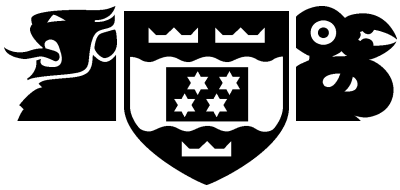
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Program Trace Formats for Software Visualisation

Craig Anslow, Stuart Marshall, James Noble, Kirk Jackson, Mike McGavin, and Robert Biddle

Technical Report CS-TR-06/1
May 2006

Abstract

Developers must understand a software component or piece of code before they can reuse it. Software visualisation has the potential to assist this understanding by visualising the internal structure and behaviour of components. However it is difficult to create visualisations. We create visualisations by test driving reusable components, and store the output in a formal, transportable, and filterable program trace format. Using our intermediary program trace formats, static and dynamic information of a component can be transformed into useful visualisations either at real time or in the future for developers to understand.

Author Information

James Noble is a Professor in Computer Science and Stuart Marshall is a lecturer at Victoria University of Wellington (VUW). Robert Biddle is a Professor in Human Computer Interaction at Carleton University. Craig Anslow, Kirk Jackson and Mike McGavin are MSc thesis students at VUW.

1 Introduction

To reuse a component (an existing piece of code or a component of a program), a developer must first understand what a component does, how it works, and how it can be reused. One way to understand how to reuse a component is to read the documentation. Another is to create visualisations of the static and dynamic behaviour of a component to enhance a developer's understanding of the internal structure and behaviour of a component.

To visualise the behaviour of a software component, certain information needs to be extracted from it. Extracting information from an executing component and gathering it in a program trace or execution trace format is difficult. One method for deriving this information is to "spy" on applications as they execute. We do this by using debuggers or modified execution environments [10, 20], so we can generate static and dynamic information such as class descriptions, method calls, method returns, field accesses, and field modifications as the component runs.

Previous work [13, 14] identified that we needed a representation or format of the information produced from test driving, known as a program trace or execution trace. In this paper we examine two formal, transportable, and filterable program trace languages that we have developed the Process Abstraction Language (PAL), and the Component Trace Language (CTL). We also show the use of the program trace languages in our visualisation tools to create visualisations of reusable components either at real time or in the future for developers to understand.

Our paper is organised as follows. In section 2, we state our motivation for program trace languages. In section 3 and 4 we describe our program trace languages. In section 5 we discuss the merits of the program trace language formats and show some of the visualisations that we have created using our program trace languages. In section 6 we address related work, and summarise the paper in section 7.

2 Visualisation Architecture for REuse (VARE)

We have created the Visualisation Architecture for REuse (VARE) [13], which is used for web-based visualisation of remotely executing object-oriented software. VARE is based on the Program Mapping Visualisation (PMV) conceptual model for describing program visualisation systems [26, 24], see figure 1. The program component collects information about a program being analysed. The mapping layer then translates this information to a format that is useful for the visualisation component. The visualisation component displays visual representations of the program.

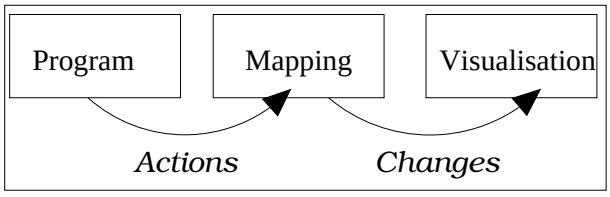


Figure 1: The PMV Model.

VARE is a client-server architecture, see figure 2. The server contains repositories and processes. With the client, the user manages the activities associated with creating and viewing a visualisation. The component repository interface lets the user select a component from the repository to create a component set. Once this is created, the user can select an engine

type from the engine repository to control the test driving of these components. Test driving is defined as "specifying a sequence of method invocation and field access/modifications and then executing the sequence on a component" [10]. The engine component corresponds to the program component in the PMV model.

