Lazy buffer semantics for partial order scenarios

Bill Mitchell (`w.mitchell@surrey.ac.uk`)
Department of Computing, University of Surrey, Guilford, Surrey GU2 7XH, UK

July 30, 2007

Abstract. There exists a unique minimal generalisation of a UML sequence diagram (SD) that is race free, known as the inherent causal scenario. However, practitioners sometimes regard this solution as invalid since it is a purely mathematical construct that apparently does not describe a concrete software engineering solution for resolving race conditions.

Practitioners often implement SDs with random access input buffers. Messages are then consumed correctly regardless of the order or time at which they arrive, which appears to avoid race conditions altogether. However, this approach changes the observable system behaviour from that specified. We refer to this approach as the lazy buffer realization of a SD.

We introduce an operational semantics for the lazy buffer realization. We prove the inherent causal scenario global behaviour is bisimulation equivalent to the global behaviour of lazy buffer semantics. Hence, in this sense, the practitioners solution is theoretically the best possible. Also this proves that the inherent causal scenario does represent a 'real-world' software solution

1. Introduction

Historically, 'ladder diagrams' have played a central role in defining requirements specifications for asynchronous concurrent systems. These have gradually evolved into sequence diagrams in UML 2.0 [24], and have become so expressive that property checking for languages defined by arbitrary sequence diagrams is undecidable [1]. Today international standards bodies such as ETSI and the ITU incorporate sequence diagrams as part of their protocol specification methodology and have formally specified many aspects of sequence diagram behaviour [33]. The OMG has also adopted this semantics as part of the current UML 2.0 standard. Thus, sequence diagrams are still a major component in defining the requirements specifications for commercial and government communication systems. This is despite the fact that the verification benefits of using state-based behavioural models for specifications have long been known, [10]. One reason for this lasting popularity is that practitioners find sequence diagrams more intuitive and 'easier' to understand than state machines, [30].

For this paper we will be concerned with asynchronous basic sequence diagrams that specify communication between concurrent processes within a distributed environment. Basic sequence diagrams can be given behavioural semantics in the form of a partial order between communication events. Such diagrams usually form the starting point for communication protocols and so are still a valid area of interest. The partial order restricts how events can occur in any observable trace of the system. The semantic partial order is known as the causal order of the specification. We call any basic sequence diagram with partial order behavioural semantics a partial order scenario. For brevity, we will refer to the standard partial order semantics as the causal semantics from now on. Also, we will refer to the behaviour of a partial order scenario with respect to the causal semantics as the causal behaviour. 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.

Industrial requirements specifications often contain inconsistencies between the causal behaviour and the order that events can occur in practice. Race conditions are amongst the most common of these inconsistencies. Essentially a race condition occurs when processes must act in concert to ensure the causal behaviour is correctly implemented, when in practice such coordination can not be guaranteed to occur in a distributed environment. Differences between distributed and non-distributed views of the specification are the root cause of such inconsistencies. In a non-distributed view the causal behaviour can always be imposed by the environment somehow enforcing the correct global coordination. In a distributed implementation there is no such global coordination mechanism and processes are only obliged to follow the local behaviour imposed on them by the causal order. [13, 14] give the original formal description of race conditions.

Inherent causal scenario

[21] proved that there is a unique minimal generalization of a partial order scenario that is race free, which is known as the inherent causal scenario. That paper characterises race conditions purely within a partial order theoretic framework. [21] proved there is a unique partial order that is the minimal race free generalisation of a given causal order. This unique partial order is known as the inherent causal order. A partial order in itself does not necessarily define a partial order scenario. [21] proved that the inherent causal order can be realized as a partial order scenario, which is then defined to be the inherent causal scenario.

Feedback from case studies shows that the inherent causal scenario gives engineers an overview of what concurrent behaviour can realistically be expected in a distributed environment, as opposed to the behaviour that the original scenario asserts must occur. However, practitioners tend to regard this solution as artificial since the inherent causal scenario is a mathematical construct that has no intuitive explanation in terms of software engineering constructs. Hence, practitioners are sometimes inclined to regard this solution as 'invalid' because it is not seen as a real-world software engineering solution for removing race conditions.

Practitioners also sometimes object to the additional concurrency introduced by the inherent causal scenario in order to resolve race conditions. A specification is meant to be as general as possible, and so introducing more concurrency is acceptable from a specification viewpoint. In the implementation phase additional concurrency can result in scheduling overheads and affect performance, so that practitioners often want to minimise concurrency at that later stage. The inherent causal scenario is the unique minimal generalisation that can remove all race conditions from the specification. That means it is not possible to avoid the subsequent additional concurrency if race conditions are to be resolved through generalisation. However, just because a specification contains several concurrent threads, this does not imply the implementation has to reflect that level of concurrency. An implementation is a refinement of the specification. So, it is perfectly acceptable for an implementation to be more synchronous than the specification through some domain specific refinement that makes this possible. Such as, for example, the use of coordination messages that are in addition to any existing in the specification. Implementation concerns should be separate from the specification, so that efficiency concerns related to concurrent threads should not be addressed at the requirements capture stage.

Lazy buffers

An alternative to generalising a scenario in order to resolve race conditions is to equip each process with an unbounded random access input buffer, from which messages are only consumed when needed. Hence, even if messages are delivered in the wrong order because some processes are not correctly acting in concert, the receiving process can still consume messages in the correct order from its input buffer. At first glance it appears this solves the problem without apparently introducing any additional concurrency and so it would seem to lead to an efficient implementation. This provides a pragmatic workaround for resolving race conditions that is more intuitive and much easier to explain and implement than the inherent causal order. Anecdotal evidence from work with Motorola and DaimlerChrysler suggests many engineering groups assume they can always realize sequence diagrams in this way whenever necessary. We will refer to this pragmatic approach as the lazy buffer realization for a scenario.

The lazy buffer realization is straightforward and intuitively appealing since it defines a concrete software engineering approach to avoiding race conditions altogether. However, it is not clear what global system behaviour it defines. Global behaviour is determined by the order that messages travel around the system, rather than when messages are consumed internally within a process. A sequence diagram specifies this globally observable behaviour, where the local externally observable behaviour is then a consequence of the global behaviour. The lazy buffer realization obscures the fact that the globally specified behaviour does not now correctly describe how messages travel around the system, but rather in what order messages must be consumed locally. Also it is not the case that this pragmatic approach does not introduce additional concurrency, rather it hides it so it becomes more difficult to detect. Thus system requirements analysis becomes more challenging where this approach is adopted. To properly understand how the lazy buffer realization has modified the flow of messages it would be very useful to understand how it relates to the inherent causal scenario.

Main Result

In the paper we prove that the globally observable behaviour for the lazy buffer realization of a scenario is weak bisimulation equivalent to that of the inherent causal scenario. Hence, at the global system level the inherent causal scenario describes the behaviour that is given by allowing processes to locally realize the specification through lazy input buffers. This proves that allowing processes to realize a specification through lazy buffers results in global behaviour that is the unique minimal generalisation necessary to remove all race conditions. In proving this equivalence we also prove that the lazy buffer realization defines a valid partial order scenario in its own right and therefore represents a coherent solution as well as one that is the optimal race resolution. Hence, in this particular situation, the pragmatic work-around turns out to be the best possible solution. Conversely, we also prove that the inherent causal scenario does represent a genuine ‘real-world’ software engineering solution. A practitioner can then have the best of both worlds. They may use the lazy buffer realization as the basis for an implementation, whilst still be able to explicitly see the correct global system behaviour in the form of the inherent causal scenario.

The causal semantics describes the global system behaviour of a partial order scenario that is meant to occur, but does not describe a communication semantics between processes that enables them to realize this behaviour. [11] describes such a communication semantics within a process algebra setting. This semantics generalises the original ITU standard semantics for MSCs [19, 20] to include various additional constructs. For partial order scenarios the [11] semantics is equivalent to the [19, 20] semantics and, for our purposes, is easier to reason with.

