A Light-weight Method for Trace Analysis to Support Fault Diagnosis in Concurrent Systems

Andrej Pietschker and Andreas Ulrich
Siemens AG
Corporate Technology
Software & Engineering
Munich, Germany
{Andrej.Pietschker, Andreas.Ulrich}@mchp.siemens.de

Abstract

This paper discusses a light-weight approach to the analysis of traces of partially ordered events collected during the execution of a concurrent or distributed system through the use of XML technology. Traces contain information about the creation and termination of threads or objects and the exchange of messages and other types of communication among them. Traces are transformed according to property patterns, visualised and analysed to support fault diagnosis of concurrent systems. We present an approach using XML technology and report the findings of an initial industrial project.

Keywords:

concurrent systems; dependability of concurrent systems; XML

1 Introduction

Software developers often face difficulties when trying to identify causes of software failures. A failure may be observed during test and then a developer has the task of reproducing the failure, to examine the error and hypothesise about the fault. Complementing his knowledge about the software system with the insight information gained through a debugger he tries to hypothesise about the cause of the failure. Debuggers deliver the main information in this approach.

Concurrent or distributed systems offer a number of advantages over centralised applications. However such systems behave highly nondeterministic and this makes testing, debugging and analysing difficult. To support these tasks tools have been developed that allow to monitor systems and produce log files of execution traces. Tracing an object-oriented concurrent system consists of collecting events from object and thread creation and termination, method invocation and execution among concurrent objects, plus relevant local events from variable values in objects. Tracing must be included as an additional feature in the systems architectural design. Traces therefore offer support during debugging, analysis and verification of concurrent systems.

There are approaches like in [6] which use a re-engineering technique to express the systems behaviour in a formal model and apply model-checking to reason about desired or undesired properties. It is not always necessary to go to the expense of creating and analysing a model. In practice some properties of interest can be expressed in terms of local properties of events.

The challenge is to present the information contained in traces to the developer in a form which supports his task of finding the fault which caused the failure.

However implementing tools for analysis to be used on a single project is usually too expensive. We want a more general modular approach which is flexible to be adapted to many situations. We propose a light-weight method for trace analysis based on XML [1].

We present a method for automatic trace analysis which by utilising standard XML technology supports fault diagnosis in concurrent systems. XML technology is used to define properties of interest, process traces and create visual representations of the result of the analysis. The advantage lies in the rapid development of trace analysis tools which can be deployed instantly in projects. The costs of additional tools and development are kept low.

Next we present an overview of the trace analysis process in Section 2 before we discuss the various components of the framework in Section 3. We conclude with the findings from a project in Section 4.

2 Overview of trace analysis process

We propose the use of XML standard technology to analyse traces of systems. In this way we minimise the development efforts for specialised tools.

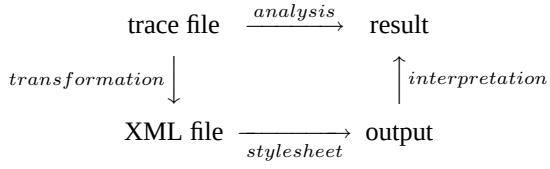


Figure 1. Using XML tools in trace analysis

The diagram presented in Figure 1 illustrates our idea. Instead of focusing on semantic analysis of traces we use a syntactic transformation step to achieve the same result in an automated setting. However it is important to note that the analysis is

limited to those properties which can be described syntactically. Therefore the approach follows the same notion that is used when solutions to equations are sought through syntactic manipulation in mathematics.

The foundation of our approach are traces in XML notation. Together with the transformation technology XSLT [3] we analyse traces based on syntactic rules. From information extracted during transformation and visual inspection of the result we aim to draw conclusions about system properties. In our trace analysis approach, we

- observe an execution trace of the concurrent system that contains a list of partially ordered events for the registration of threads and objects, communication and local events,
- convert trace files to XML-based files off-line,
- create a description of properties of interest using XML style sheets (XSL),
- apply style sheets to XML trace files utilising an XSLT-processor.

