Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams

Paul Baker, Paul Bristow, Clive Jervis, David King, Robert Thomson,
Motorola Labs, Jays Close, Basingstoke, RG22 4PD, UK
fPaul.Baker, Paul.C.Bristow, Clive.Jervis, David.King, Rob.Thomsomg@motorola.com

Bill Mitchell,
Department of Computing, University of Surrey, Guildford, GU2 7XH, UK
w.mitchell@surrey.ac.uk

Simon Burton,
DaimlerChrysler AG, Vehicle IT Research System Architecture and Middleware, Research and Technology, HPC G021, 71059 Sindelfingen, Germany
simon.burton@daimlerchrysler.com

ABSTRACT

Scenario based requirements specifications are the industry norm for defining communicating systems. These scenarios are often captured in the form of UML/MSC sequence diagrams. Errors are often introduced at this stage of the development process, which are costly to resolve if they are not detected early. This paper is concerned with the automatic detection and resolution of semantic errors that can occur in such scenarios.

The paper discusses a semantic interpretation of scenario-based requirements and various types of defects (or pathologies) that can be detected. The paper defines the semantics and defects within a partial order theoretic framework. We introduce a UML 2.0 profile that captures various domain specific communication semantics, which can be used to determine the relevance of detected pathologies when different underlying implementation assumptions are made. The paper also discusses how to automatically resolve pathologies by using this profile to adapt the communication architecture in the requirements model.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification

General Terms

Verification

Keywords

Requirements, UML 2.0, MSC, communication semantics

1. INTRODUCTION

Requirements specifications have been shown to be a significant source of defects. Published case studies, [9, 12, 14, 18], have shown that over a third of significant behavioural defects can be traced to requirements specifications. It is well known that requirements defects cost exponentially more to fix the later they are detected [6]. This situation is no different for communicating concurrent systems that use sequence diagrams to define their behaviour. Indeed, in telecommunications this issue is exacerbated by the complexity of concurrent systems which leads to a large effort spent on testing. Testing can take as much as 40-75% of the lifecycle resources [4]. Therefore, tools and techniques to improve the quality of requirements, and to automate testing, can have a large impact on productivity.

Motorola Labs have been involved with the introduction of automated verification into the development process for some time. Initially this effort was focused on developing an automatic test generation tool, *ptk* [3], that processes scenario-based requirement specifications to generate conformance tests. These scenario-based specifications are drawn using the standardised notations Message Sequence Charts [10] or UML 2.0 Sequence Diagrams [15], which are used extensively in the telecommunications industry, and more recently in the automotive industry.

However, even though it is common for system architects and designers to use the same standard notations, they typically do not contain the rigor needed for machine processing. We found that architects and designers are reluctant to invest the extra effort needed to develop rigorous models, since the test generation benefit is outside of their project scope. This meant that sometimes tests generated from sequence diagrams were invalid, and caused false positives when executed. Hence, our experiences with *ptk* confirmed the findings of earlier studies that showed requirements specifications are a significant source of defects. It became clear that new tools were needed that could automatically analyse sequence diagrams to detect pathologies. For such a tool to be seen as improving the product life cycle, and hence be readily adopted within the engineering community, it was also important that such a tool provide support in resolving issues and not just reporting them.

The paper discusses the following topics:

- We introduce a new tool called Mint, which detects defects (or pathologies) found within requirements and architecture specifications, defined using Message Sequence Charts (MSC) or UML 2.0 Sequence Diagrams.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.
Copyright 2005 ACM 1-59593-014-0/05/0009 ...$5.00.

- In Section 3 we formally define those pathologies that Mint currently detects. These are blocking conditions, non-local choice, non-local order, and false-underspecification. Of these, to the best of our knowledge non-local order and false-underspecification are new, and the classification of blocking conditions into resolvable and irresolvable is new.

- In Section 4 we introduce a new UML 2.0 profile that allows users to specify domain specific communication semantics. This in turn allows the above pathologies to be reported only when they are a problem on the target platform. This came about after early Mint users reported that some of the pathologies found were not an issue on their platform.

- We discuss how the profile can be used to automatically resolve certain types of pathology. In some cases it is possible to automatically correct defects by imposing constraints derived from the UML 2.0 profile. Or to provide the practitioner with a range of solutions depending on which aspects of the profile they wish to adopt. We give an example from an industrial case study in Section 4.4 that illustrates how the UML 2.0 profile can be used in this way.

- Finally in Section 5 we give details of a case study of Mint on a set of requirements taken from a Motorola product group.

2. MSC/UML 2.0 SEQUENCE DIAGRAM SEMANTICS

For the purposes of this paper we will treat UML 2.0 Sequence Diagrams (SDs) [15] as equivalent to Message Sequence Charts (MSCs) [10] in terms of graphical notation, and semantics. In practice the two languages are syntactically and semantically the same for most constructs, differing only in terminology, for example, MSC instances are SD lifelines, and MSC inline expressions are SD combined fragments.

It is not within the scope of this paper to give a full description of all the constructs in MSC or SDs, but we will cover the common constructs that are used in this paper. Within a SD events on a lifeline occur linearly down the page, unless a special construct such as a co-region is used (see Figure 5), which means that the events inside are unordered. Lifelines are asynchronous with other lifelines, so that an event on one lifeline that is visually lower than an event on another lifeline is not necessarily temporally later.

Messages are asynchronous, latency is assumed to be arbitrary and there are no queuing semantics associated with message channels. This means, for example, message overtaking is possible. A message is regarded as a pair of events, a send event and a receive event. In accordance with the ITU TTCN-2 standard [11] we use the notation $!m$ and $?m$ for the send/receive pair for message m.

Inline constructs can be used to compose behaviours, this includes the constructs `SEQ` for weak sequential composition, `PAR` for parallel composition, `ALT` for choice, and `LOOP` for iteration, besides others. The alternative construct denotes mutually exclusive alternatives, which are delineated by dotted horizontal lines. Figure 3 shows an alternative construct with two operands. SDs can also refer to other SDs by using an inline reference, or by using the reference construct.