Intuitively we can summarise the communication semantics from [11] as follows. A process can not send messages directly to another process. Instead, a process can only transmit messages to a global traffic channel, T. Within the process algebra, T is a special process that behaves differently to a normal process. T can always receive messages and stores them in an unbounded random access buffer, which is represented in the form of a multiset. At the moment a process is specified to receive a message, as defined by the causal behaviour, T removes the relevant message from its buffer and sends it directly to the waiting process. Hence, T acts as a global coordination mechanism that ensures messages always arrive exactly in accordance with the causal ordering. The causal behaviour is equivalent to the globally observed behaviour given by concurrently composing a system's processes and T within the process algebra.

We define the semantics of a lazy buffer realization for a scenario by generalising the algebra in [11]. In an asynchronous distributed environment there is no global coordination mechanism, so we must adapt the communication model in [11] to reflect this. We will suppose that there is still a traffic channel T, through which all messages must be transmitted. T will still be represented as a special process. However, T will only act as a transmission medium. Within T messages can overtake one another, and have arbitrary latency. To achieve this T will store messages in a multiset as before, but will now deliver messages in an arbitrary fashion, without reference to the specification. This represents asynchronous message transmission in a distributed environment. Notice that T is still global in the sense that all messages must pass through it, but makes no attempt to coordinate messages according to the specification as do the original [19, 20] semantics.

Clearly in such an environment even if messages are transmitted as specified, they are not guaranteed to arrive in the same order unless sufficient coordination messages are in place. Each process will be given an unbounded random access input buffer, represented as a multiset within the process algebra. We will allow processes to leave received messages in the buffer until the specification states that they must be consumed. Note, this supposes messages are uniquely identifiable.

The lazy buffer realization of a specification is formally represented in the algebra by concurrent composition of the scenario processes and T (Definition 6.1). The main result of the paper, Proposition 6.5, states that the global behaviour defined by the inherent causal scenario is weak bisimulation equivalent to the global behaviour defined by the lazy buffer realization.

[21] proved that the global behaviour of a specification scenario differs from its inherent causal scenario if and only if the specification contains race conditions. Hence, Proposition 6.5 implies that lazy buffer realization and causal semantics of a scenario are the same if and only if the scenario is race free.

Structure of the Paper

In Section 3 we describe those aspects of [11] that are pertinent to the discussion here. [11] gives a process algebra semantics for almost all the constructs in a general sequence diagram, such as alternatives, timers, general orderings, subprocess creation and inline references. These are outside of the scope of this paper as we are solely concerned with partial order scenarios. Hence, we only need to be concerned with the communication aspects of the process algebra in [11] that relate to partial order scenarios. Because of this we can define a much simplified version of the process algebra in [11], which facilitates simpler bisimulation proofs later in the paper.

Section 4 discusses interrelated race conditions by way of an example from a case study and defines race conditions as a purely partial order theoretic construct. Section 5 sets out the context for optimally resolving race conditions and summarises the main proposition from [21] in order to make the paper as self contained as possible.

Section 6 defines an operational semantics for the lazy buffer realization of a partial order scenario. This includes a parallel composition operator that captures asynchronous distributed communication between processes via a traffic channel T. We then prove that the observed global behaviour of the local realization is weak bisimulation equivalent to the behaviour defined by the inherent causal scenario, Proposition 6.5. The concept of formally specifying lazy buffers is in itself nothing new, and has been around in Harel state-charts for a long time. However, specifying lazy buffer semantics directly for partial order scenarios appears to be novel. For our purposes though, we regard defining such a semantics as a means to an end and not a goal in its own right. Its value in this paper is to provide an effective formal tool that we can use for bisimulation proofs of the different semantics we consider.

Related Work

For the reader interested in an extensive in depth exposition of the current state of the art in this area we recommend two excellent papers [32, 25], which contain very good surveys. For this paper, we will give a briefer summary of related work in the area.

This paper is focussed on scenarios that describe communication protocols in an asynchronous distributed environment and their lazy buffer realization. There are many other issues relevant to the verification of protocols expressed as UML/MSC diagrams that have been studied. [1, 12, 28], amongst others, have considered verification of logical properties for languages defined by MSCs and MSC-Graphs. This work has applied model checking techniques to sequence diagrams in order to detect standard properties such as dead-lock, live-lock and ensuring trace coverage. Much of this work has been based on synthesising finite state automata from sequence diagrams. [7, 8, 17, 23, 28] consider various different compositional semantics for message sequence charts, in order to construct state machines from MSC-Graphs. Other work has considered how to interpolate missing requirements from scenario based specifications [2, 3, 4, 18, 32]. 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 realized with respect to the causal semantics.

Live sequence charts (LSCs) [15] are a variation on mainstream MSC/UML scenarios. It is possible to synthesise state machines from LSCs [16, 27, 29], just as with sequence diagrams and MSCs. One of the aims for LSCs has been to allow greater expressitivity. For example, by permitting exemplary and mandatory behaviour to be annotated directly within a scenario. At present LSCs do not have the same following in industry as they have in academia, although many of the ideas from LSCs have now found their way into UML 2.0 sequence diagrams.

Sequence diagrams are often used to capture conformance test purposes. Research into automatic test generation from partial order scenarios is another active research area [5, 6, 9, 26].

Graphical Notation

In the paper we will use UML sequence diagrams (SDs) as the graphical language for describing partial order scenarios. We will assume the reader is broadly familiar with the basic concepts of UML SDs. In this section we briefly describe the semantics for those aspects of SDs that we use in the paper. Consider the SD depicted graphically in

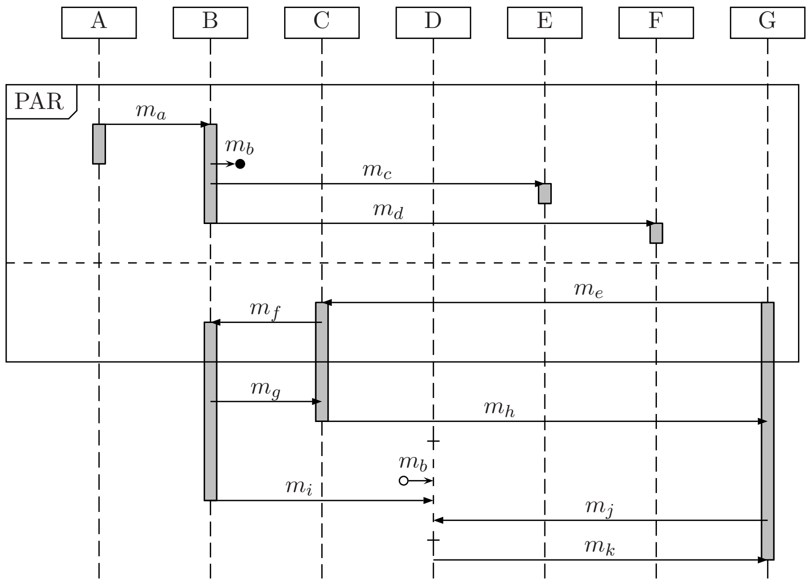

Figure 1. Inherent causal scenario resolving race conditions for Figure 5

Figure 1. 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 SDs 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 or distinct threads of a parallel construct. Within a coregion events are not locally ordered. Each coregion can only occur on a single time line. It is depicted by a short dashed line delineated by short horizontal lines. For process D in Figure 1, events $?m_b$, $?m_i$ and $?m_j$ are unordered as they occur within a coregion.

