemma 9.13: Let M and N be race free partial order scenarios. If N contains no virtual lower cycles then we can embed M in a race free scenario $M \downarrow N$ which is convergent with respect to N. Hence

$$(M \downarrow N); N$$

is race free.

If M contains no virtual upper cycles then we can embed N in a race free scenario $M \uparrow N$ so that M is convergent with respect to $M \uparrow N$. Hence

$$M; (M \uparrow N)$$

is race free.

Proof

We will prove $M \downarrow N$ exists by construction. Let $<_M$ be the causal order for M and $<_N$ be the causal order for N. Let E_M be the events for M and E_N be the events for N. Define a pair $?x \in E_N$ and $e \in E_M$ to be divergent when $P(?x) \in P(\overline{e})$ but $P(\underline{!x}) \cap P(\overline{e}) = \emptyset$.

We initialize $M \downarrow N$ to be M and add messages as follows. Let $?x \in E_N$ and $e \in E_M$ be a divergent pair. Let $u \in \mathsf{max}(E_M, <_M)$ where $e <_M u$. Let $P(!x) = P$ and $P(u) = Q$. Add a new message w to $M \downarrow N$ where $!w \in E(Q)$ and $?w \in E(P)$. It may be necessary to add P to $\mathcal{P}_M$ if this is not already present. Modify the process ordering in $M \downarrow N$ for P so that $u <_P !w$.

By adding $?w$ to P we have forced $e <!x$, where $<$ is the causal order for $(M \downarrow N); N$. However, we may have now introduced a race since it is possible that there is some $?x_1 \in$

$\mathsf{min}(P, <_N)$. If this is so we must introduce another message w_1 to $M \downarrow N$. Add $!w_1$ to P and add $?w <!w_1$. Also add $?w_1$ to process $P(!x_1)$. This removes the race between $?x_1$ and $?w$. However, there may be $?x_2 \in \mathsf{min}(E(P(!x_1)), <_N)$. In which case we have introduced a race between $?w_1$ and $?x_2$. We continue to add extra messages w_i until all such races are removed. Since N has no virtual lower cycles there will be some value k such that after the addition of w_k we do not introduce further races. Notice that all the new messages w_i are added to the end of M. So that $a <_M b$ if and only if $a < b$ for every a, $b \in E_M$.

For each divergent pair add new messages to $M \downarrow N$ as described. The final scenario is by construction convergent with respect to N. Also by construction M has been embedded in $M \downarrow N$.

The construction for $M \uparrow N$ is dual to the construction of $M \downarrow N$. Initialise $M \uparrow N$ to N. Let $?x$ and e be a divergent pair. Let $u \in \mathsf{min}(E_N, <_N)$ where $u <_N !x$. Add a new message w where $!w$ is added to process $P(e)$ and $?w$ is added to process $P(!x) = P$. Change the process order in $M \uparrow N$ so that $?w <_P !x$. As before we may have added a race condition in introducing this new message. Just as before we repeatedly add new messages w_i until no new races are introduced. Since M contains no virtual upper cycles this process is guaranteed to terminate. By adding new messages in this way for each divergent pair we construct $M \uparrow N$ as required.

□

Lemma 9.13 gives us the construction we need to resolve all races within an iterative scenario, provided it has no virtual cycles.

Theorem 9.14: Let $\mathbb{S}$ be an iterative scenario composed of race free partial order scenarios that contain no virtual cycles. Then we can resolve all race conditions in $\mathbb{S}$ by embedding each composite partial order scenario within a convergent scenario.

Proof

The proof is by recursion. Suppose we can not write $\mathbb{S}$ in the form $M_1; \mathbb{S}'; M_2$, for some partial order scenarios M_1, M_2 and iterative scenario $\mathbb{S}'$. In that case $\mathbb{S}$ can only be of the form M^∞ or M for some partial order M. When $\mathbb{S}$ is a partial order scenario then we can replace it by an embedding of the refinement causal order. Consider the case $\mathbb{S} = M^\infty$. By the hypothesis M contains no virtual cycles. Therefore $M \downarrow M$ is convergent with respect to M. That is $M \downarrow M$ is convergent in the sense of definition 9.8. Hence, $(M \downarrow M)^\infty$ is

race free by Theorem 9.9. That completes the proof for the base case.