The traces of a SD are given by constructing all possible interleavings of the events from the processes in the diagram (after inlining referenced diagrams) that are consistent with the implied temporal ordering defined by the diagram. So, a send must happen before its receive partner, and an event higher on a lifeline must happen before one below, unless a special construct is used. One non-obvious behaviour is that fragments (i.e. a section of a SD) are weakly composed, so for example, an event above a reference could occur before, during, or after the reference. The order is dependent on the implied orders in the diagram, and not the visual order. Unrelated events can be ordered by using a general-order arrow between them. We also treat conditions as a synchronisation barrier between lifelines, this is non-standard, but is a useful means of ordering unrelated events. An example of an MSC with general-order arrow, and a condition is shown in Figure 4. For the complete definition of the MSC language see the ITU MSC-2000 standard [10].

A basic SD represents a set of traces that can be defined solely in terms of a single partial order on the events in the diagram. This partial order is known as the *causal order*. The traces of a basic diagram are precisely the set of total orders on the events in the diagram that are an extension of the causal order. Hence, for any trace T of SD S, an event e_1 can occur earlier in the trace than another event e_2 if and only if $e_1 \not> e2$, where $<$ is the causal order of S. Diagrams containing the alternative construct, for example, are not basic diagrams since each alternative requires a separate partial order to define its trace semantics. Similarly diagrams containing unbounded iteration are not basic. Examples of diagrams that are basic are any that only contain messages, internal actions, states, continuation symbols, process creation and destruction and the parallel construct. Note this categorisation is not complete.

A consequence of alternative and loop constructs in UML 2.0 Sequence Diagrams is that the general property checking problem is undecidable for arbitrary SDs [2].

3. PATHOLOGIES IN REQUIREMENTS

Automated test generation relies on the initial requirements SDs being semantically consistent. This is equally important when requirements scenarios are used for developing architecture models. For these reasons Motorola has developed a tool, Mint, to automatically detect pathologies in MSC and UML 2.0 Sequence Diagrams.

The current tool detects a variety of pathologies that make a SD semantically inconsistent. In a distributed environment each lifeline in a SD should completely describe the expected behaviour for the associated process. It is possible within a SD to specify global behaviour that may not be a result of the concurrent local behaviour from each process. Certain kinds of global behaviour may only be achieved through implicit access to some global state that might not exist in a distributed environment. Below we list the pathologies detected by the current Mint tool:

- *Blocking conditions.* These are a form of race condition. They describe a discrepancy between the message ordering specified in the requirements scenario and the order that events can occur in practice.

- *Non-local choice.* These pathologies occur where in-

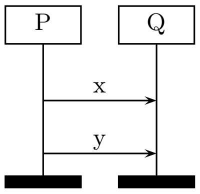

Figure 1: Resolvable blocking condition.

dependent processes must take non-deterministic mutually exclusive actions without sufficient coordination to guarantee exclusivity.

- *Non-local ordering.* These occur when events on separate life-lines are ordered with constructs that can not force the ordering to occur in practice. For example, if a general-ordering arrow is used between events on separate life-lines.
- *False-underspecification.* These occur where the local order for a process is weaker than the implied global order for the whole scenario. Therefore, the behaviour for an individual process can not be inferred from the specification of the process alone.

The first three pathologies are the most serious and must be addressed whenever they are detected. The exception to this is where a scenario is based on a refinement of the standard semantics that renders the pathology benign in practice. For example certain buffering semantics (Section 4) render particular types of blocking conditions benign. The last pathology is less serious, so although it should be avoided, it does not prevent a correct implementation of the requirements.

3.1 Blocking conditions

Blocking conditions are a form of race condition. We prefer the term blocking condition because this is closer to the process behaviour when such pathological specifications are implemented.

Definition 3.1 (Blocking condition). *Let S be a basic MSC/UML SD with causal ordering $<$, as defined by ITU MSC semantics [10]. There is a* blocking condition *with $?y$ if there exists an event $x \neq !y$, which occurs earlier in the causal order than $?y$ (i.e. $x < ?y$), but does not occur earlier in the causal order than $!y$, (i.e. $x \not< !y$).*

Blocking conditions occur because an instance has no control over when it receives messages. Figures 1 and 2 illustrate two examples of blocking conditions. In Figure 1, it is straightforward to implement the behaviour of each instance as a separate process. Process P would send x before y, and process Q would receive x before y. The problem occurs when these processes are running concurrently — message y could overtake message x in transit, and arrive first. In this case process Q would be blocked with the receipt of y, when it was expecting message x.

Examples such as Figure 1 have not been problematic in practice due to properties of the implementations that are used. For example, assuming constant message latency would resolve the blocking condition. Therefore, we call this type of blocking condition a *resolvable* blocking condition.

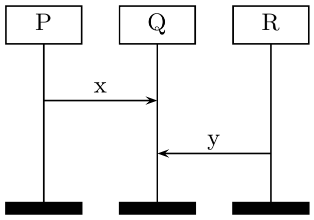

Figure 2: Irresolvable blocking condition.

Figure 2 also contains a blocking condition, but this case is one that has proven to be problematic in practice, and we therefore call it an *irresolvable* blocking condition. With this example message x, and message y can be sent in either order, but there is no possible way to enforce that message x arrives before message y, without adding additional coordination. One way of resolving this case is to use a coregion for the receive messages on instance Q, which would specify that messages x and y can be received in any order.

Definition 3.2 (Resolvable and Irresolvable). *A blocking condition between event x and $?y$ is* resolvable *if there exists an event w which occurs earlier in the causal order than x (i.e. $w < x$), and w also occurs earlier than $!y$ (i.e. $w < !y$), or w is on the same instance as $!y$. Consequently, a blocking condition that is not resolvable is said to be* irresolvable.

Definition 3.2 classifies a blocking condition as resolvable or irresolvable with respect to the standard communication semantics [10]. This is the classification that was applied in the original version of Mint. User feedback made it clear that this semantics is too restrictive. The UML profile we introduce in Section 4 represents a rigorous encapsulation of the different semantics that were encountered in practice. As we introduce the UML profile we will modify the classification to reflect these semantics.