A parallel construct in an SD, denoted by keyword `PAR`, describes a set of concurrent threads that occur in the diagram. Horizontal dotted lines delineate the different threads. Hence, events from one thread are not causally ordered with respect to events from any other thread. Figure 1 contains a parallel constructs split into two threads. The first thread contains messages m_a, m_b, m_c and m_d, while the second thread contains messages m_e and m_f. 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 $!m_g$ is ordered after each of $?m_f$ and $!m_d$, 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, $!m_a$ must be before $?m_d$.

The UML notation also allows 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. So that the found event corresponding to a lost event always occur in the same scenario. Figure 1 gives an example where message m_b has been split into lost and found events. The lost event for $!m_b$ is depicted by the solid circle terminating the arrow mid-process. The found event for $?m_b$ is depicted by the empty circle that initiates an arrow into process D. 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 UML standard does not make any formal link between a lost event and a found event, that is it does not identify them as component parts of a single message. However, our convention is only used in simplifying the visual layout of our example and does not affect the theoretical results in any way.

2. Partial order scenarios

In this section we define the causal semantics for partial order scenarios. We use the same message semantics as the MSC 2000 standard [33]. 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, i.e. there is no $x \in E$ where $x < x$

  $<$ is transitive, i.e. if $x < y$ and $y < z$ then $x < z$

  $<$ is asymmetric, i.e. there are no elements x, $y \in E$ such that $x < y$ and $y < x$

– 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 $b < a$.

– For $x, y \in E$ when it is not the case that $x < y$ we write $\neg(x < y)$.

– Two elements x and y of E are unordered if $\neg(x < y)$ and $\neg(y < x)$.

We define a set to be unordered if every pair of distinct elements from that set are unordered.

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. Let E be the set of all send and receive events between all processes.

DEFINITION 2.2. A partial order scenario Sc on processes $\mathcal{P}$ is

– a collection of disjoint sets $E(P) \subseteq E$, for each $P \in \mathcal{P}$

– a set of partial orders $<_P$, where $<_P$ is a partial order on $E(P)$ and is referred to as the process order for P

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 it is possible for $P = Q$.

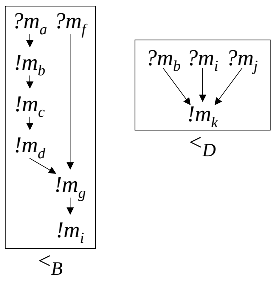

Figure 2. Examples of Process Orderings for Figure 1

For example, in Figure 1 all the process orders are total except for $<_B$ and $<_D$. These process orderings are depicted as Hasse diagrams, shown in Figure 2.

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 of pairs given by the relevant order relations. Next, we define the causal ordering that represents the behavioural semantics for a partial order scenario.

DEFINITION 2.3. The causal ordering $<_{\mathcal{C}}$ on a partial order scenario Sc is the transitive closure of the relation given by

$$\begin{array}{l}\bigcup_{P\in\mathcal{P}}(<_P) \cup \\ \{(!e,\ ?e) \mid !e \in E(P) \text{ and } ?e \in E(Q) \text{ for some } P,\ Q \in \mathcal{P}\}\end{array}$$

The set of pairs $(!e,\ ?e)$ is used to assert that orderings between processes can only be a consequence of message exchanges. Hence, the causal ordering combines process orderings solely through the causality between send and receive event pairs. As an example, Figure 3 depicts the causal ordering for the sequence diagram in Figure 1.

Note, it is possible for there to be two events x and y, both in the same process P, where $x <_{\mathcal{C}} y$ but $\neg(x <_P y)$. Without loss of generality we will assume this is not the case from now on. That is, when $x, y \in E(P)$, we assume $x <_C y$ if and only if $x <_P y$. This is acceptable within our context as we want to study the externally observable system behaviour and will never need to consider when the process ordering is not the same as the causal ordering. Hence, if we are given a causal ordering it will be straightforward to extract the process orderings from it.

$$\begin{tikzcd}
?m_a \arrow[d] & !m_a \arrow[l] & ?m_e \arrow[d] & !m_e \arrow[l] & \\
!m_b \arrow[r] \arrow[d] & ?m_b \arrow[ddddr] & !m_f \arrow[d] & & \\
!m_c \arrow[r] \arrow[d] & ?m_c & ?m_f \arrow[d] & & \\
!m_d \arrow[rr] \arrow[d] & & !m_g \arrow[r] \arrow[dl] & ?m_g \arrow[r] & !m_h \arrow[d] \\
?m_d & !m_i \arrow[d] & & & ?m_h \arrow[d] \\
 & ?m_i \arrow[r] & !m_k \arrow[d] & ?m_j \arrow[l] & !m_j \arrow[l] \\
 & & ?m_k & &
\end{tikzcd}$$

Figure 3. Causal Order for Figure 1

DEFINITION 2.4. For a causal ordering $<_{\mathcal{C}}$, a system trace is a total order extension of $<_{\mathcal{C}}$. For a process $P \in \mathcal{P}$ with process order $<_P$, a trace of P is a total order extension of $<_P$.

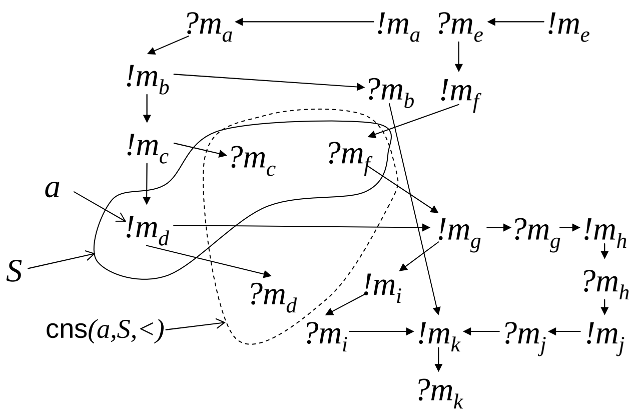

Figure 4. Examples of S and $\mathsf{cns}(a, S, <)$ for Figure 3

3. Causal Communication Semantics

In this section we define an elementary process algebra term $P(\mathsf{Sc})$ that defines the causal behaviour of a scenario Sc up to bisimulation equivalence. In order to define $P(\mathsf{Sc})$ we first have to set up some notation.

DEFINITION 3.1. For a set $S \subseteq E$ and partial order $<$ on E define

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

Intuitively we can think of $\mathsf{n}(S, <)$ as the 'next' elements in E after the set S according to the partial order $<$. $\mathsf{m}(S, <)$ is the set of minimal events in S with respect to $<$. Notice that $\mathsf{cns}(a, S, <)$ is an unordered set, since the minimal elements of a set are themselves always unordered. cns is an abbreviation for consecutive. 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 $b \in \mathsf{cns}(a, S, <)$, (Lemma 4.2 of [21]). Therefore, 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.

Consider a partial trace of events for the scenario in Figure 1 where messages m_a, m_b, and m_e have been sent and received, and where message m_f and m_c have been sent and not yet received. The events that are eligible to occur next are given by S as shown in Figure 4. Suppose that $!m_d$ occurs next in the trace. In that case $?m_d$ could consecutively follow $!m_d$ in the trace, whereas $!m_g$ could not. The reason for this is that although $?m_d$ is a trigger for $!m_g$ so is $?m_f$, which has not yet occurred in the trace. Hence the events that are eligible to occur consecutively with $!m_d$ are $?m_c$, $?m_d$ and $?m_f$. That is, in this example $\mathsf{cns}(a, S, <) = \{?m_c, ?m_d, ?m_f\}$. These events are shown in Figure 4 enclosed by the dashed line.

We can now define a recursive process algebra term that defines the causal behaviour of a scenario Sc.

DEFINITION 3.2. For a set $S \subseteq E$ and partial order $<$ on E define

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

and $P(\emptyset, <) = 0$. Where $\cdot$ denotes action prefix and $\sum$ denotes the usual choice operator for a process algebra term.