Consider the case where $\mathbb{S}$ is of the form $\mathbb{S}_1; \mathbb{S}_2^\infty; \mathbb{S}_3$, and each $\mathbb{S}_i$ is race free. Write $\mathbb{S}_1 = \mathbb{S}_1'; M_1$ and $\mathbb{S}_3 = M_3; \mathbb{S}_3'$. Suppose $\mathbb{S}_2 = N_1; \mathbb{S}_2'; N_2$. Then

$$\begin{aligned} &\mathbb{S}_1'; (M_1 \downarrow (N_1 \uparrow N_2)); \\ &((N_1 \uparrow N_2); \mathbb{S}_2'; N_2)^\infty; \\ &(N_2 \uparrow M_3); \mathbb{S}_3' \end{aligned}$$

is a race free resolution for $\mathbb{S}$. If $\mathbb{S}_2 = M^\infty$ then we can assume M is convergent without loss of generality. In this case

$$\mathbb{S}_1'; (M_1 \downarrow M); M^\infty; (M \uparrow M_3); \mathbb{S}_3'$$

is race free. That completes the proof. □

Notice that we do *not* claim this resolution is canonical. This is still an active area of research, and it is not yet clear what will be the optimal resolution for races that are caused by iteration of partial order scenarios.

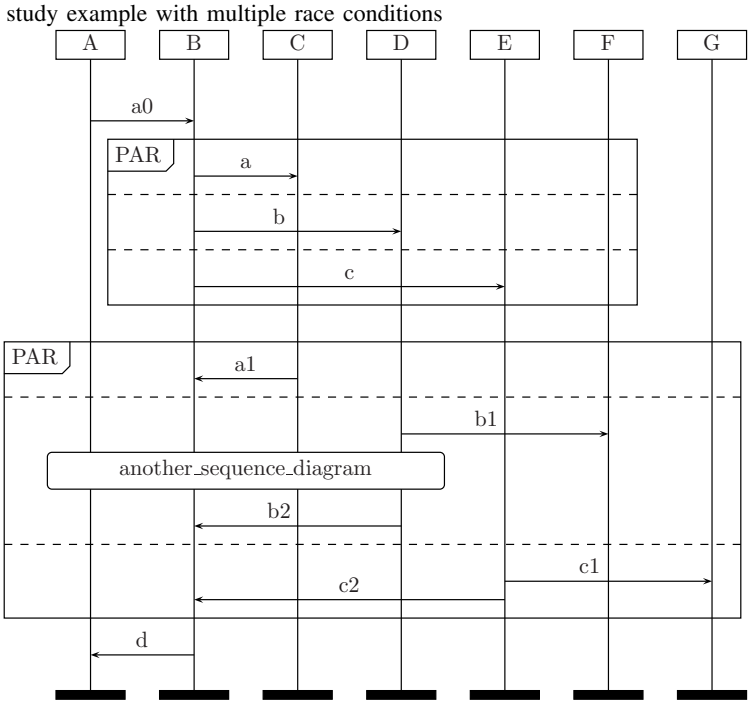
X. Industrial Case Study Examples

In collaboration with Motorola Research Labs, we have been conducting a number of case studies [5], [23] into automating pathology detection in MSC telecommunication specifications. The two examples examined in this section come from a proprietary study involving roughly a hundred UML 2.0 sequence diagrams. These formed part of the specification for a High Speed Downlink Packet Access (HSPDA) protocol stack.

There were approximately fifteen diagrams containing multiple race conditions found in the study. Of these, five contained multiple race conditions caused by iterative loops. The two examples included in this section contained the most complex race conditions from the study. As a result of the positive feedback from the case studies, Motorola is planning to incorporate the research presented here into a prototype tool suitable for larger scale evaluations with engineering groups, and to conduct further research into extending the results to more general UML sequence diagrams.

Figure 20 is an anonymized example from the Motorola case study, which contains multiple race conditions. The original diagram is a UML 2.0 sequence diagram that describes traffic channel allocation and activation between various processes for the HSPDA protocol. Process

Fig. 20. UML 2.0 case study example with multiple race conditions



A has delegated the task of determining what resource to allocate to process B. The inline reference in the MSC is a linear ordering of some events not shown here. These events can be ignored for the purposes of the example because they are linearly ordered.