3.2 Non-local pathologies

Non-local pathologies occur when the behaviour of process A depends on some aspect of the run-time behaviour of process B which is not observable by A. Non-local pathologies fall into two categories, non-local choice and non-local ordering. Informally, non-local choice occurs where a choice of paths taken by B is not observable by A, but nevertheless affects the permitted behaviour of A. Non-local ordering occurs where A must wait for some event on B that is not observable to A.

We define these concepts more precisely. In the following, a trace prefix is simply a prefix of some trace in the MSC. An event x in a trace is not observable by instance A if x occurs on some instance other than A. Two traces t_1 and t_2 are observably identical to an instance A if the traces t_1' and t_2', obtained by removing all events not observable by A from t_1 and t_2 respectively, are equal.

Definition 3.3 (Non-local pathology). *Let x be an active event on instance A, i.e. a send or an action, but not a receive or a timeout event. A non-local pathology for A is a triple (x, t_1, t_2) where t_1 and t_2 are valid trace prefixes such that t_1 and t_2 are observably identical to A, but $t_1 \cdot x$ is a valid trace prefix whereas $t_2 \cdot x$ is not.*

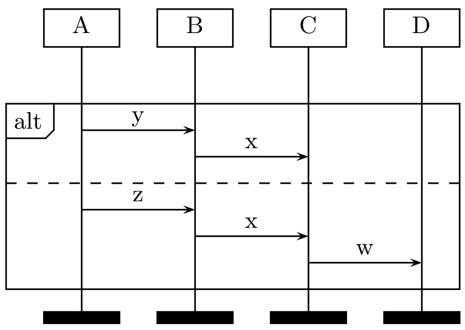

Figure 3: Non-local choice.

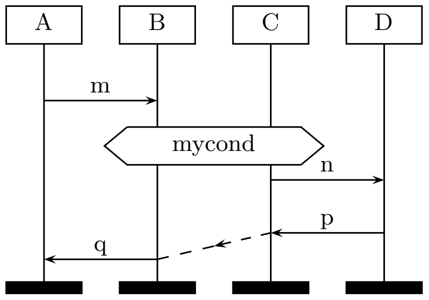

Figure 4: Non-local ordering.

Definition 3.4 (Non-local choice/ordering).

- *Let x be an active event on instance A, let t be a valid trace prefix, and w be some sequence of events unobservable to A such that $t \cdot w \cdot x$ is also a valid trace prefix. Then x is prevented by* non-local ordering *if $t \cdot x$ is not a valid prefix.*
- *Let (x, t_1, t_2) be a non-local pathology for A. Then x is prevented by* non-local choice *if there is no sequence of events w unobservable to A such that $t_2 \cdot w \cdot x$ is a valid trace prefix.*

Suppose that x is prevented by a non-local ordering because of trace t and sequence w of events unobservable by A. Then the triple $(x,\ t \cdot w,\ t)$ is a non-local pathology for A. Hence both non-local ordering and non-local choice are special cases of non-local pathology.

An example of non-local choice is shown in Figure 3. After receiving message x, C makes a decision on whether to send w or not. C should send w if and only if message z and not message y is sent between A and B. However, this choice of messages by instance A is not observable to C.

Non-local orderings only occur when general ordering arrows are used, or where condition symbols are used as synchronising events. In Figure 4, C should not send message n until B has received message m, however C has no way of observing $?m$. Similarly, B cannot send message q until C has received message p, but B cannot observe $?p$.

3.3 False-underspecification

A false-underspecification occurs when two events on a single lifeline are drawn to be unordered, but when considering the complete specification they are actually ordered. A false underspecification is shown in Figure 5.

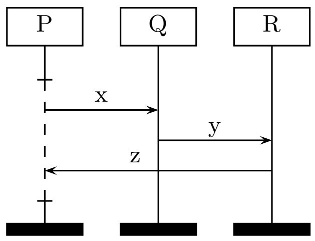

Figure 5: An example of a false-underspecification.

In Figure 5 the co-region on instance P implies that the events $!x$ and $?z$ are unordered, but considering the implied orders of the whole specification $!x$ must occur before $?z$. This is easily correct by removing the co-region.

Definition 3.5 (False-underspecification). *The* false-underspecification *pathology occurs between two events on the same lifeline that can occur in any order when considering the lifeline alone, but when considering all the lifelines they are ordered.*

3.4 Implementation

Pathologies contained in basic SDs can be detected by checking their causal orders. The causal order can be efficiently represented as a partial order graph that describes the Hasse diagram of the causal order. Any non-iterative sequence diagram is semantically equivalent to a finite set of basic SDs. Given a non-basic SD, a set of partial-order graphs representing the whole diagram can be constructed. Each partial-order graph can then be analysed for pathologies. This is the technique implemented by the Mint tool.

4. COMMUNICATION SEMANTICS

Practitioners often construct scenario specifications based on domain knowledge that is not explicitly contained within the model. Frequently, practitioners are not even aware that they are implicitly building these assumptions into the model, for example, assumptions are often made about communication channel semantics etc. With initial versions of the Mint tool, when users did not specify what communication semantics were meant to be applied this caused unnecessary errors to be reported, which users found frustrating since they regarded these errors as 'bogus'.

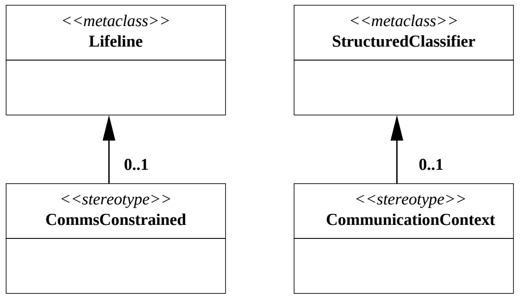

Figure 6: Communication context and communication constrained stereotypes.