Let Sc be a partial order scenario over processes $\mathcal{P}$, with causal order $<_{\mathcal{C}}$. Define

$$P(\mathsf{Sc}) = P(\mathsf{m}(E, <_{\mathcal{C}}), <_{\mathcal{C}})$$

This description is equivalent to the descriptions in [1, 11] of process algebra characterisations of the causal semantics for a partial order scenario. [21] proved (Lemma 4.5) that this particular description does indeed characterise the correct traces for Sc. That is the traces for $P(\mathsf{Sc})$ are exactly the system traces of Sc (see Definition 2.4).

4. Race Conditions

A race condition represents a semantic inconsistency between the specified order that events are meant to occur in and the actual order that it is possible for them to occur in. In our context 'possible' is interpreted within a distributed asynchronous environment. Within a partial order theoretic framework we can define a race formally as follows.

DEFINITION 4.1. Define a partial order $<$ on E to be race free when for every event x and message e:

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

A partial order scenario is race free when its causal order is race free.

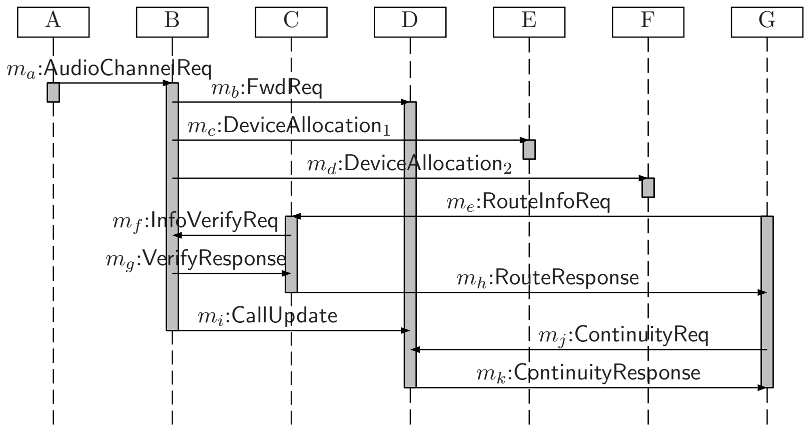

Figure 5. Motorola example containing race conditions

From this definition we see a race occurs when there is an event x and a message e such that $x < ?e$ and $x \not< !e$. That is two processes are meant to be coordinating to ensure that x occurs before e is received. In a distributed asynchronous environment that implies x occurs before $!e$. Otherwise $!e$ can be sent without reference to x, and that could mean $?e$ arrives earlier than x, contradicting the causal order. Figure 5 depicts a UML 2.0 sequence diagram taken from a Motorola proprietary case study. The study looked at approximately one hundred sequence diagrams that were part of a 3G protocol stack specification, of which fifteen contained multiple interrelated race conditions. The diagram is anonymized to protect proprietary information. In the discussion we will use the short form of message names from the sequence diagram for brevity. Altogether there are seven races in the scenario. Receive event $?m_f$ is in a race with each of $?m_a$, $!m_b$, $!m_c$ and $!m_d$. For example, $!m_d <_C ?m_f$ but $\neg(!m_d <_C !m_f)$. Thus process C must ensure that m_f is received after m_d is sent, and yet there is no coordination between processes that can enforce this. The remaining three races occur between each of $?m_b$, $?m_i$ and $?m_j$.

This is an interesting example in that it illustrates how multiple race conditions can be interrelated. Attempting to resolve each of the race conditions independently in a piecemeal fashion would not necessarily lead to a satisfactory solution. For example, message m_f is independent of m_d (in that it can be sent at any time after $?m_e$, and m_e is one of the initial messages in the scenario). Resolving the race between $?m_f$ and $!m_d$ by itself would not resolve the races between $?m_f$ and $?m_a$, $!m_b$ and $!m_c$. Rather it would complicate the process of identifying a solution to the other races.

5. Optimal Race Resolution

In this section we give a brief introduction to the inherent causal scenario defined in [21]. Also, we explain how the inherent causal scenario defines a canonical race free generalisation for partial order scenarios.

Faced with the complex set of race conditions exemplified by Figure 5, a natural question to ask is whether there is always a non-trivial generalisation of a scenario that removes race conditions. In the partial order setting for this paper that translates into the question:

> Let Sc be a partial order scenario with events E, processes $\mathcal{P}$ with process orders $<_P$ for $P \in \mathcal{P}$ and causal order $<_\mathcal{C}$. Is there a non-trivial partial order scenario over the same set of events and processes with a race free causal order $<_{\mathcal{C}'}$ that is a generalisation of $<_\mathcal{C}$.

Partial order $<_1$ is a generalisation of $<_2$ if, when regarding a partial order as a set of pairs, $(<_1) \subseteq (<_2)$. That is $<_1$ has weakened some of the constraints defined by $<_2$. This question motivated the research in [21]. [21] proved there is a unique minimal race free generalisation of a partial order scenario, which is referred to as the inherent causal scenario. In addition, [21] proves that any race free partial order scenario that simulates the behaviour of a partial order scenario must also simulate the behaviour of its inherent causal scenario. So that the inherent causal scenario is unique up to simulation equivalence.

In order to construct the inherent causal scenario, first a particular partial order known as the inherent causal ordering is constructed. The inherent causal order is generalised from the causal order of a scenario as explained in the following definition.

DEFINITION 5.1. 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. $!e < ?e$

The inherent causal scenario is the realisation of the inherent causal order in the form of a partial order scenario. The inherent causal order is race free in the sense of Definition 4.1. It is also the unique minimal generalisation of $<_\mathcal{C}$ that is race free.

Returning to the example in Figure 5, the inherent causal scenario for this is given in Figure 1. Thus, Figure 1 represents the unique minimal race free generalisation of Figure 5. Note race conditions have been resolved by introducing additional concurrency into the original scenario. Message flows have been split into concurrent threads within a parallel construct to prevent them racing against each other. As mentioned in the introduction, we have used the lost and found construct as a visual convenience to split m_b into two parts.

The partial order given in Figure 3 is therefore the inherent causal order of Figure 5, as well as being the causal order of Figure 1. Note, although the inherent partial order scenario is unique, its graphical representation is not. Thus, Figure 1 is only one way of graphically depicting the behaviour of the inherent causal scenario as a UML sequence diagram.

Finally in this section we quote the result from [21] that formally describes how the inherent causal scenario gives a canonical race free generalisation of a partial order scenario, up to simulation equivalence.

DEFINITION 5.2. The inherent process behaviour of a partial order scenario Sc is defined to be

$$P_I(\mathsf{Sc}) = P(\mathsf{m}(E, <_I), <_I)$$

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 $P' \sqsupset Q'$

PROPOSITION 5.3. [Theorem 7.2 [21]]

1. $<_I$ is race free.
2. $(<_I) \subseteq (<_{\mathcal{C}})$, and $P_I(\mathsf{Sc}) \sqsupset P_c(\mathsf{Sc})$
3. For any race free partial order $<$ that preserves message ordering, let $P_< = P(\mathsf{m}(E, <), <)$. Then $P_< \sqsupset P(\mathsf{Sc})$ iff $(<) \subseteq (<_I) \subseteq (<_{\mathcal{C}})$ and $P_< \sqsupset P_I(\mathsf{Sc})$

That is $P_I(\mathsf{Sc})$ is the canonical process that simulates $P_c(\mathsf{Sc})$ and is race free.

6. Lazy Buffer Realization of Scenarios

For this section the purpose of a scenario specification is to show how processes must behave locally, rather than what global behaviour must be enforced. Rather than assume a specification scenario contains adequate coordination to ensure there are no race conditions, in this section we assume that a receiving process must determine in what order messages were actually sent, or at least in what order incoming messages should be consumed. A simple mechanism to achieve this is to provide each process with its own unbounded random access buffer, where incoming messages are stored until they are needed.