In total we have the following six race conditions in Figure 20. Event $?a1$ is in a race with $!b$ and also with $!c$. Event $?c2$ is in a race with $!a$ and also with $!b$. Also event $?b2$ is in a race with $!a$ and also with $!c$. It may be that the authors implicitly assumed the downlink latency from B is much shorter than the uplink latency for the other processes. If this were true, it may be possible in practice for the specification to be realizable. However, it is far safer to rewrite the specification without these race conditions.

One way to remove these races would be to regroup the messages within a single parallel construct. Messages a and $a1$ could be grouped within the same thread of a parallel construct. Similarly b, $b1$, $b2$ and the inline reference could be grouped in a second thread. Finally c, $c1$ and $c2$ could be grouped in the third thread. Figure 21 depicts this solution. It seems reasonable to suppose this will not contradict what the authors originally intended.

Figure 21 is exactly the inherent causal scenario of Figure 20. In this case the inherent causal

Fig. 21. Inherent Causal Scenario for Figure 20 as MSC

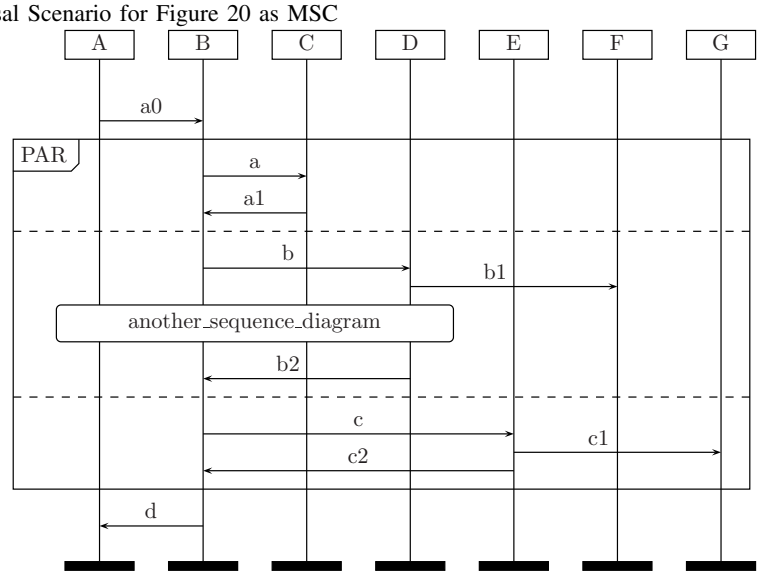
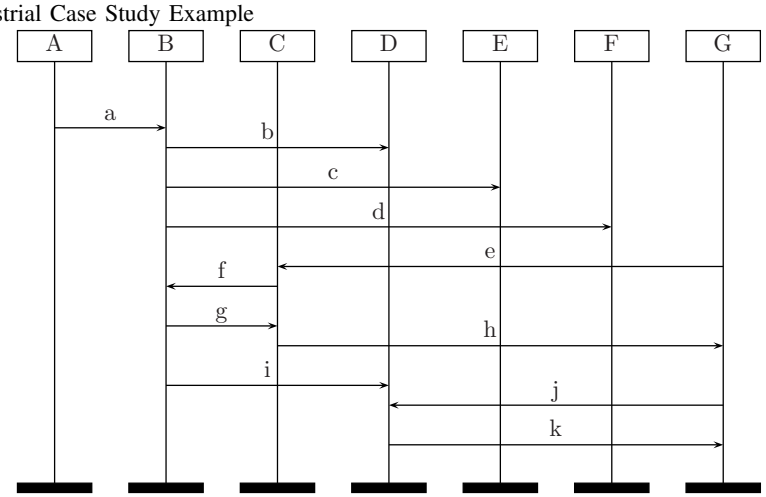


Fig. 22. Second Industrial Case Study Example



order for Figure 20 would seem to represent the specification intended by the authors, rather than the causal order of Figure 20 itself.

Figure 22 describes a second anonymized example from a case study where several race conditions were contained in a single scenario. This example can be viewed as multiple interacting threads that represent different features within one scenario.

