Towards Rigorous Requirements by Automating Appraisals

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

Simon Burton,
DaimlerChrysler AG, Research and Technology, Information and Communication, 059/X910 - RIC/SM, 71059 Sindelfingen/Germany
simon.burton@daimlerchrysler.com

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

ABSTRACT

Deploying advanced automated testing techniques, such as model-based testing, relies upon the development of rigorous models. Our extensive experience in trying to develop and deploy model-based testing within a large industrial setting has led us to the conclusion that developing requirement models is essential for good model-based testing practice. However, not only are requirements specifications generally incomplete, but it is also difficult to get system architects and designers to produce requirements with the rigor needed for automation. Hence, incentives are needed that tend towards the development of rigorous requirement models. To this end, we introduce the Mint tool that enables and helps automate the early detection of errors during requirements development and appraisal. The paper describes and discusses at length the semantic interpretation of scenario-based requirements and the various types of pathologies that can be detected. We also introduce a UML 2.0 profile for applying domain specific communication semantics that can be used to determine the relevance of these pathologies.

1. INTRODUCTION

Like many other large industrial organisations Motorola is looking to reduce the cost and time required for the development of systems and software. In general, experience has shown that during the development of telecommunication software as much as 40-75% of the resources are spent on testing [2, 6]. Also, recent studies indicate that 50% of test failures are caused by defects found in the requirements [15], and these defects can cost as much as one-hundred times more to fix in the development phase than if they were fixed in the requirements or design phase [3]. Therefore, great effort is expended conducting Formal Technical Reviews (FTRs) [14], or appraisals, as a means of discovering such defects earlier on in the software lifecycle. To date, however, appraisals generally rely upon manual analysis, with few, if any, tools for enabling the automated detection of defects. As a consequence, early defect discovery is becoming a key focal point in reducing development costs.

To address these issues Motorola Labs has been focused on introducing automation and reuse into the testing process. Automation is enabled by the development of abstract, yet rigorous models, throughout the development process, and reuse generally through the use of standards and common frameworks. As part of this automation strategy we have developed the automatic test generation tool, *ptk* [3], that processes scenario-based requirement specifications to generate conformance and load tests. These scenario-based specifications are typically used to define part or all of the system requirements. However, even though it is common for system architects and designers to use scenario-based notations, they typically do not contain the rigor needed for machine processing. We also found that architects and designers were reluctant to invest the extra effort needed to develop rigorous models, as the benefit of automated test generation did not immediately justify the extra effort within their project scope. To address this issue we developed Mint, a tool [3] that would automatically discover potential defects within scenario-based specifications, in a user-friendly manner. With this tool we are providing the capability to reduce the effort required in appraising requirement specifications, hence, providing a strong incentive for the system architects and designers to develop the rigorous models needed for automation.

This paper introduces the Mint tool as a means for automating the discovery of defects found within requirements and architecture specifications, defined using Message Sequence Charts (MSCs) [8] or UML 2.0 Sequence Diagrams [16]. Some of these defects have been described before such as non-local choice, and race conditions, but the others to our knowledge are new, such as non-local ordering, false-underspecification, and blocking condition classifications.

We also introduce a UML 2.0 profile that allows users to specify domain specific communication semantics that can be considered by Mint during the analysis of requirements and architecture specifications. This profile came about because in practice some of the defects detected with respect to the UML standard semantics are not defects with respect to

the semantics of the target platform. Hence, the classification of defects has to be adapted for the particular semantics that are expected in practise. The profile gives the user a simple mechanism for rigorously incorporating the appropriate semantics into their requirements model. It is also possible to use this profile to infer resolutions for certain types of pathology. In some cases it is possible to automatically correct semantically inconsistent requirements by imposing constraints derived from the UML 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 profile can be used in this way. Finally, we describe some initial evaluations of Mint, where it has been applied to real scenarios.

2. MSC/UML 2.0 SEQUENCE DIAGRAM SEMANTICS

For the purposes of this paper we will treat UML 2.0 Sequence Diagrams (SDs) [16] as equivalent to Message Sequence Charts (MSCs) [8] 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 [10] 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 4 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.