In this section we describe a modified form of the structural semantics given in [11] that defines our lazy buffer realization for a partial order scenario. Our goal is to construct a model where we can formally reason about how this buffer mechanism affects the observable global system behaviour. Hence, our model must directly describe how messages are delivered to a buffer and then consumed from the buffer.

Let Sc be a partial order scenario over processes $\mathcal{P}$. For $P \in \mathcal{P}$ we will define a process term $\mathsf{Pr}(P)$, which together with a parallel composition operator $\|_d$, defines the lazy buffer realization for Sc (where the subscript d denotes that $\|_d$ is a *d*istributed form of concurrent composition).

As in [11], processes do not transmit messages directly to one another. Instead, there is a special transmission channel process, T, where all messages are sent. Unlike the transmission channel in [11] T will now simply transfer messages in an arbitrary fashion to simulate asynchronous message passing. It will be assumed that the messages are always delivered to the correct process. For convenience we assume messages have unique labels that identify which process they can be delivered to.

A process term $\mathsf{Pr}(P)$ is a triple $\mathsf{Pr}(In, S, <)$, where In is an input buffer multi-set, S is a set of concurrent events that are eligible to occur next in a trace for the process, and $<$ is the partial order that constrains what order events occur in for that process. The distributed behaviour for a process $P \in \mathcal{P}$ is defined to be $\mathsf{Pr}(\emptyset, \mathsf{m}(E(P), <_P), <_P)$. Here the input buffer is initialised as empty, the initial set of events are the minimal events in P with respect to the process order for P, and of course the process order for P defines how trace events are constrained. The buffer mechanism splits the act of receiving a message into message delivery followed later by message consumption.

In order to have a simple notation to track which processes have finished and which are still active we introduce a primitive term $\mathtt{End}(P)$

Receive $$\frac{\phi = \mathsf{Pr}(In, S, <_P)}{\phi \xrightarrow{?e} \mathsf{Pr}(In \cup \{?e\}, S, <_P)} \; ?e \in E(P)$$

Consume $$\frac{\phi = \mathsf{Pr}(In \cup \{?e\}, S \cup \{?e\}, <_P)}{\phi \xrightarrow{\tau} \mathsf{Pr}(In, \mathsf{cns}(?e, S, <_P), <_P)}$$

Send $$\frac{\phi = \mathsf{Pr}(In, S \cup \{!e\}, <_P)}{\phi \xrightarrow{!e} \mathsf{Pr}(In, \mathsf{cns}(!e, S, <_P), <_P)} \; !e \notin S$$

Transmit $$\frac{\phi \xrightarrow{!e} \phi'}{T(B) \parallel_d \phi \xrightarrow{!e} T(B \cup \{?e\}) \parallel_d \phi'}$$

Deliver $$\frac{\phi \xrightarrow{?e} \phi'}{T(B \cup \{?e\}) \parallel_d \phi \xrightarrow{?e} T(B) \parallel_d \phi'}$$

Terminate $\mathsf{Pr}(\{\ \}, \{\ \}, <_P) = \mathtt{End}(P)$

Figure 6. Lazy Buffer Realization Semantics for Partial Order Scenario

for each $P \in \mathcal{P}$. This term can not perform any action and represents a parameterised version of the 0 process.

DEFINITION 6.1. Let τ denote the silent action. Define the distributed composition operator $\parallel_d$ to have the operational semantics given in Figure 6. Further, $\parallel_d$ is defined to be associative and commutative. For each $P \in \mathcal{P}$, define

$$\mathsf{Pr}(P) = \mathsf{Pr}(\emptyset, \mathsf{m}(E(P), <_P), <_P)$$

Define the lazy buffer realization of Sc to be

$$P_d(\mathsf{Sc}) = T(\emptyset) \parallel_d \mathsf{Pr}(P_0) \parallel_d \cdots \parallel_d \mathsf{Pr}(P_n)$$

Notice that in Definition 6.1 we only define $\parallel_d$ for composition between primitive process terms and traffic channel T. This is sufficient for our purposes of discussing the behaviour of partial order scenarios. The Receive, Consume and Send rules define how processes handle message passing. The Transmit and Deliver rules define how the transmission channel T takes messages from processes, places them in its buffer B and at some arbitrary time later delivers them to the input buffer of the appropriate process.

The Receive rule states that a process can accept any receive event at any point during execution, as long as the event belongs to that process. A process initially places any received event into its input buffer. Suppose that the process term for P is $\mathsf{Pr}(In, S, <_P)$ at some point during its execution. The set S defines concurrent events that are eligible to occur next in the execution of P. The Send rule allows an event $!e$ to be sent by P whenever $!e$ is one of the events in S irrespective of what events are in buffer In.

The Consume rule states that an event $?e$ can be taken from the input buffer In if and only if that is also an element of S. That is an event can only be consumed from the input buffer when that event is one of the concurrent events that is eligible to occur next according to the process order for P. Since In is a multiset the order that events are consumed from it are solely determined by S and $<_P$. We have modelled message consumption as an internal action by using τ to describe the external action that will be observed when messages are consumed. The Consume rule is the only rule that defines an interaction between S and In. It is the Consume rule that defines how processes adopt a lazy approach to internally consuming message events.

The Transmit rule states that whenever a send event $!e$ is executed by a process then T will store the corresponding receive event $?e$ in the transmission buffer B. The Deliver rule states that T can remove an event from its buffer and place it in the input buffer of a process whenever that process is able to accept such an event. Since the Receive rule allows a process to accept a receive event at any time this implies T can deliver messages in an arbitrary manner.

Note that once an event a occurs (and is consumed in the case of a receive event) then $\mathsf{cns}(a, S, <_P)$ defines the set of events that can be consecutive to a in a trace of P. This is analogous to the causal semantics given in Definition 3.2. Hence, the set S will always contain concurrent events that are consistent with $<_{\mathcal{C}}$.

Each of the $\mathsf{Pr}(P)$ processes locally realizes the causal order $<_{\mathcal{C}}$ in the sense that messages can only be consumed in the order given by $<_P$, as opposed to attempting to enforce the causal order $<_{\mathcal{C}}$. Also a message $!e$ is sent only when the correct stimuli have been consumed in the correct order from the input buffer.

We claimed in the introduction that by realizing a specification with lazy buffers we resolve race conditions. Race conditions are a problem as they can result in deadlocks. For example, a process that receives messages in the wrong order can deadlock because it is not able to processes the current message until after it has consumed the following message, but it is not able to consume the next message until it has processed the current message. We prove in Proposition 6.3 that lazy buffer realization can not deadlock. This demonstrates that our semantics enables processes to continue to act and consume messages with respect to the local process ordering even when events are not delivered correctly at the global level.

DEFINITION 6.2. Let Sc be a partial order scenario over processes P_i for $1 \leq i \leq n$. Let $\mathtt{End}_i = \mathtt{End}(P_i)$. Write $Q_0 \xrightarrow{\alpha}_{\star} Q_m$, where $\alpha = a_1 \cdot a_2 \cdots a_m$, when there are processes Q_i and transitions $Q_i \xrightarrow{a_i} Q_{i+1}$ for $0 \leq i \leq m-1$.

Define a sequence of events $\beta = b_1 \cdot b_2 \cdots b_k$ to be a deadlock trace for $P_d(\mathsf{Sc})$, if there exists a process $Q \neq \mathtt{End}_0 \parallel_d \cdots \parallel_d \mathtt{End}_n$ where $P_c(\mathsf{Sc}) \xrightarrow{\beta}_{\star} T(B) \parallel_d Q$, for some buffer B, and $T(B) \parallel_d Q$ can not perform any action according to the rules in Figure 6.