Altogether there are seven races in the scenario. Receive event $?f$ is in a race with each of $?a$, $!b$, $!c$ and $!d$. The remaining three races occur between each of $?b$, $?i$ and $?j$. This example illustrates the value of resolving all the race conditions in a single global solution. For example,

message f is independent of d (in that it can be sent at any time after $?e$, and e is one of the initial messages in the scenario). Resolving this race by itself would not resolve the races between $?f$ and $?a$, $!b$ and $!c$. Rather it would complicate the process of identifying a solution to the other races. The inherent causal order for this scenario separates a, b, c and d into one concurrent thread, and messages e and f into a different thread. Adding this concurrent region resolves all four races simultaneously. One graphical depiction for the inherent causal scenario of Figure 22 is given in Figure 23. We have again used lost and found events as a graphical convenience to simplify the diagram.

Finally we provide an embedding of the refinement order for Figure 22 into a partial order scenario. This characterises the minimal strengthening of Figure 22 that can be achieved by adding new messages in order to impose the refinement ordering. In this depiction, we have resolved the race conditions by introducing three new messages. These are x from process B to C, y from process D to B and z from D to G. The embedding is given in Figure 24. Message x has resolved the races between $?f$ and $?a$, $!b$, $!c$ and $!d$. Message x ensures that process C is informed when B has sent messages $!b$, $!c$ and $!d$ and that it is now safe to send f. We include a coregion around $?x$ and $?e$ in order to avoid introducing new race conditions into the scenario. Messages y and z resolve the races between $?b$, $?i$ and $?j$. Message y ensures that b has been received before message i is sent, and z ensures that i is received before j is sent. Note we have to place y in a concurrent thread separate from c, d, e, f and x or else we would be introducing several new race conditions into the scenario. Similarly we include a coregion around $?z$ and $?h$ for the same reason.

The inherent causal ordering and the refinement ordering provide the practitioner with useful insight into the semantically consistent behaviour that can be extrapolated from the scenario. However, the practitioner may decide to combine the two orderings in some bespoke way to construct a solution that fits their own circumstances.

In this example it is possible that a combination of Figure 23 and Figure 24 give the appropriate solution to the race conditions in the original scenario. In the original scenario it could be that e and f are meant to be in a separate thread to a, b, c and d. However the races between b, i and j could be the omission of coordinating messages which should have been included. Deciding which parts of the inherent causal and refinement orderings the author intended to combine is not something that can be automated. Providing the practitioner with both solutions allows

Fig. 23. Inherent Causal Scenario for Figure 22 as MSC

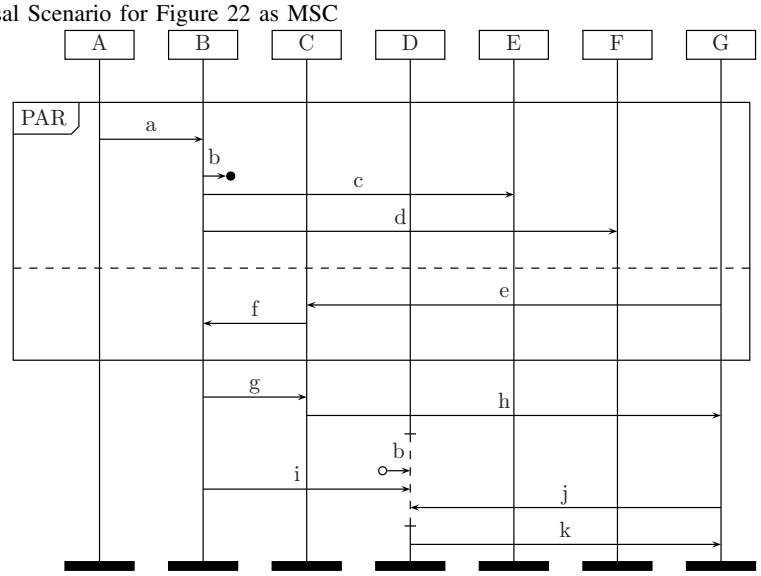
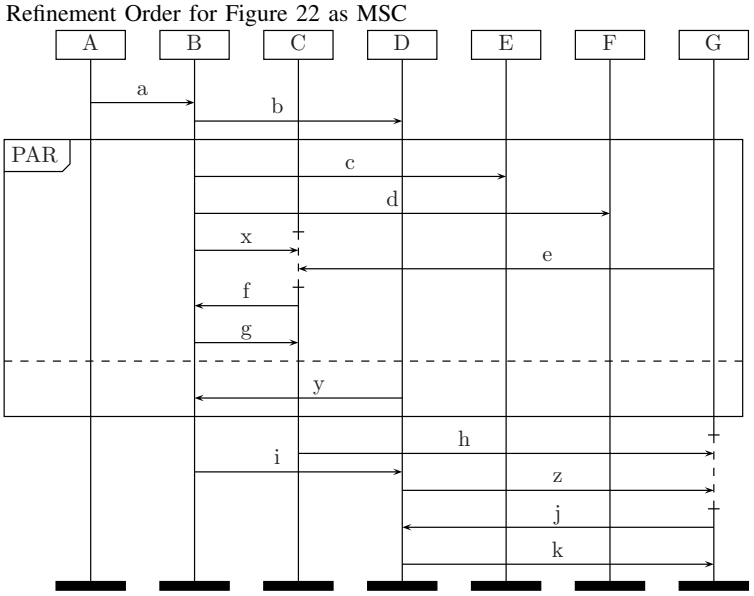