The engine generates a program or test drive trace as output, which is stored in the test drive report repository. A program trace contains all information required to describe a program execution such as the order of object creation, method invocations, field accesses and field modifications. It is then used as input to a transformer, which corresponds to the mapping component in the PMV model. The transformer repository interface lets the user select the transformer to use and the program trace to use with it. The transformer then transforms the program trace into an appropriate visualisation.

The finished visualisation is stored in the visualisation repository. The visualisations contain a description of the components they are associated with, the user who created them, and notes that help the understanding of the visualisation. The visualisation interface lets a user choose a particular visualisation and control its presentation.

Mehner [17] claims that a non-proprietary intermediary schema is needed for describing program traces. We have created two sets of requirements for program trace languages [12]. One is for the types of information that could be visualised, and the second for the output of an engine which is used as input to a transformer.

2.1 Information to be Visualised

Our first set of requirements focuses on three different types of information: what a component does, how a component works, and how a component can be reused.

It is important to understand what a component does by treating it as a black-box, looking at external side-effects and the results that occur as a result of interacting with the public interface. By understanding what a component does we may see if the component can be reused in a new context.

Understanding how a component works by looking at the internals of a component, treating it as a white-box, is useful for visualising because it may open up opportunities for modifying the components behaviour to what is required by replacing sub-components, extending components or overloading methods.

Understanding how a component can be reused or modified in a context that it may not have been designed for is important to developers. Visualisations of the uses of a component through its public interface may help show how the component can be reused.

Some software visualisation systems have their visualisation tool built directly into the information gathering tool. In VARE we have separated the engine from the transformer. Having the engine and the transformer separate allows different visualisation tools to use the same program trace to create different visualisations, while a single visualisation tool can make use of many program traces from multiple engine tools. Making the separation means that the program traces must be transported from the engine to the transformer in a platform-independent and accepted format.

2.2 Program Trace Format

Our second set of requirements for a program trace format is that it has a formal specification, transportable over the Internet, and filterable, so that information can be extracted and specific details can be retrieved. It must also be storable, re-playable, and support live streaming to visualisation tools. Finally the program trace format must be programming-language independent, platform-independent, and scalable to large components and long visualisations. We have decided on exploring XML to encode these program traces.