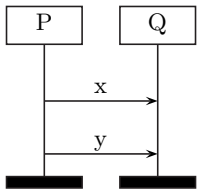

Figure 1: Resolvable Blockinig Condition

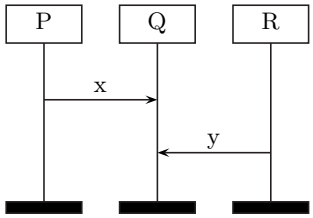

Figure 2: Irresolvable Blockinig Condition

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 [8].

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 [1].

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

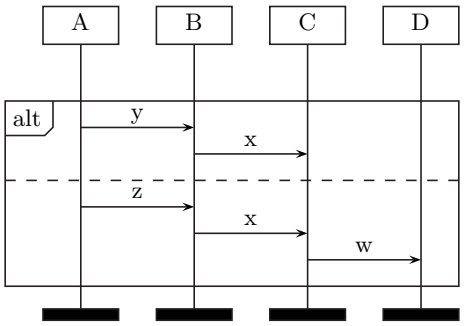

Figure 3: Non-local choice

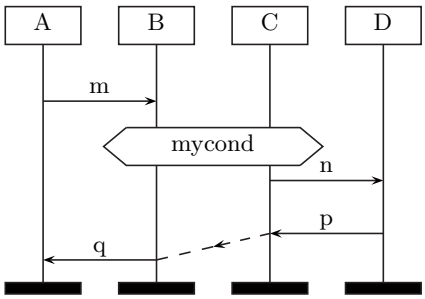

Figure 4: Non-local Ordering

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 independent 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 should be corrected unless there are certain properties of the implementation. Examples of properties that allow the pathologies to be ignored include buffering semantics, which are covered in Section 4. 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 [8]. 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, only when it can send them. For example, 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. The implementation of P would send x before y, and the implementation of 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.

In practice, certain assumptions can be made about the behaviour of processes, for example, a fixed message latency, which would avoid a problem with the implementation of the MSC on the left. Therefore, we call this type of blocking condition a *resolvable* blocking condition. In practice, within Motorola and DaimlerChrysler, such an example as Figure 1 has not been problematic, due to properties of the implementations that are used.

With the MSC Figure 2, however, there is also 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 co-ordination. Another simple way of avoiding this case is to use a co-region for messages on instance Q, which specifies that messages x and y could 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 then 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.

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

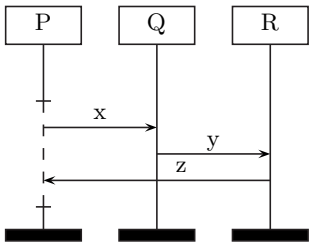

Figure 5: An example of a false-underspecification.

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. A non-local pathology occurs if there are valid trace prefixes t_1 and t_2 such that t_1 and t_2 are observably identical to A, but t_1x is a valid trace prefix whereas t_2x is not.*

DEFINITION 3.4 (NON-LOCAL CHOICE/ORDERING). *Let x be an active event on instance A, and let t_1 and t_2 be trace prefixes that are observably identical to A, such that t_1x is a valid trace prefix, whereas t_2x is not. Then x is prevented by* non-local ordering *if there is some sequence of events w unobservable by A such that t_2wx is a valid trace prefix. Otherwise x is prevented by* non-local choice*.*

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 when Mint uses condition symbols 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.

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). *A* false-underspecification *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

Mint internally constructs a partial-order graph for each basic SD, where the partial-order graph represents the causal order. Given a non-basic SD, a set of partial-order graphs representing the whole diagram is constructed. Each partial-order graph is then analyzed for pathologies and a table of pathologies found is generated.

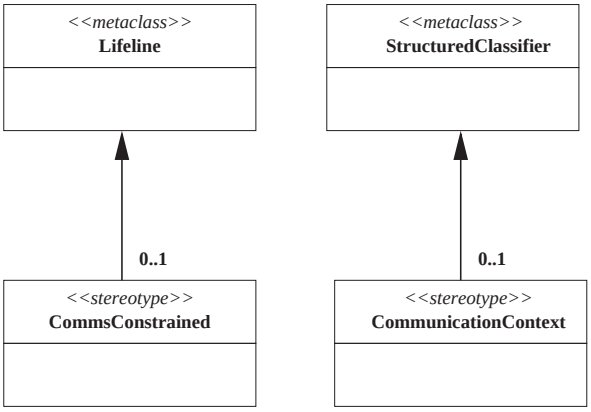

Figure 7: Communication context and communication constrained stereotypes.