Fig. 24. Embedding of Refinement Order for Figure 22 as MSC



them to understand with greater clarity what semantically consistent behaviour is captured by the specification.

XI. Conclusion

The paper has proved that there is a canonical solution for correcting all race conditions within a partial order scenario by weakening the causal relationship. The inherent causal ordering that

defines the solution can also be presented in MSC or UML format by use of coregion, parallel and general ordering constructs.

The paper has also proved the dual result, that there is a canonical refinement of the specifications that corrects all race conditions. This is the inherent refinement ordering. Unlike the inherent causal order, the inherent refinement ordering cannot be represented by a partial order scenario. However, we can embed it within a partial order scenario, although the embedding is not unique. Together these inherent orderings provide a useful insight into the semantically consistent specifications that are possible for a distributed system. The construction of the inherent refinement ordering is based on the premise that additional messages are required to coordinate concurrent threads. Whereas the inherent causal ordering construction is based on the premise that concurrent threads have been synchronized at a given point when this should not occur.

Both the inherent causal and inherent refinement orderings can be automatically generated within a scenario authoring tool, particularly one that supports UML 2.0 sequence diagrams. The graphical depiction of a partial order scenario is not unique, and so there is great choice in deciding how to automate the construction of the inherent causal scenario. The paper has used illustrations, which can be automatically generated, that emphasize the concurrency within the inherent scenarios by maximizing the use of coregions and parallel constructs. It is also possible to automate the construction for the embedding of the refinement order into some partial order scenario. It is not yet clear if there is a canonical way to achieve this. Note this is a different problem than representing the refinement order as a UML/MSC sequence diagram. In that case, the representation is straightforward to automate since we are free to use general order constructs across processes as necessary.

Inherent and refinement orderings are likely to be of most use in complex scenarios that contain several race conditions that have a common cause, such as the examples in Section X. Practitioners are not intended to view the inherent and refinement orderings as prescriptive. The intention is that by providing these solutions to the practitioner they will be able to appreciate what semantically consistent behaviour can be extrapolated from their specifications. In the case study the most useful aspect of the inherent and refinement orderings from the practitioners perspective was that they clearly illustrated what was possible in an asynchronous distributed environment, and helped to identify where constraints would be needed to realize their intended

design. Generating error traces that identify individual race conditions would not have provided this overview because of the complex interrelationships between the race conditions.

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] R. Alur, K. Etessami, M. Yannakakis, Realizability and verification of MSC graphs, Proceedings of the 28th International Colloquium on Automata, Languages, and Programming, Springer, New York, pp 797-808, 2001.

[4] H. Ben-Abdhallah, S. Leue, 1997, Syntactic detection of process divergence and non-local choice in message sequence charts, Proceedings of the 3rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, pp 259-274, 1997.

[5] P. Baker, P. Bristow, C. Jervis, D. King, B. Mitchell, Automatic Generation of Conformance Tests From Message Sequence Charts, Proceedings of 3rd SAM Workshop 2002, The Broader Applicability of MSC and SDL, pp 170-198, LNCS 2599.

[6] M. Beyer, W. Dulz, F. Zhen, Automated TTCN-3 Test Case Generation by Means of UML Sequence Diagrams and Markov Chains, Proceedings of 12th Asian Test Symposium (ATS'03), IEEE 2003.