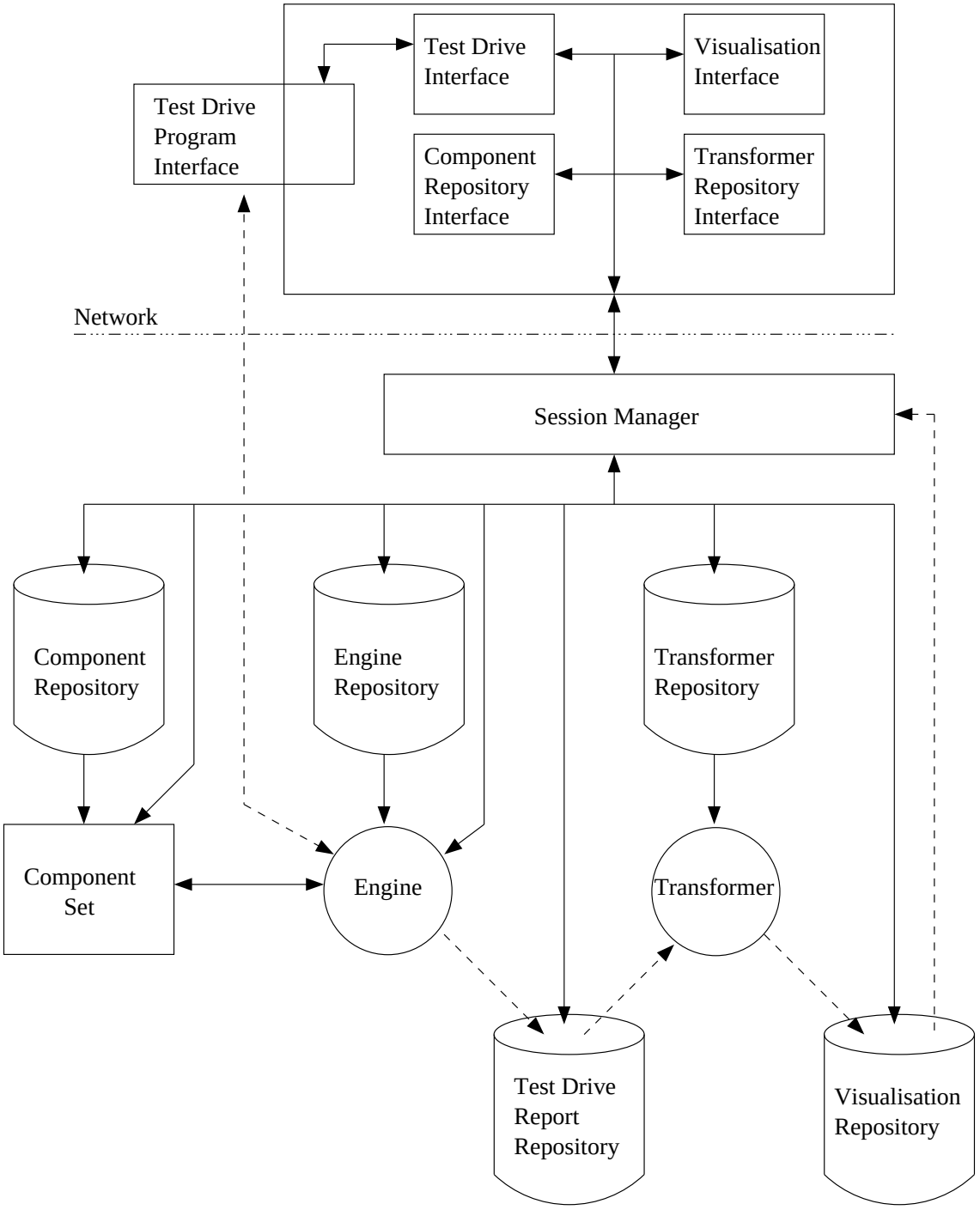


Figure 2: The VARE architecture is based on a client/server model, with the server being split into repositories and processes. Dashed lines represent test drive or visualisation input/output, while solid lines represent control, queries or responses.

The XML specification [33] was primarily developed to make it easier to exchange information in an open and understandable way, with communication technology such as the Internet in mind. The Simple Object Access Protocol (SOAP) [32] can be used for remote method invocation. The following design goals given in the XML specification are the most relevant reasons for considering the use of XML to represent program traces:

- *That the design of an XML document should be formal and concise.* Program traces should be designed in a formal way.
- *That XML documents should be straightforwardly usable over the Internet.* Program traces should be transportable over the Internet.
- *That it should be easy to write programs which process XML documents.* A program trace should be filterable by visualisation tools and XML parsers such as SAX [16] and DOM [30].

This paper focuses on the intermediary program trace format from an engine to a transformer in VARE. The next two sections describe our experiences with our two XML based program trace languages: the Process Abstraction Language (PAL) and Component Trace Language (CTL). Both languages have features for representing both static and dynamic information.

3 Process Abstraction Language (PAL)

PAL [13, 14] is an XML specification for object models and is designed for streaming information to visualisation tools to generate useful visualisations. PAL describes object-oriented programs and has been used for representing information that can be extracted from the execution of C++ and Java programs by using debugging technologies.

PAL has elements for describing classes, super-classes, methods, and fields. PAL also describes the dynamic behaviour of programs, including objects, run-time representations of classes, method calls with their arguments and return values, and different threads of control.

A PAL session begins with an XML pal root element. The pal element always has a version number, and the value of it is required to be the correct value by the DTD of that version. A PAL session also has an identifier and may have a session name to distinguish itself from other PAL sessions. All execution information is contained within the pal element. Once it is finished, the end tag is output and the session has ended. The following sections describe the static and dynamic information of a PAL session.

3.1 Static Information

Immediately after the start of a PAL session, a list of available types from a program can be given. The type collection element, see figure 3, contains a brief list of type specifications about each type in a program, informing the client of types that it may request more information about during the course of a session. Each type is labeled with a classification category that says what sort of type it is, and has a unique identifier.