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. This causes unnecessary errors to be reported by the Mint tool, which users find frustrating since they regard these errors as 'bogus'.

In this section, we introduce a means of defining *communication constraints*, thereby, allowing the introduction of domain specific semantics that can be considered when analysing scenario specifications. This allows the Mint tool to deduce that certain pathologies may be caused by a gap in the semantic model. Mint can then suggest enhancements to the model that will resolve these pathologies.

We define these communication constraints using a simple UML 2.0 profile [16] — 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 [4] 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 7. 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

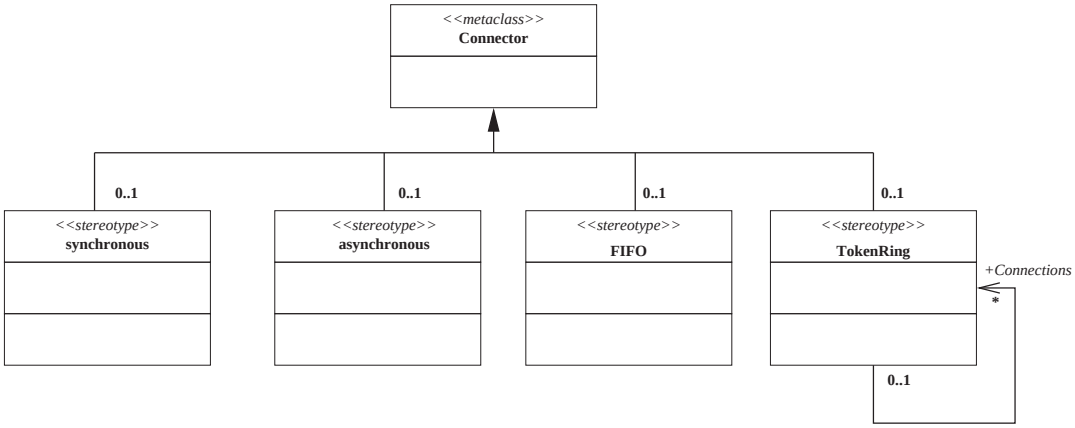

Figure 6: Communication channel stereotype.

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 6. 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 stereotype `<<FIFO>>`
4. channels that act with token-ring semantics, with 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 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 $x < y$

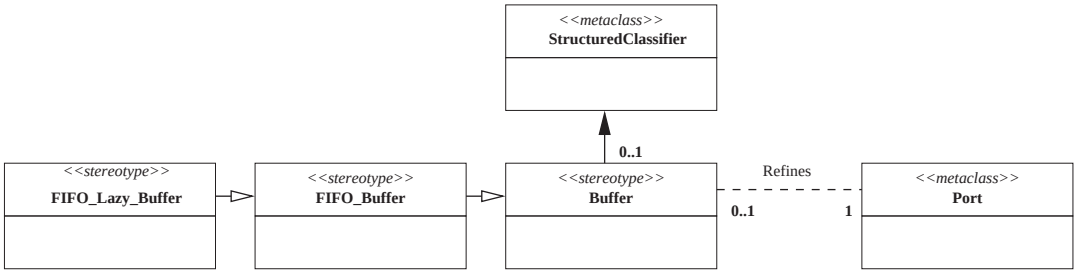

Figure 8: Communication buffer stereotype.

- $x <_1 !m$ 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 force the SD events to always occur in practise in the order given by $<_{\mathsf{FIFO}}$, so that no process will be blocked by any other. 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 practise 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 9 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, Section 4.3. In Figure 9 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.

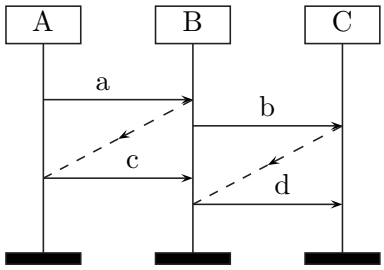

Figure 9: FIFO blocking resolution.