In this section, we introduce a means of defining *communication constraints*, thereby, allowing the introduction

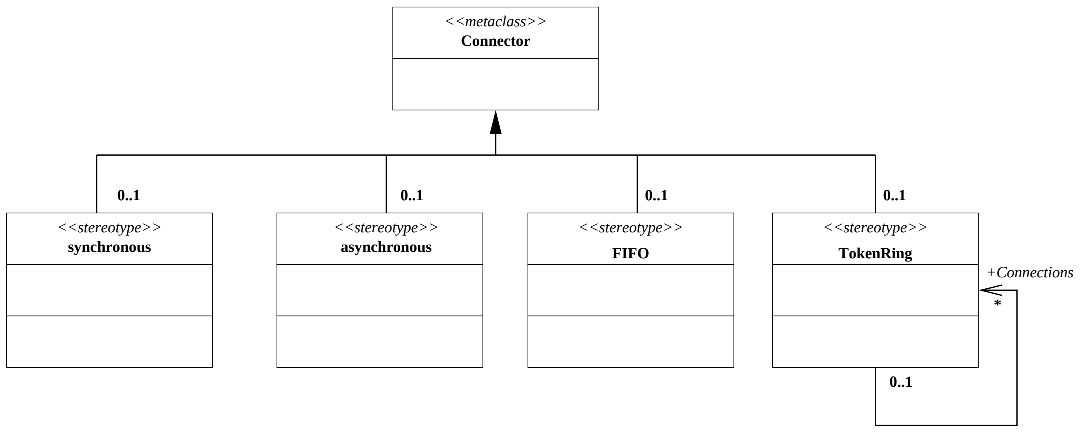

Figure 7: Communication channel stereotype.

of domain specific semantics that can be considered when analysing scenario specifications. This enables an analytical tool, such as Mint, to deduce that certain pathologies may be caused by a gap in the semantic model. It is then possible to automatically construct enhancements to the model that will resolve these pathologies in certain circumstances.

We define these communication constraints using a simple UML 2.0 profile [15] — referred to as the Communication constraint UML Profile (CUP). In doing so, constraints can be applied to UML diagrams that allow the user to define domain specific communication semantics, which in some cases will avoid blocking conditions. Each communication constraint refines the causal order of a sequence diagram to produce a new partial order that defines the sequence diagram semantics. Hence we are able to constrain the behavioural semantics for sequence diagrams purely within a partial order theoretic framework. This has the beneficial side effect of allowing the earlier results on automated test generation from SDs [3] to generalise to SDs with CUP semantics.

Initially, we allow users to define communication constraints by adding specific semantics to UML 2.0 Architecture Diagrams that are considered when analysing SDs. We introduce `<<CommunicationContext>>`, a stereotype that can be applied to classifiers, which encapsulates both architectural constraints and behavioural operations that can be specified using SDs — see Figure 6. In doing so, it provides a binding between constraints and a partial model for analysis purposes. Note that the application of communication constraints with other forms of UML behaviour definition, such as state machines, is outside the scope of this paper. Next we introduce `<<CommsConstrained>>`, a stereotype for SDs that defines those objects, and their associated parts, to which some communication constraints have been applied. If no communication constraints have been applied to a specific object then this stereotype will not apply. Next we introduce two types of communication constraints:

1. *communication channels* – specialised channel semantics for connectors that impact the communication of events between objects, and
2. *communication buffers* – specialised buffer semantics for ports.

4.1 Communication channels

Communication channels are typically represented using connectors or interfaces within an UML 2.0 Architecture Diagram. Hence, we allow the definition of specialised communication channels through the definition of a channel stereotype — see Figure 7. Initially, we have defined four types of channel semantics within CUP as follows:

1. synchronous channels, with stereotype `<<synchronous>>`
2. asynchronous channels, with stereotype `<<asynchronous>>`
3. channels that act with FIFO semantics, with the stereotype `<<FIFO>>`
4. channels that act with token-ring semantics, with the stereotype `<<TokenRing>>`

4.1.1 Asynchronous channels

The `<<asynchronous>>` stereotype represents asynchronous communication channels. That is, the stereotype imposes no order on the delivery of messages that are transmitted along such a channel. Hence, where messages are transmitted along asynchronous channels the semantics of a SD is given by the usual partial order semantics.

4.1.2 Synchronous channels

The `<<synchronous>>` stereotype represents synchronous communication channels. A channel can belong to this particular stereotype only when the channel latency is negligible with respect to the system. Suppose that a message m is sent between processes A and B along a synchronous channel; then no other event connected to A or B is capable of occurring between $!m$ and $?m$.

The causal order for a SD is affected by synchronous channels. Let S be a SD and $<$ be the standard causal order defined by the partial order semantics for S. This semantics assumes all messages are asynchronous.

Define a new partial order $<_{\mathsf{SC}}$ to be the transitive closure of the following binary relation $<_{sc}$.

- $x <_{sc} y$ if $x < y$

- if m is transmitted over a synchronous channel and if $x <?m$ then $x <_{sc} !m$, and if $x >!m$ then $x >_{sc} ?m$, where $x \neq !m$

The partial order $<_{\mathsf{SC}}$ defines the causal order for a SD containing synchronous and asynchronous channels.

4.1.3 FIFO channels

The `<<FIFO>>` stereotype represents channels that preserve the order in which messages are transmitted.

Let S be a SD and let $<$ be the standard causal order defined by the partial order semantics for S. Define a partial order $<_{\mathsf{FIFO}}$ to be the transitive closure of the following binary relation $<_{\mathsf{fifo}}$.

- $x <_{\mathsf{fifo}} y$ if $x < y$
- $?m <_{\mathsf{fifo}} ?n$ where m and n are messages transmitted along the same FIFO channel, and $!m <!n$

The partial order $<_{\mathsf{FIFO}}$ defines the causal order for a SD containing FIFO channels and asynchronous channels.