The trace analysis process follows the "information seeking mantra" [11] of overview, zoom and filter, and details-on-demand.

- *Visualisation.* Through visualisation trace information is made accessible to the user. We chose to use the SVG format. Scalable vector graphics (SVG) [13] are an XML-based format for graphical representation. There are a variety of tools available for visualising SVG content, see [12] for further references. Using an existing viewer to visualise trace files freed us from the overhead of creating a visualisation tool.
- *Filtering.* Traces of systems tend to be large and therefore become unmanageable. A possibility is to reduce the size and information contained in a trace. These operations have to be accurate and consistent.
- *Details-on-demand.* Not all information should be present in a trace representation at all times. This would clutter the display and make finding relevant information difficult. However when inspecting events closer local information needs to be relayed to the user.

3 Implementation of trace analysis

Typically trace analysis involves checking for race conditions, safety, and liveness properties. These analyses can include proof [10] or model-checking [8]. Whereas the former one requires heavy user support the latter is restricted by the state space generated from the model.

On the other hand tools for XML are ready available on a variety of platforms. Their performance is improving with each version.

- *XML format.* We defined a common trace format in XML. This provides a standard interface to our trace analysis tools.
- *XML technology.* We propose to use standard XML technology to process traces. With already existing XSLT processors we can focus on analysis rather than on its implementation issues.

3.1 Trace format

Events are collected during the execution of a concurrent system. We distinguish the following event types:

- *Registration events.* Devices, processes, threads, and objects register and unregister during system runtime. These events define the lifetime of their source and allow us to match other events to hardware or software processes.
- *Communication events.* Send and reception of messages are communication events. Remote method invocation belongs to this group of events too.
- *Local events.* This group contains events which happen locally to a thread or object. It includes events like variable assignments and assertion checks.

Since a trace is input to our analysis approach, facilities in the system under observation must exist to create trace events at appropriate places in the source code of the system. Different methods to insert such probes exist. The most promising of them are, of course, those which work automatically. At Siemens, we explore different techniques for different platforms, such as Microsoft COM, Java RMI, CORBA and others, to do this job [5].

To be of use for trace analysis, each trace event that is a communication or local event must obey the following structure in order to assist model reengineering in the next step.

- Name and type of the event: Send, Receive, or Local.
- ID of the issuing thread or object.
- ID of the source thread and object (for Receive).
- ID of the destination thread and object (for Send).
- Message parameters of the event: a list of typed parameters that includes a message name and other message attributes (for Send and Receive events).
- Local parameters of the issuing thread or object: a list of typed parameters that reflect its current state (for Local events).

Furthermore, Registration events occur that introduce newly created threads, processes or objects in a trace. We assume that the local order of all events is preserved by their order of appearance in the trace. The problem of matching receive and send events is crucial to determine the partial order of events and eventually base the analysis on the recorded trace. Much depends on how the distributed system is instrumented and what exactly is monitored. We assume that in a pair of source and destination processes, each Receive event matches with a single Send event. Lamport defines a partial order over events in a distributed system using the binary happened-before relation ($\rightarrow$) [9] which is the theoretical basis in our approach. It is defined as follows.

- If event e in thread t preceeds event e' in the same thread t then $e \rightarrow e'$.
- If event e is a send event in thread t and event e' is the corresponding receive event in thread t' then $e \rightarrow e'$.
- The happend-before relation is transitive.

Figure 2 illustrates the XML format of an event. An event is described by its *type* and *operation*, e.g. a send event has *communication* as its type and *send* as its operation. Each event then contains an element *parameters* which holds information about the origin of the event as explained above. Additionally each

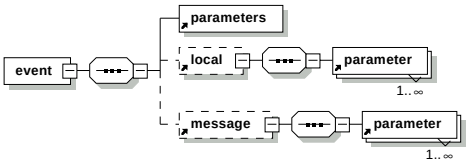


Figure 2. Structure of trace events in XML

event can have an element *local* which contains information about the state of local variables. Communication events contain an element *message* with the particular content of the message.