The information contained in a type element is that of super classes, methods, and fields. Figure 4 shows a definition of a class called SimpleClass which inherits EvenSimplerClass, as defined in the type collection element. Information about EvenSimplerClass could also be requested if required. SimpleClass contains a constructor, set() and get() method, and a field.

```
<typecollection>
 <typespec classification="generic" idref="t9"
  name="int"/>
 <typespec classification="class" idref="t1"
   name="EvenSimplerClass">
   <context contextname="sourcefile"
    contextvalue="tp6.cc"/>
 </typespec>
 <typespec classification="class" idref="t2"
   name="SimpleClass">
   <context contextname="sourcefile"
    contextvalue="tp6.cc"/>
 </typespec>
</typecollection>
```

Figure 3: PAL typecollection element.

3.2 Dynamic Information

Dynamic information is collected in an execution element, which may have an identifier to allow different executions to be distinguished. The execution identifier can be generated from the session identifier. Some dynamic elements that can be generated are the event, new class instance, delete class instance, method call, and method return elements. An event element must contain an element inside to represent specific information about the event.

When a new object is instantiated, a new class instance event is generated. The new class instance element describes both a unique identifier that will be used for referencing this class in the future, and a type reference to define what class the object is an instance of. When an object is deleted, a delete class instance event is reported.

Once a class has been instantiated, methods can be called on it. We show these as event elements with method call elements inside them. The method call element attributes can provide:

- method call id: A unique identifier.
- method id reference: A reference to the identifier of the method being called, as described within the type element of the associated class.
- class instance id reference: A reference to the class instance that the method is called on which may be absent if the method is static.
- thread number: An identifier referring to the thread. All method calls with the same thread identifier are being called within the same thread of the program execution.
- caller position reference: An identifier representing the position in the program that the method was called from, within the calling method. Two method calls that cite the same caller position reference have been called from the same static location in the program code.
- caller class instance id reference, caller method call id reference, caller method id reference: Identifiers referring to the dynamic event that caused this method to be invoked, if any. An identifier is provided for both the class instance, and the method invocation identifier. A third attribute is available to reference the static identifier of the method

```
<type name="SimpleClass" typeid="t2">
 <context contextname="sourcefile"
  contextvalue="tp6.cc"/>
 <classdata>
  <superclasses>
   <superclass access="public">
    <typeref typeidref="t1"/>
   </superclass>
  </superclasses>
  <methods>
   <method access="public" methodid="f23"
    name="SimpleClass">
     <modifier name="constructor"/>
   </method>
   <method access="public" methodid="f25" name="setA">
     <typeref>
      <modifier name="pointer"/>
     </typeref>
     <argument argumentid="a24">
      <typeref typeidref="t9"/>
     </argument>
   </method>
   <method access="public" methodid="f27" name="getA">
     <typeref typeidref="t9"/>
   </method>
  </methods>
  <variables>
   <variable access="private" name="_a" variableid="v17">
     <typeref typeidref="t9"/>
    </variable>
  </variables>
 </classdata>
</type>
```

Figure 4: PAL type element.

itself. This is so it is still possible to have some information about the caller, even if the calling of that method was not available for some reason.

Inside the method call element, there may be an argument values element. Within the argument values element, the values passed to the method as arguments are given. The specific argument to which that each value refers is specified by the argument id reference attribute of the argument value element. The value element inside may contain either a raw value that contains some representation of the actual value being passed, or an abstract value, which can refer to the identifier of a class instance.