PROPOSITION 6.3. Let Sc be a partial order scenario over processes P_i for $1 \leq i \leq n$.

$$P_d(\mathsf{Sc}) \text{ has no deadlock traces.}$$

Proof

Suppose there is a partial trace $\beta = b_1 \cdots b_k$ of $P_d(\mathsf{Sc})$. Then we can suppose there are terms
$\phi_i^k = \mathsf{Pr}(In_i^k, S_i^k, <_{P_i})$, and
$\psi_k = T(B_k) \parallel_d \phi_0^k \parallel_d \cdots \parallel_d \phi_n^k$ where

$$P_d(\mathsf{Sc}) \xrightarrow{\beta}_{\star} \psi_k$$

That is each ϕ_i^k is the process that P_i has become once $P_d(\mathsf{Sc})$ has transformed into ψ_k. B_k contains the events that are still to be delivered at the end of the trace. Let E_R denote the set of receive events in the set of all events E.

Although $T(B)$ is constrained by the operational semantics of Definition 6.1 it can always deliver any messages remaining in B. Also, any process ϕ_i^k can send a message to T so long as there is some send event in S_i^k. Thus a deadlock can occur if and only if $B_k = \emptyset$ and

$$\forall\, 0 \leq i \leq n.\ (S_i^k \subseteq E_R) \ \text{ and } \ (S_i^k \cap In_i^k = \emptyset)$$

Let

$$?e \in \mathsf{m}\Big(\bigcup_{0 \leq i \leq n} S_i^k, <_{\mathcal{C}}\Big)$$

and suppose that $?e \in S_i^k$ for some i, and that $!e \in E(P_j)$ for some j. If $!e$ has not already occurred in β this can only be because there is some $x \in S_j^k$ where $x <_{P_j} !e$. This contradicts that $?e$ is minimal. Hence, $!e = b_r$ for some $1 \leq r \leq k$. Since $B_k = \emptyset$ this can only be true if $?e \in In_i^k$, which is a contradiction. This completes the proof. □

DEFINITION 6.4. For a set of events X, let $!X$ be the set of all send events contained in X, and let $?X$ be the set of all receive events in X. Given a term P'_d of the form

$$T(B') \parallel_d \mathsf{Pr}(P_1)' \parallel_d \cdots \parallel_d \mathsf{Pr}(P_n)'$$

where $\mathsf{Pr}(P_i)' = \mathsf{Pr}(In'_i, S'_i, <_{P_i})$ for $1 \leq i \leq n$, let

$$S(P'_d) = B' \cup \bigcup_{1 \leq i \leq n} !S'_i$$

This set will be used later in the proof of the main result of the paper Proposition 6.5.

When P'_d represents the system state at some point, then $S(P'_d)$ represents the set of send events in all the system processes and all the receive events in the transmission buffer that are eligible to occur next.

We can now state and prove the main result of the paper. The globally observable behaviour defined by the lazy buffer realization for Sc is weak bisimulation equivalent to the global behaviour defined by the inherent causal scenario for Sc.

PROPOSITION 6.5. Let Sc be a partial order scenario, and let $\simeq$ denote weak bisimulation equivalence. Then

$$P_I(\mathsf{Sc}) \simeq P_d(\mathsf{Sc})$$

Recall that the process term $P_I(\mathsf{Sc})$ on the left is defined in terms of the causal semantics of Definition 3.2. $P_I(\mathsf{Sc}) = P(\mathsf{m}(E, <_I), <_I)$, which defines the globally observable behaviour of the inherent causal scenario.

Proof

Let Sc be a partial order scenario over processes $\mathcal{P} = \{P_i \mid 1 \leq i \leq n\}$. So we may write

$$P_d(\mathsf{Sc}) = T(\emptyset) \parallel_d \mathsf{Pr}(P_1) \parallel_d \cdots \parallel_d \mathsf{Pr}(P_n)$$

Let $P_d = P_d(\mathsf{Sc})$ and let

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

To prove weak bisimulation equivalence we prove the following stronger result. Let $\alpha = a_0 \cdots a_k$ be a sequence of events from E. We will prove there exists

$$P'_d = T(B') \parallel_d \mathsf{Pr}(P_1)' \parallel_d \cdots \parallel_d \mathsf{Pr}(P_n)'$$

where $\mathsf{Pr}(P_i)' = \mathsf{Pr}(In'_i, S'_i, <_{P_i})$ for $1 \leq i \leq n$ and

$$P_d \xrightarrow{\quad\alpha\quad}_\star P'_d$$

if and only if there exists some

$$P'_I = P(S', <_I)$$

where $S' \subseteq E$ such that $P_I \xrightarrow{\quad\alpha\quad}_\star P'_I$ and where

$$S(P'_d) = S'$$

See Definition 6.4 for the definition of $S(P'_d)$. Once all the events in α have occurred, the set $S(P'_d)$ represents all the send events that are permitted to concurrently occur next in each process in Sc and all the current receive events in the buffer for the transmission channel. A corollary of the proof hypothesis is that $P_d(\mathsf{Sc})$ and P_I can always perform the same traces, and at every point during the execution of a trace they contain exactly the same events that are eligible to concurrently occur next. We are not interested in process receive events as they are consumed internally, only receive events delivered from the transmission buffer will be externally observable.

Induction statement

This stronger statement can be proved by induction on the length of α. The base case where the length of $\alpha = 0$ is trivial so we can move on to the induction step.

For the induction step we can assume the above is true, and we need to show that for any a, there is a transition $P'_d \xrightarrow{a} P''_d$ if and only if there is some transition $P'_I \xrightarrow{a} P''_I$ where P''_d is related to P''_I as above. From the definitions for the relavent process terms it is straightforward to prove that $P'_d \xrightarrow{a} P''_d$ if and only if there is some transition $P'_I \xrightarrow{a} P''_I$. The difficulty lies in showing the rest of the induction hypothesis holds. The proof now splits into two cases depending on whether a is a send or receive event.

Case a =!e

Consider first where $a =!e$, and suppose $P'_d \xrightarrow{!e} P''_d$. Let

$$P''_I = P(S'', <_I)$$

where $S'' = \mathsf{cns}(!e, S', <_I)$. To prove $S'' = S(P''_d)$ we will show that $?S'' =?S(P''_d)$ and $!S'' =!S(P''_d)$, which we will prove in that order.

By the induction hypothesis we may suppose that $!e \in S'_r$ for some r. Let $S'_r = \{!e\} \cup S''_r$. We can then write P'_d in the form

$$P'_d = T(B') \parallel_d \mathsf{Pr}(In'_r, \{!e\} \cup S''_r, <_{P_r}) \parallel_d Q$$

where Q is just a place holder to collect together all the other $\mathsf{Pr}(P_i)'$ terms. Therefore there is a transition

$$\begin{aligned} &P'_d \xrightarrow{!e} \\ &\quad T(B' \cup \{?e\}) \parallel_d \mathsf{Pr}(In'_r, \mathsf{cns}(!e, S'_r, <_{P_r}), <_{P_r}) \parallel_d Q \end{aligned}$$

Let $B'' = B' \cup \{?e\}$, and $(<_r) = (<_{P_r})$. From the definition for $<_I$ it is immediate that $?e \in \mathsf{cns}(!e, S', <_I)$, since all of the elements in $S' - \{!e\}$ are disjoint from $!e$ with respect to $<_I$. This means the only new receive event in S'' that was not already present in S' is $?e$. Hence, B'' is the set of receive events for S'', which implies $?S'' =?S(P''_d)$.

It remains to prove that $!S'' =!S(P''_d)$. We first prove that $!S'' \subseteq !S(P''_d)$. We need to consider what new send events are present in S'' that are not elements of S'. It will be enough to prove that for any new send event $!g \in S''$ then $!g \in \mathsf{cns}(!e, S'_r, <_r)$. By inspection any new send events in S'' must come from the set $\mathsf{n}(!e, <_I)$.