The format is specified as a document type description (DTD) which enables us to use a validating XML parser to check a trace file for syntactic correctness.

3.2 Visualisation component

Traces are converted into their graphical representation by style sheets using a style sheet processor or XSLT-processor for short [3]. Figure 3 illustrates this approach. Numerous processors

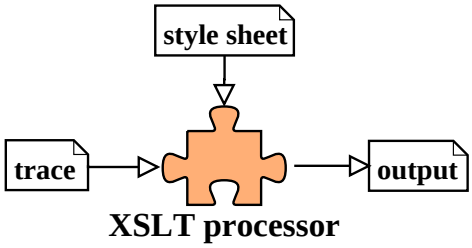


Figure 3. Conversion of traces files using XSLT-processor

are available for free already, see [14].

Using the Adobe SVG Viewer plug-in for web-browsers [7] we can zoom in and out of the graphical representation. In this way a user can gain an overview or zoom into the graphic to find detailed information. Because it is a vector based format the quality of the graphic is good at any zoom level. Two views of a trace are provided:

- *Thread view.* The view is similar to a message sequence chart, where the vertical lines represent the active time of a task. Communication events between tasks are represented by lines connecting the corresponding send and receive events (see Figure 4).
- *Object view.* The presentation is similar to message sequence charts, the vertical lines represent an object's life time.

Events in the graphic are colour coded. The use of colours makes identifying events easier. Types of events can be distinguished at a glance. The yellow boxes in Figure 4 mark local events for example. Colours are defined in a cascading style sheet that is stored in separate file and can be adjusted to personal preference.

Figure 4 shows a simple trace in thread view where all events are placed at a fixed distance. This is used when we are interested

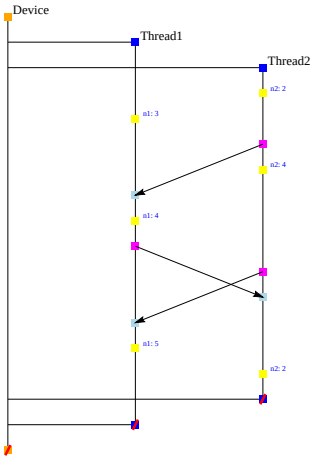


Figure 4. Thread view of a simple trace

in the sequence of events only. However we can also use the time stamps in the trace to create a graphic reflecting the time span between events as distance. Such a graphic can be useful when timing related properties are investigated.

The details-on-demand paradigm has been implemented in two ways. One is to use a separate HTML file and to link the entries from the graphic to the textual representation. In this way the graphic contains the information regarding the interaction of objects or threads and the HTML file provides the detailed view of event entries. Combining these views can be achieved by using an HTML frame.

A second implementation was considered which used the capabilities of SVG to animate text. Moving over a particular event with the mouse pointer would display information like event parameters, or messages. Moving away the pointer would hide the information again. This approach has the benefit that it uses only one file to display all the information. However with large files, the speed of opening a file and animation proved to be a performance bottleneck.

3.3 Filtering

The size of the graphic and the amount of information it represents can still present a problem. Removing irrelevant information from traces through filters could reduce the size considerably and make the relevant information stand out during visual inspection.

Following the approach for visualisation we implemented filters using style sheets and left the processing to an XSLT-processor again.