Figure 5 shows the setA() method called on a SimpleClass object. The caller method is unknown in this simulation so those attributes do not appear, but the static location of the program code that the method was called has been labeled. Any other method invocations whose call position reference attribute is the same have been called from exactly the same point. Finally, the value 55 is being passed to the argument and shown inside a raw value element. If an object was being passed to the method, it could have been reported using an abstract value element. Instead of giving raw information such a memory address, an abstract value element has attributes that reference the object using its previously announced identifier. A visualisation tool can use abstract value elements to keep track of object references as they are passed around the program.

```
<event eventid="51">
 <methodcall callpositionref="dp47"
  classinstanceidref="dcl40" methodcallid="dmc49"
  methodidref="f25" threadnum="1">
  <argvalues>
   <argvalue argumentidref="a24">
    <value>
     <rawvalue>
      55
     </rawvalue>
    </value>
   </argvalue>
  </argvalues>
 </methodcall>
</event>
```

Figure 5: PAL methodcall element.

The method call event only indicates when a method has been called, but many other events can happen before it returns. When the method does return, a method return event is generated. Figure 6 shows the method return of setA().

We conducted a few trials with PAL to test how it handled moderate sized implementations. We generated PAL files for some sample program executions. These executions were of Java applications ranging from 100-1000 lines of code and made use of the JDK standard libraries. Each PAL file was in the range of tens-of-megabytes to hundreds-of-megabytes in size. These trials identified the weaknesses of PAL. First it combines both static and dynamic information, and second allows for the storage of the entire information set needed to describe a specific test drive. This leads to redundancy in situations where we were storing multiple test drives. The static information which is built on static program code is identical when we test drive the same component multiple times. The component may also depend

```
<event eventid="52">
 <methodreturn methodcallidref="dmc49">
  <returnvalue>
   <value>
    <rawvalue>
     0x0
    </rawvalue>
   </value>
  </returnvalue>
 </methodreturn>
</event>
```

Figure 6: PAL methodreturn element.

on other common components that are shared by other components likely to be test driven. One example of this is the libraries packaged in the JDK.

4 Component Trace Language (CTL)

To remove the redundant information in the PAL program traces, we decided it was necessary to store static information relevant to a single component separately from the dynamic information relevant to a single test drive. The result was the Component Trace Language (CTL) which is comprised of Reusable Component Descriptions (RCD, for static information) and eXtensible Trace Executions (XTE, for dynamic information).

There are many benefits to this approach. Firstly, the files are smaller, which is beneficial both in storage and in transportation costs. Secondly, we have separate files for each component as there are no easy ways of identifying which version of a component was contained within a PAL file. Thirdly, there was no easy means of identifying which types comprised a component, and which types were used by the component. The next two sections describe RCD and XTE.

4.1 Reusable Component Descriptions (RCD)

The information included in RCD consists of the names of any included packages, and the names of any classes within those packages. RCD can be created by analysing a component's compiled or source form. Figure 7 shows the test_classes package and the class TestClass1.

```
<rcd>
 <rcd:package
  xmlns:rcd="http://www.mcs.vuw.ac.nz/~stuart">
 <rcd:packagename>test_classes</rcd:packagename>
 <rcd:class>
 <rcd:classname>TestClass1</rcd:classname>
 <rcd:superclassname>java.lang.Object
  </rcd:superclassname>
...
</rcd>
```

Figure 7: RCD package and class elements.

```
<rcd:methods>

 <rcd:constructor>
  <rcd:modifiers abstract="false" access="public"
   final="false" interface="false" native="false"
   static="false" strictFP="false" transient="false"
   volatile="false" synchronised="false"/>
 </rcd:constructor>

 <rcd:method>
  <rcd:type>
   <rcd:name>java.lang.String</rcd:name>
  </rcd:type>
  <rcd:modifiers abstract="false" access="public"
   final="false" interface="false" native="false"
   static="false" strictFP="false" transient="false"
   volatile="false" synchronised="false"/>
  <rcd:methodname>method1</rcd:methodname>
 </rcd:method>

 <rcd:method>
  <rcd:type>
   <rcd:name>void</rcd:name>
  </rcd:type>
  <rcd:modifiers abstract="false" access="public"
   final="false" interface="false" native="false"
   static="true" strictFP="false" transient="false"
   volatile="false" synchronised="false"/>
  <rcd:methodname>main</rcd:methodname>
  <rcd:argument>
   <rcd:argumentname>args</rcd:argumentname>
   <rcd:type>
    <rcd:name>java.lang.String[]</rcd:name>
   </rcd:type>
  </rcd:argument>
 </rcd:method>

</rcd:methods>

<rcd:fields>

 <rcd:field>
  <rcd:type>
   <rcd:name>java.lang.String</rcd:name>
  </rcd:type>
  <rcd:fieldname>_field</rcd:fieldname>
  <rcd:modifiers abstract="false" access="private"
   final="false" interface="false" native="false"
   static="false" strictFP="false" transient="false"
   volatile="false" synchronised="false"/>
 </rcd:field>

</rcd:fields>
```