From the definition of the causal ordering $<_\mathcal{C}$ it follows that

$$\mathsf{n}(!e, <_I) \subseteq \{?e\} \cup \{x \in E(P_r) \mid !e <_\mathcal{C} x\}$$

Consider the case where $?g \in \mathsf{n}(!e, <_I)$. By definition that means $!e <_I ?g$, and there is no x such that $!e <_I x <_I ?g$. From the definition of $<_I$ that can only hold if $?g =?e$. If this were not the case then the inequality could only hold if $!e <_I ?e <_\mathcal{C} !g <_I ?g$, which contradicts that $?g \in \mathsf{n}(!e, <_I)$. Therefore, $\mathsf{n}(!e, <_I)$ consists of only one receive event $?e$ plus some number of send events. Since $?e \in \mathsf{cns}(!e, S', <_I)$, we know $\mathsf{cns}(!e, S', <_I) - S'$ consists of $?e$ plus a number of send events.

Consider when $!g \in \mathsf{n}(!e, <_I)$. Then $\neg(?e <_\mathcal{C} !g)$, for otherwise $!g \notin \mathsf{n}(!e, <_I)$. Therefore, we have $!e <_r !g$. Further there is no event $x \in E(P_r)$ such that $!e <_r x <_r !g$. Hence $!g \in \mathsf{n}(!e, <_r)$.

That is any new send event in S'' that is not already in S' must actually be an element of $\mathsf{n}(!e, <_r)$.

A new send event $!g$ in S'' must also be an element of $\mathsf{cns}(!e, S', <_I)$. Note, any such $!g$ is also minimal in $S' - \{!e\}$ with respect to $<_I$. Let $!g$ be a new send event in S'', so that $!g \in \mathsf{cns}(!e, S', <_I) \subseteq \mathsf{n}(!e, <_I)$, and hence $!g \in \mathsf{n}(!e, <_r)$. To complete the proof that $!S'' \subseteq !S(P''_d)$ we only need to prove that $!g$ is also minimal in $S' - \{!e\}$ with respect to $<_r$, since that will complete the proof that $!g \in \mathsf{cns}(!e, S'_r, <_r)$.

For a contradiction, consider if there is some element $y \in S'_r$ where $y <_r !g$, and $y \neq !e$. We know y is not a send event. If this were not the case, then $y \in !S'_r \subseteq S'$. That would imply $!g \notin \mathsf{cns}(!e, S', <_I)$, which is a contradiction. Therefore, we may assume $y = ?f$ for some f. Since $?f \in E(P_r)$ this event has not yet occurred. Therefore there is some event $z <_C ?f$, where

$$z \in B' \cup \bigcup_{j \neq r} S'_j$$

Hence, $z <_I !g$, $z \in S'$ and z is distinct from $!e$. Therefore we have a contradiction since this again implies $!g \notin \mathsf{cns}(!e, S', <_I)$. This proves that $!g \in \mathsf{cns}(!e, S'_r, <_r)$. Hence, in the case where $a = !e$ we have shown that

$$!S'' \subseteq S(P''_d)$$

The proof that $!S''_r \subseteq S''$ is analogous so that we have proved $S'' = S(P''_d)$. This completes the proof of the induction step for the case $a = !e$.

Case a $=$?e

Consider next the case where $a = ?e$ for some message e. We follow the same pattern as for the previous case in that we prove the receive events for S'' and $S(P''_d)$ are the same, and then that their send events are the same. We may write P''_d in the form

$$P''_d = T(B'') \parallel_d \mathsf{Pr}(P_1)'' \parallel_d \cdots \parallel_d \mathsf{Pr}(P_n)''$$

where $\mathsf{Pr}(P_i)'' = \mathsf{Pr}(In''_i, S''_i, <_{P_i})$.

From the definition for $<_I$ it follows that for $x \in E$, $?e <_I x$ if and only if one of the two conditions below hold

- There is some $g \in E$ such that $x = !g$ and $?e <_C !g$
- There is some $g \in E$ such that $x = !g$ and

$$?e <_C !g <_C ?g$$

Note the second item is true if and only if $?e <_I \ !g <_{\mathcal{C}} \ ?g$. Whichever case applies, it follows that every element of $\mathsf{n}(?e, <_I)$ is a send event in E.

The induction hypothesis implies that $?e \in B'$, and hence there is a transition $P'_I \xrightarrow{?e} P''_I$. We can also assume without loss of generality that $?e$ is consumed from the relevant input buffer at this point, if that is permitted by the process order. Hence, we have proved that the receive events in S'' are exactly the events in $B'' = B' - \{?e\}$, which implies that $?S'' = ?S(P''_d)$.

What remains to be proved is that $!S'' = !S(P''_d)$. To prove that it is enough to prove the send events in S'' are the union of all the $!S''_i$. First we prove that $!S''_j = !\mathsf{cns}(?e, S'_j, <_{\mathcal{C}}) \subseteq \mathsf{cns}(?e, S', <_I)$, where $?e \in E(P_j)$.

Let $!g \in \mathsf{n}(?e, <_j)$, which implies $?e <_j \ !g$, and hence $?e <_I \ !g$. Also notice that

$$\forall x \in E(P_j).\ \neg(?e <_j x <_j \ !g)$$

Given that we know $u <_I v \Rightarrow u <_{\mathcal{C}} v$ we can prove

$$\forall x \in E(P_j).\ \neg(?e <_I x <_I \ !g)$$

To prove this, assume for a contradiction that there is some $x \in E - E(P_j)$ where

$$?e <_I x <_I \ !g$$

This implies $?e <_{\mathcal{C}} x <_{\mathcal{C}} !g$ since $(<_I) \subseteq (<_{\mathcal{C}})$. However, it then follows that there is some $?h \in E(P_j)$ such that $x <_{\mathcal{C}} ?h <_j \ !g$. Therefore, $?e <_{\mathcal{C}} \ ?h <_j \ !g$, which implies $?e <_j \ ?h <_j \ !g$ and we have a contradiction as required.

Thus, we have proved $!g \in \mathsf{n}(?e, <_I)$. A similar argument also shows that

$$!g \in \mathsf{cns}(?e, S', <_I)$$

Hence,

$$!\mathsf{cns}(?e, S'_j, <_{\mathcal{C}}) \subseteq \mathsf{cns}(?e, S', <_I)$$

and so we have proved that $!S''_j \subseteq S''$. Since none of the other processes have changed during this transition it is still true that for $i \neq j$, $!S'_i \subseteq S''$. The converse, that $!S''$ is contained in the union of all the $!S''_i$ for $1 \leq i \leq n$, is analogous and so we omit the details. This completes the proof that $!S'' = !S(P''_d)$, and so we have proved $S'' = S(P''_d)$.

That completes the induction step and so completes the proof of the bisimulation equivalence.

□

As an illustration of Proposition 6.5, we can return to Figures 5 and 1. If the scenario in Figure 5 is realized according to the lazy buffer semantics of Definition 6.1, then the behaviour that would be externally observable is that shown in Figure 1. Process D, for example, is specified in Figure 5 to realize a total order on its events. However, this behaviour can not be realized within a distributed asynchronous environment. The actual behaviour that would be observed is given by the process partial order for D in Figure 2. As we can see from Figure 2, in a distributed environment this linear specification would be locally realized as three independent receive events followed by a final send event.

7. Conclusion

Practitioners use scenario specification languages like UML or MSC to rapidly develop requirements specifications. Frequently they use these languages in a semi-formal manner. Scenarios sometimes tend towards specific examples that contain domain specific assumptions instead of defining exemplary generalisations. This can lead to specifications that are semantically inconsistent at the global level of process coordination. Such discrepancies can result in race conditions.

The inherent causal scenario and lazy buffer realization defined in the paper represent two orthogonal views of how race conditions may be resolved. The inherent causal scenario generalises a specification by just enough to remove race conditions. Thus, the inherent causal scenario generalises a specification to ensure that the actual observed global behaviour will match the specified behaviour. However, the inherent causal ordering is difficult to relate to standard software engineering constructs, which can make it difficult for practitioners to accept.