The FIFO and synchronous causal orders are different. This can be seen from the example shown in Figure 2. In this example we will allow x to be sent along an asynchronous channel, and vary the semantics for y. Consider when first we constrain y to be sent along a synchronous channel. In this case we have $?x <_{\mathsf{SC}} !y$. Next consider when we constrain y to be sent along a FIFO channel. In this case $\neg(?x <_{\mathsf{FIFO}} !y)$. When a SD contains a mixture of asynchronous, synchronous and FIFO channels, the causal order is defined by taking the transitive closure of the union of $<_{\mathsf{SC}}$ and $<_{\mathsf{FIFO}}$. That is the causal order is defined as the transitive closure of the binary relation $<_1$ defined by:

- $x <_1 y$ if $x < y$
- $x <_1 !m$ if there is some message m transmitted along a synchronous channel where $x <?m$
- $?m <_1 ?n$ where m and n are messages transmitted along the same FIFO channel, and $!m <!n$

With the above definition of $<_{\mathsf{FIFO}}$ we can no longer use Definition 3.1 to define blocking conditions for FIFO channels. For example, FIFO semantics would resolve the blocking condition in Figure 1. By this we mean FIFO communication will ensure the SD events always occur in practice in the order given by $<_{\mathsf{FIFO}}$, so that no process will be blocked at any time. However, according to Definition 3.1 $?x$ and $?y$ still cause a blocking condition with respect to $<_{\mathsf{FIFO}}$. Hence, Definition 3.1 no longer captures the concept of one process being blocked by another during execution in the case of FIFO channels. Definition 4.1 characterizes blocking conditions for FIFO semantics.

Definition 4.1. *Receive events $?m$ and $?n$ connected to the same process are FIFO-blocked if $?m <_{\mathsf{FIFO}} ?n$, and it is not the case that $!m <_{\mathsf{FIFO}} !n$. For any non-message event x, then x and $?n$ are FIFO-blocked if $x <_{\mathsf{FIFO}} ?n$, but it is not the case that $x <_{\mathsf{FIFO}} !n$.*

It is possible to use an alternative partial order to characterise those parts of the behaviour that can cause FIFO-blocking. We will denote this new partial order by $<_{\mathsf{F}}$. The events of an SD with FIFO semantics will always correctly occur in practice if and only if the partial order $<_{\mathsf{F}}$ has no blocking events in the sense of Definition 3.1. Thus we can preserve the theoretical structure for blocking events by using $<_{\mathsf{F}}$ rather than $<_{\mathsf{FIFO}}$.

Define $<_{\mathsf{F}}$ to be the transitive closure of the binary relation $<_{\mathsf{f}}$ defined as:

- When messages m and n are sent between processes P and Q along a FIFO channel, $?m <?n$ and $!m <!n$, then we define $?m <_{\mathsf{f}} !n$.
- Whenever $x < y$ we define $x <_{\mathsf{f}} y$.

Proposition 4.2. *Let S be a SD with FIFO behavioural semantics as defined by $<_{\mathsf{FIFO}}$. S contains no FIFO-blocking conditions if and only if $<_{\mathsf{F}}$ has no blocking conditions in the sense of Definition 3.1.*

Notice that the $<_{\mathsf{F}}$ ordering does not change the local ordering of events within a process. Since local orderings are preserved, the only difference $<_{\mathsf{F}}$ makes is to assert that FIFO channels only transmit a message after any current messages in the channel have been received. This additional assumption does not alter any pathological behaviour in the scenario that is due to the presence of blocking conditions. Figure 8 illustrates the $<_{\mathsf{F}}$ ordering for Figure 10. In this example the FIFO ordering resolves the blocking condition between $?a$ and $?c$, the blocking condition between $?b$ and $?d$, but not the blocking condition between $!b$ and $?c$. To resolve the later blocking condition will require lazy buffers, see Section 4.3. In Figure 8 orderings that are additional to those imposed by $<_{\mathsf{FIFO}}$ are shown by the dotted arrows.

The main advantage of using $<_{\mathsf{F}}$ is that it provides a straightforward way of detecting blocking conditions when there is a mixture of asynchronous, synchronous and FIFO channels. All blocking conditions can be detected by testing the union of $<_{\mathsf{F}}$ and $<_{\mathsf{SC}}$ with respect to Definition 3.1.

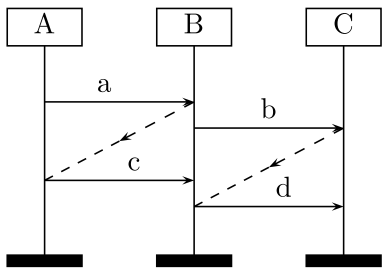

Figure 8: FIFO blocking resolution.

4.1.4 Token-ring channels

The `<<TokenRing>>` stereotype represents a token ring connection. Token-rings are a common network architecture used, for example, by the MOST bus in the automotive industry. A token ring channel can be associated with other token ring channels to denote a token ring network. Having the ability to define a token ring channel means that after a message has been sent, no other message is sent until the first message is received. By enforcing this semantics the set of possible blocking errors is reduced. This is because certain event pairs that would be blocking under the standard semantics are benign within the token-ring semantics.

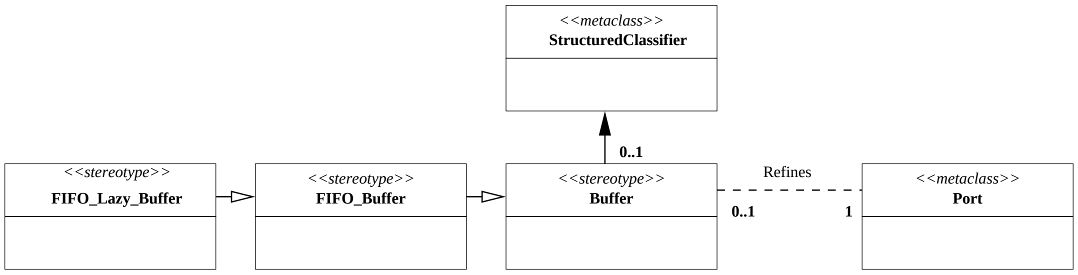

Figure 9: Communication buffer stereotype.