Using language constructs from XPath [4] we can express patterns of events we are looking for or want to ignore for the time being. Examples of such filters are:

- removal of local events
- selection of inter-thread communication events
- selection of inter-object communication events

Local events may not be of particular interest when looking at the communication between system components. Removing local events can be easily achieved using a style sheet as in Example 1.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

  <!-- Description: Stylesheet removes all events of type 'Local' -->

  <!-- Import standard behavoiur -->
  <!-- standard: copy all events -->
  <xsl:import href="filter_template.xsl"/>

  <!-- Add DOCTYPE -->
  <!-- create trace.dtd file -->
  <xsl:output method="xml" indent="yes" doctype-system="trace.dtd"/>
  <xsl:strip-space elements="*"/>

  <!-- Match local events        -->
  <!-- do not copy (delete) them -->
  <xsl:template match="event[@operation='Application' and @type='Local']">
  </xsl:template>
  <!-- end Match local events    -->

</xsl:stylesheet>
```

Example 1: Style sheet to remove local events

3.4 Operations over filters

The result of a filter operation is a valid trace file that can be processed further for visualisation or analysis. Therefore it is possible to use a combination of filters as indicated in Figure 5.

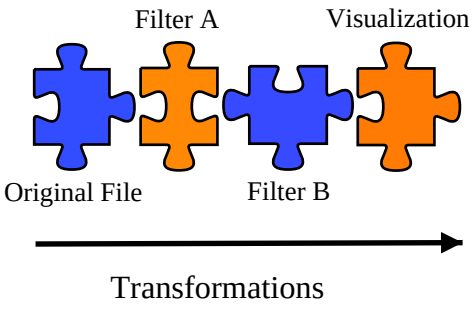


Figure 5. Sequencing of filters

The operation is called concatenation. However it may be important to note that concatenation of filters is not commutative.

$$filter1 \circ filter2 \neq filter2 \circ filter1$$

We can reuse style sheets like we did in Example 2 where we import a general style sheet in Line 8 which by default creates copies of the events in the resulting file.

We then override the rules from the general style sheet to accommodate our special needs in the particular case. Here we make use of precedence rules as defined in [3]. Rather than importing we can include style sheets like in Example 3 where in Lines 6 – 7 two style sheets are included. Rules from included style sheets have the same precedence as rules in the current style sheet. We can use these precedence orders to overload rules from imported style sheets. In this way we can reuse general patterns and thereby improve development time of our analysis tools.

3.5 Property Analysis

We can extend the ideas of using style sheets for filters to analyse properties in a trace.

- *Race analysis.* A race in a distributed system can occur if one object is accessed in more than one thread. It is a standard analysis for distributed systems.

```
 1 <?xml version="1.0" encoding="utf-8"?>
 2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 3                 version="1.0">
 4   <!-- Description: Stylesheet removes all threads with no events -->
 5
 6   <!-- Import standard behaviour -->
 7   <!-- standard: copy all events -->
 8   <xsl:import href="filter_template.xsl"/>
 9
10   <!-- Add DOCTYPE -->
11   <!-- output trace.dtd -->
12   <xsl:output method="xml" indent="yes" doctype-system="trace.dtd"/>
13   <xsl:strip-space elements="*"/>
14
15   <!-- Remove threads with no events -->
16   <xsl:template match="event[@type='Thread-Registration'
17                                    and (@operation='Create')]">
18     <xsl:variable name="found-events"
19       select="following-sibling::event[(@type='Communication' or @type='Local')
20       and parameters/@thread-id=current()/parameters/@identifier]"/>
21     <xsl:if test="count($found-events) > 0">
22       <xsl:copy-of select="."/>
23     </xsl:if>
24   </xsl:template>
25   <xsl:template match="event[@type='Thread-Registration'
26                                    and (@operation='Destroy')]">
27     <xsl:variable name="found-events"
28       select="preceding-sibling::event[(@type='Communication' or @type='Local')
29       and parameters/@thread-id=current()/parameters/@identifier]"/>
30     <xsl:if test="count($found-events) > 0">
31       <xsl:copy-of select="."/>
32     </xsl:if>
33   </xsl:template>
34 </xsl:stylesheet>
```

Example 2: Style sheet to remove "empty" threads