The lazy buffer realization avoids race conditions by giving each process enough autonomy to locally behave correctly with respect to the specification, even though that may result in global behaviour that does not match the specification. The reason for this is that although the lazy buffer realization ensures processes do correctly store and consume messages as specified, this is not observable externally. Message consumption is an internal activity. From an external perspective, the observable behaviour of a process is determined solely by when a message is delivered to the process. Hence, with respect to externally observable behaviour, the lazy buffer realization appears to allow processes to ignore the specified order messages are meant to arrive in when necessary. The lazy buffer realization is appealing since it uses a straightforward communication semantics and it is intuitively clear why it correctly resolves race conditions. From anecdotal evidence, this seems to be a commonly used realization of a sequence diagram.

The paper has proved these two apparently different approaches are, at the globally observable level, the same. Hence, realizing a scenario with lazy buffers defines the unique minimal generalisation of the scenario to remove all race conditions. Since the inherent causal scenario is itself a partial order scenario, we have also proved that the lazy buffer realization is equivalent to a partial order scenario. This demonstrates that the lazy buffer realization does represent a coherent semantic solution to race avoidance. Therefore, in this particular sense, the lazy buffer pragmatic approach is the theoretically best possible solution to resolving race conditions. The proof is an equivalence and so we have also proved that the inherent causal scenario is a genuine 'real-world' software engineering solution to resolving race conditions.

References

1. Alur, R. and Yannakakis, M. 1999, Model checking of message sequence charts, Proceedings of the Tenth International Conference on Concurrency Theory, LNCS 1661, Springer, pp 114–129.
2. Alur, R. and Etessami, K. and Yannakakis, M. 2000, Inference of Message Sequence Charts, Proceedings 22nd International Conference on Software Engineering, pp 304-313.
3. Alur, R. and Etessami, K. and Yannakakis, M.2001, Realizability and verification of MSC graphs, Proceedings of the 28th International Colloquium on Automata, Languages, and Programming, Springer, New York, pp 797-808.
4. Ben-Abdhallah, H. and Leue, S. 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.
5. Baker, P. and Bristow, P. and Jervis, C. and King, D. and Mitchell, B. 2002, Automatic Generation of Conformance Tests From Message Sequence Charts, Proceedings of 3rd SAM Workshop, Telecommunications and Beyond: The Broader Applicability of MSC and SDL, LNCS 2599, Springer, pp 170-198.
6. Beyer, M. and Dulz, W. and Zhen, F. 2003, Automated TTCN-3 Test Case Generation by Means of UML Sequence Diagrams and Markov Chains, Proceedings of 12th Asian Test Symposium (ATS'03), IEEE, pp 102–106.
7. Bontemps, Y. and Schobbens, P. 2003, Synthesis of Open Reactive Systems from Scenario-Based Specifications, Third International Conference on Application of Concurrency to System Design (ACSD'03), IEEE, pp 41-50.
8. Bontemps, Y. and Heymens, P. 2002, Turning high-level live sequence charts into automata, Proc of Scenarios and State Machines: Models Algorithms and tools, 24th International Conf. on Software Engineering, ACM.

9. Chung, S. and Kim, H. S. and Seop Bae, H. and Kwon, Y. R. and Lee, B. S. 1999, Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems, pp 72 – 82.
10. Clarke, E. and Andwing, J. 1996, Formal methods: State of the art and future directions. ACM Computing Surv., 4(28), pp 626–643.
11. Gehrke, T. and Hilhn, M. and Wehrkeim, H. 1998, An Algebraic Semantics for Message Sequence Chart Documents, in Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII), pp 3–18.
12. Gunter, E. and Muscholl, A. and Peled, D. 2003, Compositional Message Sequence Charts, International Journal on Software Tools for Technology Transfer (STTT), Springer, 5(1), pp 78–89.
13. Holzmann, G. J. and Peled, D. A. Message Sequence Chart Analyzer, United States Patent, 5,812,145.
14. Holzmann, G. J. and Peled, D. A. and Redberg, M. H. 1996, An Analyzer for Message Sequence Charts, Software Concepts and Tools, 17(2), pp 70–77.
15. Harel, D. and Damm W. 2001, LSCs: Breathing Life into Message Sequence Charts, Formal Methods in System Design, 19, pp 45–80.
16. Harel, D. and Kugler, H. 2002, Synthesizing state-based object systems from LSC specifications, International Journal of Foundations of Computer Science, 13(1), pp 5–51.
17. Leue, S. and Mehrmann, L. and Rezai, M. 1998, Synthesizing Software Architecture Descriptions from Message Sequence Chart Specifications, Proceedings 13th IEEE International Conference on Automated Software Engineering, IEEE, pp 192–195.
18. Lohrey, M. 2002, Safe Realizability of High-level Message Charts. In Proceedings of the 13th International Conference on Concurrency Theory (CONCUR).
19. Mauw, S. and Reniers, M. A. 1995, An Algebraic Semantics of Basic Message Sequence Charts, The Computer Journal, 7(5), pp 473–509.
20. Mauw, S. and Reniers, M. A. 1999, Operational Semantics for MSC'96 Computer Networks, 31(17), 1785–1799.
21. Mitchell, B. 2005, Resolving Race Conditions in Asynchronous Partial Order Scenarios, IEEE Transactions on Software Engineering, 31(9), pp 767- 784, preliminary version also appeared as [22].
22. Mitchell, B. 2004, Inherent Causal Orderings of Partial Order Scenarios, International Colloquium on Theoretical Aspects of Computing, Guiyang China, LNCS 3407, Springer, pp 114–129.
23. Mitchell, B. and Thomson, R. and Jervis, C. 2003, Phase Automaton for Requirements Scenarios, Proceedings of Feature Interactions in Telecommunications and Software Systems VII, IOS Press, pp 77-84.
24. Object Management Group. Unified Modelling Language Specification, Version 2.0 Specification, OMG, 2004. http://cgi.omg.org/.
25. Peled D. 2002, Specification and Verification using Message Sequence Charts. Electronic Notes in Theoretical Computer Science, 65(7).
26. Rudolph, E. and Schieferdecker, I. and Grabowski J. 2000, Development of a MSC/UML Test Format, Formale Beschreibungstechniken fur verteilte Systeme, Verlag Shaker, pp 153-164.
27. Sun, J. and Song Dong, J. 2005, Synthesis of Distributed Processes from Scenario-Based Specifications, proceedings of FM 2005: Formal Methods,

International Symposium of Formal Methods Europe, LNCS 3582, Springer, pp 415-431.

28. Schumann, J. and Whittle, J. 2000, Generating Statechart Designs From Scenarios, Proceedings of the 22nd international conference on Software engineering, pp 314–323.
29. Wang, H. H. and Qin, S. and Sun, J. and Song Dong, J. 2007, Realizing Live Sequence Charts in SystemVerilog, Proceedings of First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE, IEEE, pp 379-388.
30. Whittle, J. and Saboo, J. and Kwan, R. 2005, From Scenarios to Code: an Air Traffic Control Case Study, Journal of Software and Systems Modeling, Springer, 4(1), pp 71–93.
31. Tsiolakis, A. 2001, Integrating Model Information in UML Sequence Diagrams, Electronic Notes in Theoretical Computer Science, 50(3), pp 266–274.
32. Uchitel, S. and Kramer, J. and Magee, J. 2004, Incremental Elaboration of Scenario-based Specifications and Behaviour Models using Implied Scenarios, ACM Transactions on Software Engineering and Methodology (TOSEM), 13(1), pp 37—85.
33. Z.120 (11/99)ITU-T Recommendation - Message Sequence Chart (MSC)