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.

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.

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 the Mint tool, 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 8, 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:

1. FIFO buffers, with stereotype `<<FIFO_Buffer>>`
2. Lazy FIFO buffers with the stereotype `<<Lazy_FIFO_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. Figure 10 illustrates an example of a specification containing a blocking condition reported as irresolvable by the Mint tool.

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 $x <_{\mathsf{FIFO}} !n$.*

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.

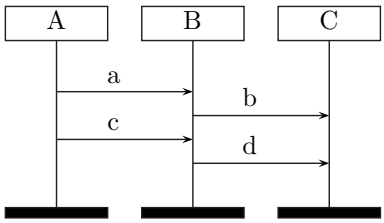

Figure 10: Lazy buffer example.

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. We are only concerned with constructing 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 can 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, Mint can suggest the channel semantics 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.

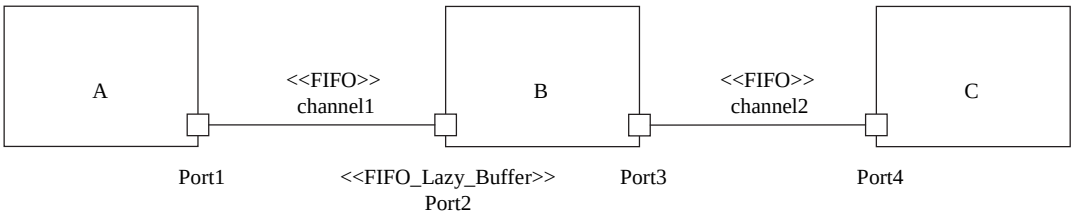

Figure 11: UML Architecture Diagram to modify buffer semantics.

The same principle can be applied if a pathology can be resolved by introducing channel semantics for, say, synchronous channels, or FIFO channels (without lazy buffers). Where Mint detects a pathology it can construct a range of possible modifications to the channel semantics via the CUP profile 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. This example can be viewed as multiple interacting threads that represent different features within one scenario. Altogether there are seven blocking conditions in the scenario. Receive event $?f$ is in a blocking condition with each of $?a$, $!b$, $!c$ and $!d$. The remaining three blocking conditions occur between each of $?b$, $?i$ and $?j$. We can see that the blocking condition between $?f$ and each of $!b$, $!c$ and $!d$ can be resolved by introducing a lazy-buffer to process B. The blocking condition between $?a$ and $?f$ is not resolved by this lazy-buffer, and is irresolvable by any of the CUP semantics.

The race between $?a$ and $?i$ can be resolved either by introducing a FIFO channel or token-ring communication between B and D. The final blocking condition caused by $?j$ 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 $?i$ and $?j$ 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.