```
 1 <?xml version="1.0" encoding="utf-8"?>
 2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 3                 version="1.0">
 4
 5   <!-- Description: Stylesheet removes all empty threads and objects -->
 6   <xsl:include href="filter_emptythread.xsl"/>
 7   <xsl:include href="filter_emptyobject.xsl"/>
 8
 9   <!-- Add DOCTYPE -->
10   <!-- output trace.dtd -->
11   <xsl:output method="xml" indent="yes" doctype-system="trace.dtd"/>
12   <xsl:strip-space elements="*"/>
13
14 </xsl:stylesheet>
```

Example 3: Combining style sheets through include

- *Deadlock analysis.* A deadlock can occur in a system when a component is actively waiting for an event that never occurs.

Usually these properties are examined using models of the system and require formal proofs [2]. This can be very expensive and may not yield the desired results.

Our analysis is based on execution traces alone, therefore we cannot give definite answers. The results can only identify potential problems. These need to be inspected further manually. Nevertheless it should be easier after potential trouble spots have been identified. Additionally we can analyse performance bottlenecks based on the time span between trace events. Such an analysis would be difficult to perform during design phase since the necessary information may not be available.

The properties are formulated in terms of conditions over trace events.

- *Potential race condition.*

$$\begin{aligned} &\forall e \in Event \\ &\quad \text{where } e/@type \in \{Local, Communication\} \\ &T_{@object\text{-}id} = \{c/parameters/@thread\text{-}id \mid c \in Event \\ &\quad \text{where } c/@type \in \{Local, Communication\} \\ &\quad \text{and } c/parameters/@object\text{-}id \\ &\quad \neq e/parameters/@object\text{-}id\} \end{aligned}$$

then we can identify a potential race as:

$$race\text{-}condition = |T_{@object\text{-}id}| > 1$$

The condition evaluates to true if more that one thread uses the object identified by $@object\text{-}id$.

- *Highly potential race condition.* When a possible race condition is identified we can then pursue the analysis to identify a highly potential race condition. We need to establish if the identified threads are truly concurrent, i.e. the events of these threads are concurrent.

$$\forall e, e' \in Event$$
$$\text{where } e/@thread\text{-}id \neq e'/@thread\text{-}id$$
$$\text{and } e/@object\text{-}id = e'/@object\text{-}id$$

then events e and e' are concurrent according to [9] if

$$e \not\rightarrow e' \text{ and } e' \not\rightarrow e$$

- *Potential deadlock.*

$$\forall e \in Event$$
$$\text{where } e/@type = Communication,$$
$$e/@operation = Send \text{ and}$$
$$e/parameters/@thread\text{-}id = t$$

we identify a potentially locked thread by:

$$deadlock\text{-}condition = e \text{ is last event in } t$$

These general patterns can be used in any concurrent and distributed system. However there are many application specific properties which can be described by patterns, too. Such patterns can be derived from system requirements to check for correct behaviour of the system under test. Moreover patterns can also describe failures which were encountered in previous tests. In subsequent regression tests these patterns are used to check if the failure has occurred again. These application specific properties usually build the major part of trace analysis in practice. Our approach eases the description of these properties as patterns and therefore boosts productivity in the testing phase.