Figure 8: RCD method and field elements.

For each class listed we include the interface description, such as it's place in the type hierarchy, and the signatures of all operations and attributes. RCD also contains information for each component, such as the component name, the component author and a natural language description of the component's capabilities. Figure 8 shows the method and field

elements of TestClass1.

The interface description is important because the information contained will later be referenced by separate XTE program traces. This information will help identify different versions of the same component, and provide some background on the components implementation and purpose. The separation of static from dynamic information gives us another important benefit. We can describe a component before a test drive is even performed. The benefit of this approach is that a developer does not need to immediately test drive a component and can do the test driving at a later stage.

RCD also allows each class within a component to list its dependencies. We intend that this be comprised of a list of other classes, both internal and external to the component, that this class's implementation or interface requires to be present. This can be useful to a developer as it helps them identify what else needs to be brought in for a test drive, or for integration into the final system.

4.2 eXtensible Trace Executions (XTE)

XTE stores test drive information from RCD. The test driven components are referenced inside XTE, so that the relevant RCD file can be located, loaded and used in the subsequent creation of the visualisation. Figure 9 shows the start of an XTE trace from the previous RCD figures which includes the name of the test driver and the date the test drive was performed. This information is useful when a test drive is to be reviewed sometime after its initial performance.

XTE has multiple event types that are nested in an execution element and include: object creation, method calls, method returns, field accesses, field modifications, exception throws and catches, and security permission requests. Each event contains a thread ID providing some support for clearly documenting multi-threaded components. Figure 10 shows the creation of the TestClass1 object and method1() being called.

The method and object creation elements contain information to identify the class or operation information stored in the associated RCD file. For instance, in the case of the method call element, the name of the method, the object invoked on, and the list of parameter types is included along with the parameter values. The combination of the method name, and the type list is sufficient to identify the unique operation signature in a static RCD file once the object id has been used to resolve what class the method was called on. However, information on the exceptions throw-able by the method, or the return type associated with the method are not stored in the method call elements within XTE. This information is stored in the RCD file instead.

We have also added in field access and field modification elements to XTE, see figure 10. Developers can now note when an operation is affected by the particular state of an object, or affects the particular state of an object. This can be useful for two reasons. Firstly, it can show when an operations behaviour may differ if the developer changes an object's state before that time, allowing future test drives to be guided by lessons learnt from earlier test drives. Secondly, it can further indicate what the side-effects of particular sequences of operations are, and how one sequence of operations may configure data that affects a later sequence of operations.

The field access element provides information on the name of the field accessed, the object that field is in, and the value of the field at that point in time. The field modification element provides similar information, but includes both the old value and the new value after modification.

Other features of XTE that are not in PAL include security permissions and exception handling. Describing security permissions shows what resources and capabilities a component

```
<xte xmlns:xte="http://www.mcs.vuw.ac.nz/~stuart">
 <xte:creator>alias</xte:creator>
 <xte:date>Thu Jun 24 15:06:06 NZDT 2004</xte:date>

 <xte:object objectid="object1">
   <xte:complextype>
     <xte:name>test_classes.TestClass1</xte:name>
   </xte:complextype>
   <xte:field>
     <xte:objectvalue xte:id="object2"/>
     <xte:name>_field</xte:name>
   </xte:field>
 </xte:object>

 <xte:object objectid="object2">
   <xte:complextype>
     <xte:name>java.lang.String</xte:name>
   </xte:complextype>
 </xte:object>

 <xte:object objectid="object3">
   <xte:complextype>
     <xte:name>java.lang.String</xte:name>
   </xte:complextype>
 </xte:object>

...
</xte:xte>
```