A token ring channel is defined based upon the intuition that a message event cannot happen if the token is taken — it must wait for the receiving process to free the token. Hence, it could be defined as follows. Let S be a SD and let $<$ be the standard causal order given by the partial order semantics for S. Define a partial order $<_{\mathsf{TK}}$ to be the transitive closure of the following binary relation $<_{\mathsf{tk}}$. For any active event x, y and message m,

- $?m <_{\mathsf{tk}} x$ if $!m < x$
- $x <_{\mathsf{tk}} y$ if $x < y$

The partial relation $<_{\mathsf{TK}}$ defines the causal order for a SD with token ring communication semantics. Moreover blocking conditions with respect to $<_{\mathsf{TK}}$ are characterised by Definition 3.1. Thus all blocking conditions for each of the semantics given by the CUP profile can be characterised with the same partial order theoretic property. Hence it is possible to check for blocking conditions in a scenario that uses a mixture of the CUP communication semantics by testing this single property.

4.2 Constraints on message semantics

Messages in UML diagrams can be defined as synchronous or asynchronous. When a message is defined to be synchronous this implies that the message must be transmitted along a synchronous channel. When a message is defined as asynchronous that does not constrain the message to be sent along any particular type of channel. It simply implies no constraint on the channel transmission semantics. Therefore it is not inconsistent to transmit an asynchronous message along a synchronous or FIFO channel. In this paper, where a message is synchronous this will be interpreted as imposing suitable stereotypes on the interacting processes. In particular, it will force the creation of a synchronous channel between the processes if one is not already present.

4.3 Communication buffers

In this section communication channel semantics are extended to include behaviour associated with buffers. In particular, we refine the behaviour of ports that define interaction points for objects within UML 2.0. Ports are typically used as connection points for channels. In doing so, communication over connectors (or communication channels) can be further refined. As illustrated in Figure 9, we introduced the concept of a buffer which must be associated with a port. Because a buffer itself can have a defined behaviour and/or architecture we allow for cases where multiple connections and buffers are needed. However, initially we define two types of specialised buffer semantics:

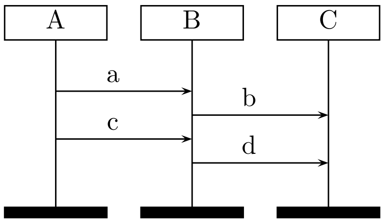

Figure 10: Lazy buffer example.

1. FIFO buffers, with stereotype `<<FIFO_Buffer>>`
2. Lazy FIFO buffers with the stereotype `<<FIFO_Lazy_Buffer>>`

FIFO buffers are standard, and we do not discuss them further here for reasons of space.

Processes may keep received events in buffers and only consume them when necessary with respect to the FIFO causal ordering. This assumes the processes are communicating via FIFO channels. When buffers are accessed in this way we refer to them as *FIFO lazy buffers* — lazy buffers for short. This means that when a message is sent to a communicating process with such a buffer it is automatically placed in the lazy buffer, and will only be consumed when that is correct with respect to the causal order. For example, if we apply lazy buffering semantics to the specification illustrated in Figure 10, the blocking condition between $!b$ and $?c$ would be resolved. In this case when B finds $?a$ in it's input buffer it will be removed and message b is sent. Note that in this case $?c$ could also be in the buffer, and if so, will not be removed from the buffer until b has been sent. The lazy buffer semantics asserts that a receive event $?x$ is not removed from the input buffer for process P if there are any events connected to P that are ordered prior to x by the FIFO causal ordering and that have not yet occurred. Note, the blocking condition between $!b$ and $?c$ in Figure 10 would be reported as irresolvable when applying the standard semantics.

Where lazy buffers are used the definition of blocking condition must change as follows:

DEFINITION 4.3. *Receive events $?m$ and $?n$ connected to the same process are lazy-blocked if $?m <_{\mathsf{FIFO}} ?n$, and it is not the case that $!m <_{\mathsf{FIFO}} !n$. For any non-message event x belonging to a process without lazy buffering, then x and $?n$ are lazy-blocked if $x <_{\mathsf{FIFO}} ?n$, but it is not the case that $x <_{\mathsf{FIFO}} !n$.*

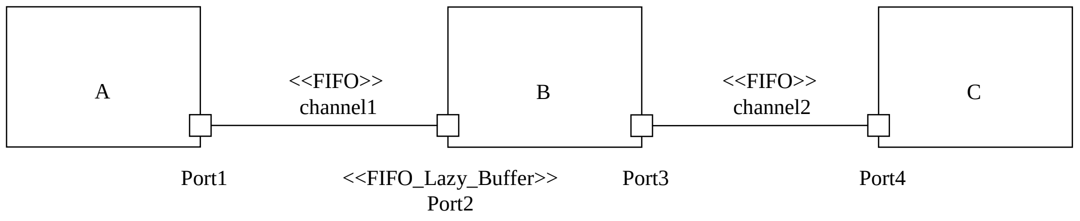

Figure 11: UML Architecture Diagram to modify buffer semantics.

This property does not apply when x is a timing constraint, which is not regarded as a causal event in this discussion. Timing events are regarded as modality events. That is, they characterise some property of the execution trace of the system that can only be determined at runtime. Note that this definition no longer considers the case, where $!x < ?m$ for messages x and m.

As in the case of FIFO ordering, we can use an alternative partial order semantics when we are only interested in detecting blocking conditions, which allows us to use the same definition for blocking across all CUP constraints. This makes the process of detecting blocking conditions straightforward whichever mix of communications constraints are present. Detection only requires the construction of a single partial order to represent all relevant constraints and testing whether that contains any blocking conditions in the sense of Definition 3.1.

For a sequence diagram S with causal order $<$ define the lazy buffer ordering $<_{\mathsf{LB}}$ as the transitive closure of the binary relation $<_{\mathsf{lb}}$ defined as:

- When $!x$ and y belong to the same process define $!x >_{\mathsf{lb}} y$ iff $!x >_{\mathsf{FIFO}} y$.
- Define $?x >_{\mathsf{lb}} y$ iff y is a receive event and y belongs to the same process and $?x >_{\mathsf{FIFO}} y$.
- When x and y are events belonging to different processes, $x <_{\mathsf{FIFO}} y$ and there is no event z where $x <_{\mathsf{FIFO}} z <_{\mathsf{FIFO}} y$, then define $x <_{\mathsf{lb}} y$.