Because we are using syntactic patterns we are somehow limited in the description of properties. The order of events in the trace file has to reflect the happened-before relation. For example a send event has to be placed textually before the matching receive event. In our approach we have no provision to circumvent violations of this requirement. We are able to analyse only the recorded interleaving of events. The consideration of other interleavings would require the construction of a state-lattice of the trace which is expensive to obtain. We avoid it in favour of rapid analysis. It turns out that our approach suffices for many properties of interest in practice.

3.6 Deployment of trace analysis tools

Making style sheets and processor available to engineers on the project would have included the necessity of training in XSLT technology. A technology called compiled style sheet was chosen for deploying the analysis tools instead of delivering style sheets to the user (see [16, 15]). A compiler creates in this case a java archive from a given style sheet. Performing a transformation therefore meant to run a java program on the command line. This gave us the possibility to continue to use XSLT to express the properties we were looking for and being able to simply install java archives for the developers containing the implemented filters, preprocessors and converters.

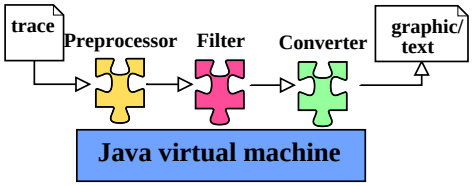


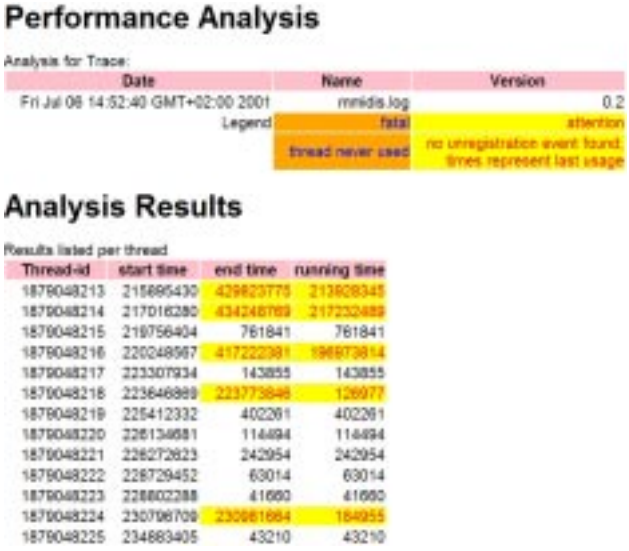
Figure 6. Deployment of trace analysis tools

4 Example

The project where the technology has been put to the test was an embedded software software system running on Windows CE. The traces were created in a proprietary format and converted off-line into the XML-based trace format.

Of particular interest became the analysis of the cause which slowed down the application considerably. The traces itself were rather large, containing millions of events. The challenge was to find calls which could have caused the slowdown.

First the performance of calls was analysed. We used a style sheet which would find a pair of call and return events and calculate the time difference. The result was presented as a HTML file.



Performance Analysis

Analysis for Trace:

Date	Name	Version
Fri Jul 06 14:52:40 GMT+02:00 2001	mmdis.log	0.2
Legend	fatal	attention
	thread never used	no unregistration event found, times represent last usage

Analysis Results

Results listed per thread

Thread-id	start time	end time	running time
1879048213	215895430	429823775	213928345
1879048214	217016280	434248769	217232489
1879048215	219756404	761841	761841
1879048216	220248987	417222381	196973814
1879048217	223307934	143855	143855
1879048218	223646889	223773846	126977
1879048219	225412332	402261	402261
1879048220	226134681	114494	114494
1879048221	226272823	242954	242954
1879048222	226729452	63014	63014
1879048223	226802288	41660	41660
1879048224	230798709	230861664	184955
1879048225	234883405	43210	43210

Figure 7. Performance analysis

After identifying the performance bottlenecks we sought to find the cause of these delays. An inclination drove us to have a closer look at error events. Error events were in this case either local events containing the word “ERROR” in the message or

communication events which contained a specific return result. Again we implemented a style sheet to filter out and visualise only those events.

The colour coding scheme, the reduced amount of information, and the ability to get a quick overview over the trace all led us to suspect an implementation flaw as the cause of the slowdown quickly.

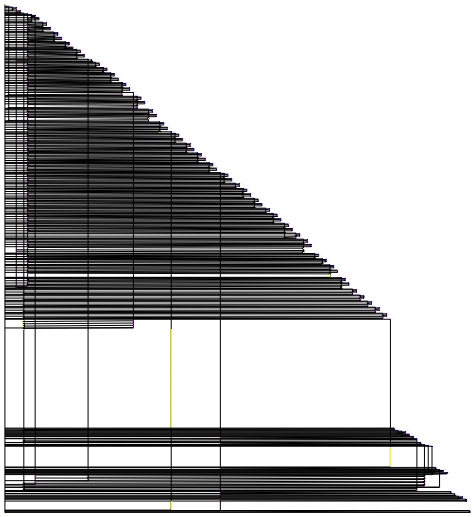


Figure 8. Finding the cause of a bottleneck

Figure 8 shows the overview the analysis presented us with. The graphic contains 26656 events and is shown at an extreme zoom level to provide an overview. A gap in the lower middle part of the trace is clearly visible which is caused by a number of clusters of error events marked yellow. We envisaged that the application is slowed down to process these unnecessary requests. A closer look revealed that a component tried to access a missing device in synchronous mode. Processing within this object paused until a timeout enabled the system to continue. Holding the state of available devices in a central component brought the desired improvement.