[7] Yves Bontemps, Pierre-Yves Schobbens, Synthesis of Open Reactive Systems from Scenario-Based Specifications, (ACSD'03)

[8] Yves Bontemps, Patrick Heymens, Turning high-level live sequence charts into automata, Proc of Scenarios and State Machines: Models Algorithms and tools, 24th International Conf. on Software Engineering, May 2002, ACM.

[9] Sang Chung, Hyeon Soo Kim, Hyun Seop Bae, Yong Rae Kwon, Byung Sun Lee, Testing of Concurrent Programs based on Message Sequence Charts, Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems, 1999. Proceedings, 1999, pp 72 - 82.

[10] T. Gehrke, M. Hilhn, H. Wehrkeim, An algebraic semantics for message sequence chart documents. In FORTE/PSTV'98, pages 3-18. Kluwer Academic Publishers, 1998.

[11] T. Gilb, Principles of Software Engineering Management, Addison Wesley Longman, 1988.

[12] Gerard J. Holzmann, Doron A. Peled, Message Sequence Chart Analyzer, United States Patent, 5,812,145.

[13] Gerard J. Holzmann, Doron A. Peled, and Margaret H. Redberg, An analyzer for message sequence charts, Software Concepts and Tools, 17(2), 1996.

[14] E. Gunter, A. Muscholl, D. Peled, Compositional Message Sequence Charts, TACAS 2001.

[15] David Harel, Werner Damm LSCs: Breathing Life into Message Sequence Charts, Formal Methods in System Design, 19, 45-80, 2001

[16] David Harel, H. Kugler, Synthesizing state-based object systems from LSC specifications, Proceedings of the 5th International Conference on Implementation and Application of Automata (CIAA'2000).

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

[18] S. Leue, L. Mehrmann, M. Rezai, M, Synthesizing software architecture descriptions from Message Sequence Chart specifications, Proceedings 13th IEEE International Conference on Automated Software Engineering, 1998.

[19] R. Lutz, Targeting Safety-Related Errors During Software Requirements Analysis, Proc. First ACM SIGSOFT Symp. Foundations of Software Eng, 1993.

[20] Lutz, Analyzing software requirements errors in safety-critical, embedded systems, Requirements Engineering, 1993., Proceedings of IEEE International Symposium on 4-6 Jan. 1993

[21] M. Lohrey, Safe realizability of high-level message charts. In Proceedings of the 13th International Conference on Concurrency Theory (CONCUR), 2002.

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

[23] B. Mitchell, R. Thomson, C. Jervis, Phase Automaton for Requirements Scenarios, Feature Interactions in Telecommunications and Software Systems VII, 77-84, 2003, IOS Press.

[24] Nelson, Clark, and Spurlock. *Curing the Software Requirements And Cost Estimating Blues*, PM: Nov-Dec, 1999.

[25] D. Peled: Specification and Verification using Message Sequence Charts, Proceedings of the IFIP International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII), 2000, and Electronic Notes in Theoretical Computer Science 65, No 7, 2002.

[26] E. Rudolph, I. Schieferdecker, J. Grabowski: Development of a MSC/UML Test Format. 153-164, Formale Beschreibungstechniken fur verteilte Systeme, 2000. Verlag Shaker 2000, ISBN 3-8265-7491-5.

[27] Telelogic, *Tau documentation*, Telelogic Web Site: `http://www.telelogic.com`.

[28] S. Uchitel, J. Kramer, J. Magee, Incremental elaboration of scenario-based specifications and behavior models using implied scenarios, ACM Transactions on Software Engineering and Methodology (TOSEM), V13, N1, pp 37—85, 2004.

[29] J. Whittle, J. Saboo, R. Kwan, From scenarios to code: an air traffic control case study, Proceedings. 25th International Conference on Software Engineering, 2003.

[30] J. Whittle, J. Schumann, Generating Statechart Designs From Scenarios, Proceedings 22nd international conference on on Software engineering, 2000.

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

[32] Object Management Group (OMG), *Unified Modeling Language (UML): Superstructure, Version 2.0*, 2003. Available from `http://www.omg.org`.

[33] E. Wong, J. R. Horgan, W. Zage, D. Zage and M. Syring, Applying Design Metrics to a Large-Scale Software System, (Motorola), Proceedings of the 9th International Symposium on Software Engineering Reliability (ISSRE '98), Paderborn, Germany, November 4-7, 1998.