PROPOSITION 4.4. *Let S be a SD with lazy buffering. Then S contains lazy-blocking conditions if and only if $<_{\mathsf{LB}}$ contains blocking conditions in the sense of Definition 3.1.*

4.4 Automatic Pathology Resolution

Where Mint detects a blocking condition, it is sometimes possible to automatically suggest ways of attaching FIFO channels, token-ring channels, buffers and lazy-buffers to processes in order to avoid the pathology. For the example in Figure 10, the channel semantics can be modified as shown in Figure 11 in order to resolve the pathology. By using the CUP profile these solutions can also be presented to the user in a familiar notation, which will be more appealing where some UML toolset is being applied.

It is not always the case that there is a unique solution with CUP semantics that can resolve multiple detected blocking conditions. Nor is it always possible to resolve all blocking conditions with CUP semantics. Where a pathology is detected, a range of possible modifications to the channel semantics via the CUP profile can be constructed that resolves the pathology. The practitioner can now make an informed choice about which of these semantics are appropriate as a solution.

Figure 13 describes an anonymized example from a case study where several blocking conditions are contained in a single scenario. The case study considered sequence diagrams for a HSDPA protocol stack. This example illustrates multiple interrelated blocking conditions. Altogether there are seven blocking conditions in the scenario. Receive event $?m_f$ is in a blocking condition with each of $?m_a$, $!m_b$, $!m_c$ and $!m_d$. The remaining three blocking conditions occur between each of $?m_b$, $?m_i$ and $?m_j$. We can see that the blocking condition between $?m_f$ and each of $!m_b$, $!m_c$ and $!m_d$ can be resolved by introducing a lazy-buffer to process B. The blocking condition between $?m_a$ and $?m_f$ is not resolved by this lazy-buffer, and is irresolvable by any of the CUP semantics.

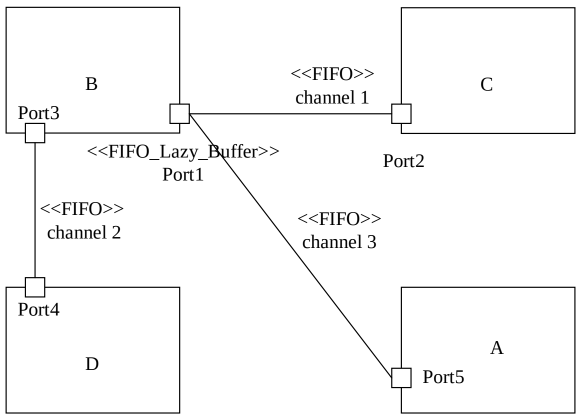

Figure 12: UML Architecture Diagram for Figure 13.

The race between $?m_b$ and $?m_i$ can be resolved either by introducing a FIFO channel or token-ring communication between B and D. The final blocking condition caused by $?m_j$ is not resolvable through any of the CUP communication semantics and will require some other mechanism to resolve it. For example it may be that the scenario author may wish to enclose $?m_b$, $?m_i$ and $?m_j$ in a coregion. The partial resolution outlined above can be automatically constructed by Mint and presented to the user as an architecture diagram Figure 12, together with a report on the remaining blocking conditions that remain. Note Figure 12 only describes those parts of the architecture that need modification.

5. MINT EVALUATION

Mint currently implements the above pathology checks for MSCs and UML 2.0 Sequence Diagrams. Mint can be used

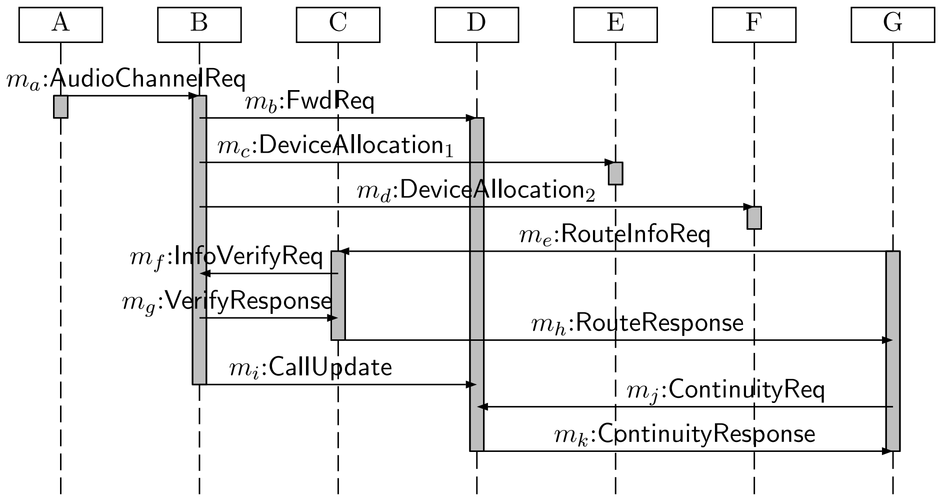

Figure 13: Industrial case study example.

as a standalone tool, but it does not include graphical editors. For graphical editing Mint has been used with the Telelogic Tau MSC editor, and TauG2 for editing UML 2.0 Sequence Diagrams [17], and the ESG editor for MSCs. The input for Mint is the standard textual representation of an MSC. Almost all of the MSC-2000 language, and UML 2.0 Sequence Diagrams language is supported. That is messages, action boxes, inline expressions, references, and so on. Features not supported include lifeline decomposition, gates, exceptions, and other obscure features which are not commonly used by engineers or supported in some of the editors.

When running Mint a report is given of each pathology detected, and its line and column number in the textual file. With the TauG2 graphical editor navigation is provided from the error message to the event causing the error in the diagram by clicking on the error message.