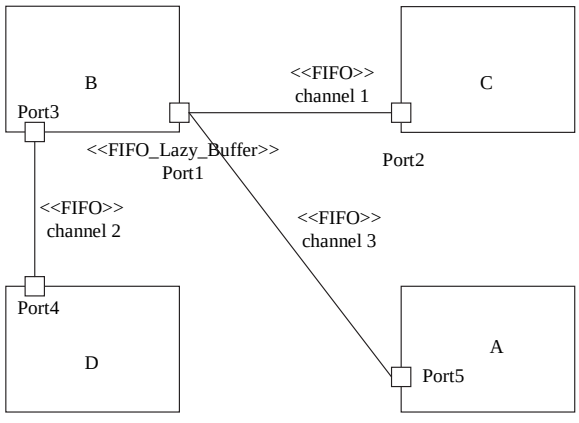

Figure 12: UML Architecture Diagram for Figure 13

5. MINT EVALUATION

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

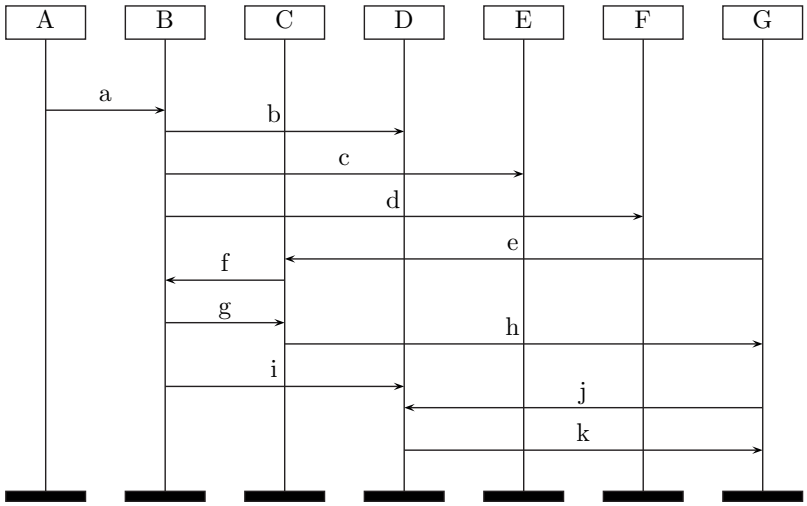

Figure 13: Industrial Case Study Example

Motorola's target systems. The work on communication semantics currently being undertaken, and described in Section 4, will enable Mint to tailor its analyses 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

Here we summarise some software tools related to Mint and *ptk*.

- MESA [11] - was originally developed by Stefan Leue et al, at the University of Waterloo, and is now maintained at the University of Freidburg. MESA provides an MSC editor, and can detect non-local choice, timing consistency, and can also generate Promela for Spin [5] process modeling. MESA is mainly a research vehicle, and is currently only available for non-commercial use.

- UBET [12] - 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

During deployment of requirements based test generation technology we found that requirements and architecture teams were reluctant to develop rigorous specifications, since they saw no direct benefit. Hence, *ptk* [4] was deployed within testing teams who would take the original requirements and rework them into a more rigorous form and then use these for test generation, with good results. In some cases we found that our test-centric approach lead to ptk being used as a graphical scripting tool. As a result, the technology within ptk was not being used to its full potential to generate larger and more accurate test suites than could be achieved by manual development of test scripts. Furthermore, focusing exclusively on testing meant that the benefits of a more integrated process were not being realised.

To counter these problems we altered our strategy to provide technologies, such as requirements validation and the detection of feature interactions [13]. This change was intended to give requirements and architecture teams incentives for constructing more rigorous models, through reduced appraisal costs, which would then enable automated test generation. Pursuing this approach we found that requirements are almost always incomplete, or *partial*, meaning that only a subset of all possible required system behaviours is specified.

Having only partial requirements has implications on the type of analysis that can be conducted on requirements, as detailed in Section 3. Hence, we developed Mint, a tool for the automated discovery of pathologies, or potential defects, during appraisals of partial requirements specifications. Initial evaluations of Mint have been promising and have lead to the identification of defects very 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] R. Alur and M. Yannakakis, *Model checking of Message Sequence Charts*, Proceedings of the Tenth International Conference on Concurrency Theory, Springer Verlag, 1999

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

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

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

[5] 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.

[6] Ghiassi, M., K.I.S. Woldman, *Dual Programming Approach to Software Testing*, Software Quality Journal, 3:45-58, 1994.

[7] Hartman, Alan, *AGEDIS Model-Based Test Generation Tool*, `http://www.agedis.de`.

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

[9] International Telecommunications Union: ITU-T Recommendation Z.100, *Specifications and Description Language (SDL)*, 2000. Available from `http://www.itu.int`.

[10] International Telecommunications Union: ITU-T Recommendation Z.100 X.292, *TTCN-2 standard, Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation (TTCN)*, 1997.

[11] Leue, Stefan, *MESA: MSC Editor Simulator Analyzer*. Available from: `http://tele.informatik.uni-freiburg.de/Mesa/Mesa_doc/index.html`.

[12] Lucent Technologies. *UBET documentation*. Information is available from `http://cm.bell-labs.com/cm/cs/what/ubet/index`.

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

[14] NASA, *Formal inspection standard - NASA-STD-2202-93*, `http://satc.gsfc.nasa.gov/fi/std/fistdtxt.txt`.

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

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

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

1

[1]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.