Figure 9: XTE starting elements.

```
<xte:execution>

 <xte:objectcreation xte:objectid="object1">
  <xte:complextype>
   <xte:name>test_classes.TestClass1</xte:name>
  </xte:complextype>
 </xte:objectcreation>

 <xte:fieldmodification xte:objectid="object1">
  <xte:fieldname>_field</xte:fieldname>
  <xte:oldvalue>
   <xte:objectvalue xte:id="object3"/>
  </xte:oldvalue>
  <xte:newvalue>
   <xte:objectvalue xte:id="object2"/>
  </xte:newvalue>
 </xte:fieldmodification>

 <xte:methodreturn>
  <xte:primitivevalue/>
  <xte:methodname>test_classes.TestClass1
  </xte:methodname>
  <xte:typename>test_classes.TestClass1</xte:typename>
 </xte:methodreturn>

 <xte:methodcall xte:receiverid="object1"
  xte:senderid="">
  <xte:methodname>method1</xte:methodname>
  <xte:typename>test_classes.TestClass1</xte:typename>
 </xte:methodcall>

 <xte:fieldaccess xte:objectid="object1">
  <xte:fieldname>_field</xte:fieldname>
  <xte:objectvalue xte:id="object2"/>
 </xte:fieldaccess>

 <xte:methodreturn>
  <xte:objectvalue xte:id=""/>
  <xte:methodname>method1</xte:methodname>
  <xte:typename>test_classes.TestClass1</xte:typename>
 </xte:methodreturn>

</xte:execution>
```

Figure 10: XTE execution elements.

is trying to access outside of its own implementation. A developer can then decide on whether or not the component under inspection will be able to operate in the context needed, and whether or not the component tries to perform actions that may compromise the security of the system it would be integrated into. The specific information stored is the location that the permission was requested from, a natural language description of the permission, and a keyword identification of the permission type. Some permission types include: "read from network", "write to network", "access to file", "modify file", and "execute program".

Tracing exception handling shows how a system making use of the component can detect and handle events at the boundaries of typical behaviour. A reliable and robust system should have the ability to handle exceptional behaviour and to support recovery or error reporting. If a developer wishes to integrate a component into such a system, they would need some indication that the component would not break these non-functional requirements. The list of possible exceptions that could be thrown by a particular operation is listed in the associated RCD file.

Our testing on the size of the program traces once compressed has shown that CTL is approximately half the size of PAL program traces when using small program traces. However testing the scalability of the program traces for large components and long visualisations has yet to be explored.

5 VARE Visualisation Tools

In this section we describe our VARE visualisation tools that use C++, Java, and C# components to create visualisations in Scalable Vector Graphics (SVG) [31], Javascript, Python and Tk, using PAL and CTL program trace languages.

Abstraction Tool (AT) [13] is a prototype that has been developed to extract information from applications and present the information using PAL. AT is an implementation of an engine from VARE. AT test drives programs written in C++, using the GNU Debugger (GDB) [8]. It is written in the Python scripting language. The main tasks of AT are to drive GDB, and to output PAL based on what was seen during execution. AT also uses SOAP for remote method invocation, to allow AT to be controlled by another application. Figure 11 shows the available classes and the messages for each type which have been test driven.

Spider [11] is a prototype for exploring and documenting reusable components in a web environment. Spider is an implementation of an engine from VARE. Figure 12 shows a test drive of the Java Calendar component. Spider documents components with RCD and test drive traces with XTE by interpreting information stored in a component, detecting events in the run-time environment, and interrogating the runtime environment's state. Spider uses the Java Reflection [19] and Java Debugger Interface [18] technologies to extract information.