Starting with a trace file and not knowing for sure what we were looking for we found that this kind of support proved valuable to the project.

5 Conclusions

Using XML as the basis of a trace format for concurrent systems we gain interoperability and access to ready-made tools for trace visualisation and analysis. We created a number of style sheets which are used as a library. Combining style sheets and concatenating filters enables us to implement customised analysis tools with high speed. This enables interactive analysis of complex problems where the a designer or test engineer can request new features, receives them within a very short time and can therefore drive the analysis process.

Manual trace analysis is tedious and error prone. We provide a way to use XML tools to implement the "information seeking mantra" of overview, zoom and filter, and details-on-demand. This way trace analysis is not fully automated because the results still require an interpretation by the user. The ability to create customised filters at high speed not only provides for interactive analysis but also keeps costs of development low.

Performance could be an issue with certain filters. However this will improve by using a faster computer, anticipation of a faster XSLT-processor or use a dedicated database with an XML-interface to store trace events. We are interested to continue the research into properties which are of interest for trace analysis, to improve our library by adding more filters, and converters to be used on future projects.

References

[1] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 — W3C recommendation 10 February 1998. Technical Report REC-xml-19980210, World Wide Web Consortium, Feb. 1998.

[2] K. M. Chandy and J. Misra. Deadlock absence proofs for networks of communicating processes. *Information Processing Letters*, 9(4):185–189, Nov. 1979.

[3] J. Clark. XSL Transformations (XSLT) 1.0 — W3C recommendation 16 November 1999. Technical Report REC-xslt-19991116, World Wide Web Consortium, nov 1999.

[4] J. Clark and S. DeRose. XML Path Language (XPath) 1.0 — W3C recommendation 16 November 1999. Technical Report REC-xpath-19991116, World Wide Web Consortium, Nov. 1999.

[5] K. Grabenweger, E. Reyzl, and H. Sauer. Trace-based testing of middleware. In *5th Conference on Quality Engineering in Software Technology*, Nürnberg, Germany, 2002.

[6] H. Hallal, A. Petrenko, A. Ulrich, and S. Boroday. Using SDL Tools to Test Properties of Distributed Systems. In E. Brinksma and J. Tretmans, editors, *Formal Approaches to Testing of Software — FATES '01*, pages 125–140, Aalborg, Denmark, August 2001.

[7] A. S. Incorporated. SVG developer tutorial. `http://www.adobe.com/svg/basics/intro.html`, July 2001.

[8] O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Computer Aided Verification*, pages 172–183, 1999.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, pages 558–565, July 1978.

[10] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[11] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343, Washington, Sept. 3–6 1996. IEEE Computer Society Press.

[12] `http://www.w3.org/Graphics/SVG/SVG-Implementations.htm8`.

[13] W3C. Scalable Vector Graphics (SVG) 1.0 specification — W3C proposed recommendation 19 July, 2001. Technical Report PR-SVG-20010719, World Wide Web Consortium, July 2001.

[14] `http://www.xmlsoftware.com/xslt/`.

[15] `http://xml.apache.org/xalan-j/`.

[16] `http://www.sun.com/software/xml/developers/xsltc/`.