Mint is relatively new and still undergoing evaluation with a number of Motorola engineering groups. Recently it has been applied in a case study of UML 2.0 Sequence Diagram specifications for part of a communications protocol stack. Mint was applied to approximately a hundred and fifty SDs and detected numerous pathologies. After filtering pathologies that can be automatically resolved by Mint using the CUP profile those remaining fell into the following categories:

- 6 diagrams containing multiple non-local choice conditions
- 2 diagrams containing multiple non-local choice conditions between break and loop constructs
- 5 diagrams containing multiple resolvable blocking conditions caused by loop constructs
- 1 diagram containing an irresolvable blocking condition
- 1 diagram containing multiple resolvable blocking conditions between different parallel constructs

The Mint tool is also undergoing trial within DaimlerChrysler's Research and Technology group. They applied the tool to eighty-one MSCs and found thirteen pathologies which were confirmed by manual inspection. Quite a number of other pathologies were also found, however, these were not considered to be relevant due to the differences in communication semantics between DaimlerChrysler's and Motorola's target systems. The work on communication semantics currently being undertaken, and described in Section 4, will enable Mint to tailor its analysis to a given scenario and remove such irrelevant pathology reports.

In summary Mint appears to be successful at detecting semantic inconsistencies in industrial size case studies, and has been found valuable by groups who have used it.

6. RELATED WORK

The OMG are currently proposing a new UML profile for schedulability and performance aspects of real time systems (SPT), [16]. Schedulability and quality of service are complementary to the issues considered here. The SPT profile assumes that the communication semantics are already correctly described within the model. The SPT profile is concerned with building real time systems that are capable of systematic analysis and testing, at both the abstract level before implementation, and at the concrete level after implementation. The CUP profile is concerned with automatically resolving behavioural inconsistencies in a model specification by modifying the communication semantics, and hence applies to an earlier stage of requirements capture than the SPT profile. The CUP profile is also given a partial order theoretic semantics, which facilitates automated checking of the types of pathologies described in the paper.

In [7] a hierarchy of communications models is defined for basic MSCs, and notions of implementability of MSC are given. This differs from our approach in that we have used a UML Profile to define our communication semantics, and our approach applies to the complete MSC language, not just basic MSCs.

There are many software verification tools for communicating systems, the two most closely related to Mint are:

- MESA [5] - was originally developed by Stefan Leue et al, at the University of Waterloo, and is now maintained at the University of Freiburg. MESA provides an MSC editor, and can detect non-local choice, timing consistency, and can also generate Promela for Spin process modelling. MESA is mainly a research vehicle, and is currently only available for non-commercial use.

- UBET [1] - was developed in Bell Research Labs, and Lucent Technologies. UBET provides an MSC editor, and can graphically highlight race conditions and timing violations in an MSC. The user is able to select different queuing semantics for these checks. UBET also can be used to generate test scripts in MSC and process models in Promela, the language of the Spin verification tool.

7. CONCLUSION

In deploying test generation technologies, such as *ptk* [3], we found that many requirements models contained semantic inconsistencies that resulted in flawed tests. When new tools were then produced to identify and report these requirements defects, it was found that engineering groups sometimes used non-standard semantics implicitly that rendered some of the pathologies benign. This lead to the CUP profile, which rigorously captures the most common communication semantics that were used in practice. This profile can be used to reverse engineer an appropriate semantics from detected pathologies in the requirements specifications. Apart from the behavioural inconsistencies captured by the CUP profile, Mint also detects a variety of other pathologies. These include non-local choice and non-local order. The paper has formally defined these various pathologies within a partial order theoretic framework, which makes it possible to automatically detect these pathologies in UML sequence diagrams.

Initial evaluations of Mint have been promising and have lead to the identification of defects early in the requirements process. We are currently continuing to evaluate Mint and are intending to both expand the tool's repertoire of pathology types and improve the reporting of complex errors to users.

8. REFERENCES

[1] Alur, R., Holzmann, G.J., and Peled, D., *An analyzer for Message Sequence Charts*, Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), Springer-Verlag, 1996.

[2] Alur, R., and Yannakakis, M., *Model checking of Message Sequence Charts*, Proceedings of the Tenth International Conference on Concurrency Theory, Springer-Verlag, 1999.

[3] Baker, P., Bristow, P., Jervis, C., King, D., and Mitchell, B., *Automatic Generation of Conformance Tests From Message Sequence Charts*, Telecommunications and Beyond: The Broader Applicability of MSC and SDL, pp 170-198, LNCS 2599, 2003.

[4] Beizer, B., *Software Testing Techniques*, New York, New York: Van Nostrand Reinhold, 1983.

[5] Ben-Abdallah, H., and Leue S., *MESA: Support for Scenario-Based Design of Concurrent Systems*, Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), LNCS 1384, Springer-Verlag, 1998.

[6] Boehm, B., and Basili, V.R., *Software Defect Reduction Top 10 List*, IEEE Computer Society Press, Vol. 34, No. 1, 2001.

[7] Engels, A., Mauw, S., and Reniers, M. A., *A Hierarchy of Communication Models for Message Sequence Charts*, FORTE, 1997.

[8] ETSI ES 201 873-1, *Methods for Testing and Specification; The Testing and Control Notation version 3 (TTCN-3); Part 1: TTCN-3 Core Language*, European Telecommunications Standards Institute (ETSI), 2001.

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

[10] International Telecommunications Union: ITU-T Recommendation Z.120, *Message Sequence Chart (MSC)*, 2000. Available from `http://www.itu.int`.

[11] International Telecommunications Union: ITU-T Recommendation X.292, *The Tree and Tabular Combined Notation (TTCN-2)*, 1997. Available from `http://www.itu.int`.

[12] Lutz, R., *Targeting Safety-Related Errors During Software Requirements Analysis*, Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering, 1993.

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

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

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

[16] Object Management Group (OMG), *UML Profile for Schedulability, Performance, and Time V1.1*, 2005-01-02. Available from `http://www.omg.org`.

[17] Telelogic, *Tau documentation*, Telelogic Web Site: `http://www.telelogic.com`, 2005.

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

MOTOROLA and the Stylized M Logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.