VARE.NET is a prototype to show that Microsoft technologies can enable the Internet distribution of the test driving process. VARE.NET is an implementation of the VARE architecture. Figure 13 shows a test drive of the Microsoft.NET's System.Windows.Forms.Form component. The VARE.NET engine captures C# program events by using RAIL [4] to rewrite CIL byte code into RCD and XTE program trace formats. The test drive interface is an ASP.NET application and test driven components can be viewed remotely by using SOAP for remote method communication.

XML Data Storage Environment (XDSE) [2] is a prototype used for storing and retrieving PAL, RCD and XTE program traces. XDSE is an implementation of the test drive report and visualisation repositories from VARE. Figure 14 shows a query which produces an RCD file. The main feature of XDSE is to store PAL, RCD and XTE program traces and then query them when required to generate a visualisation. XDSE is implemented with a native XML database, SOAP, Apache Tomcat, and XQuery.

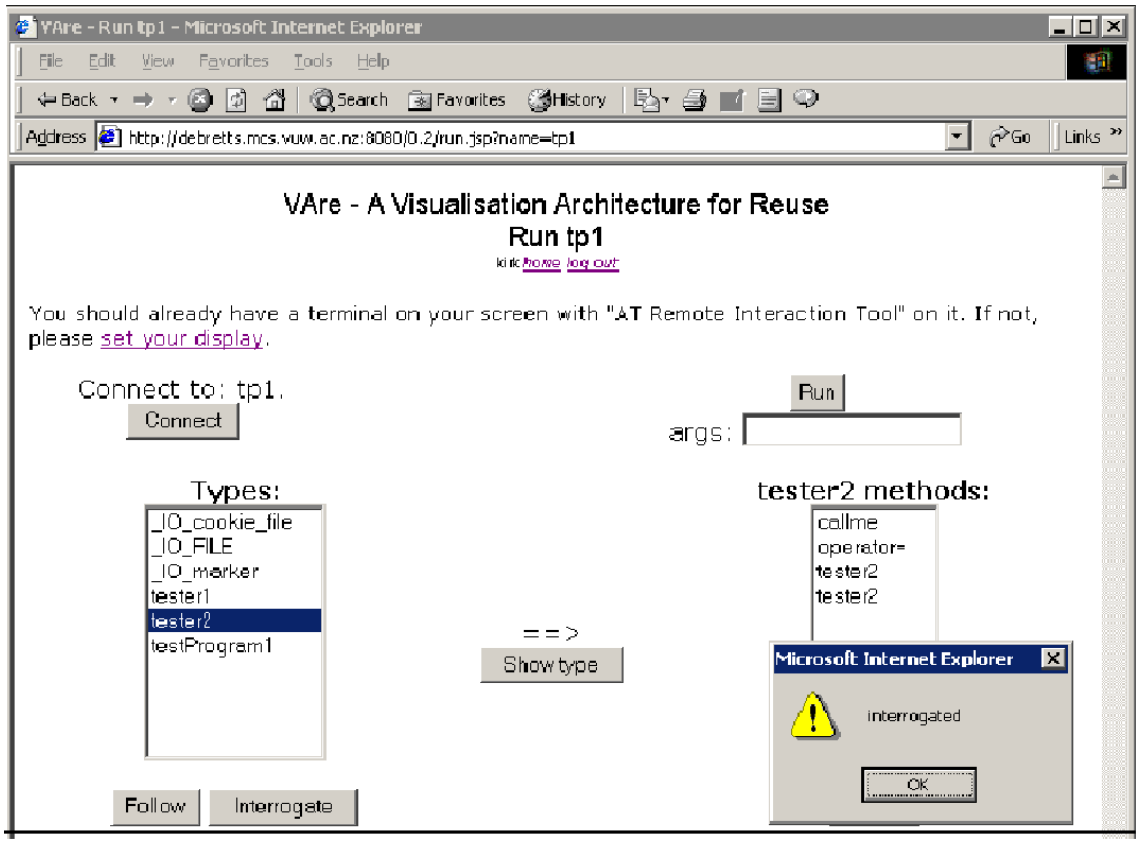


Figure 11: AT, along with a Javascript and SOAP interface test driving a simple C++ program.

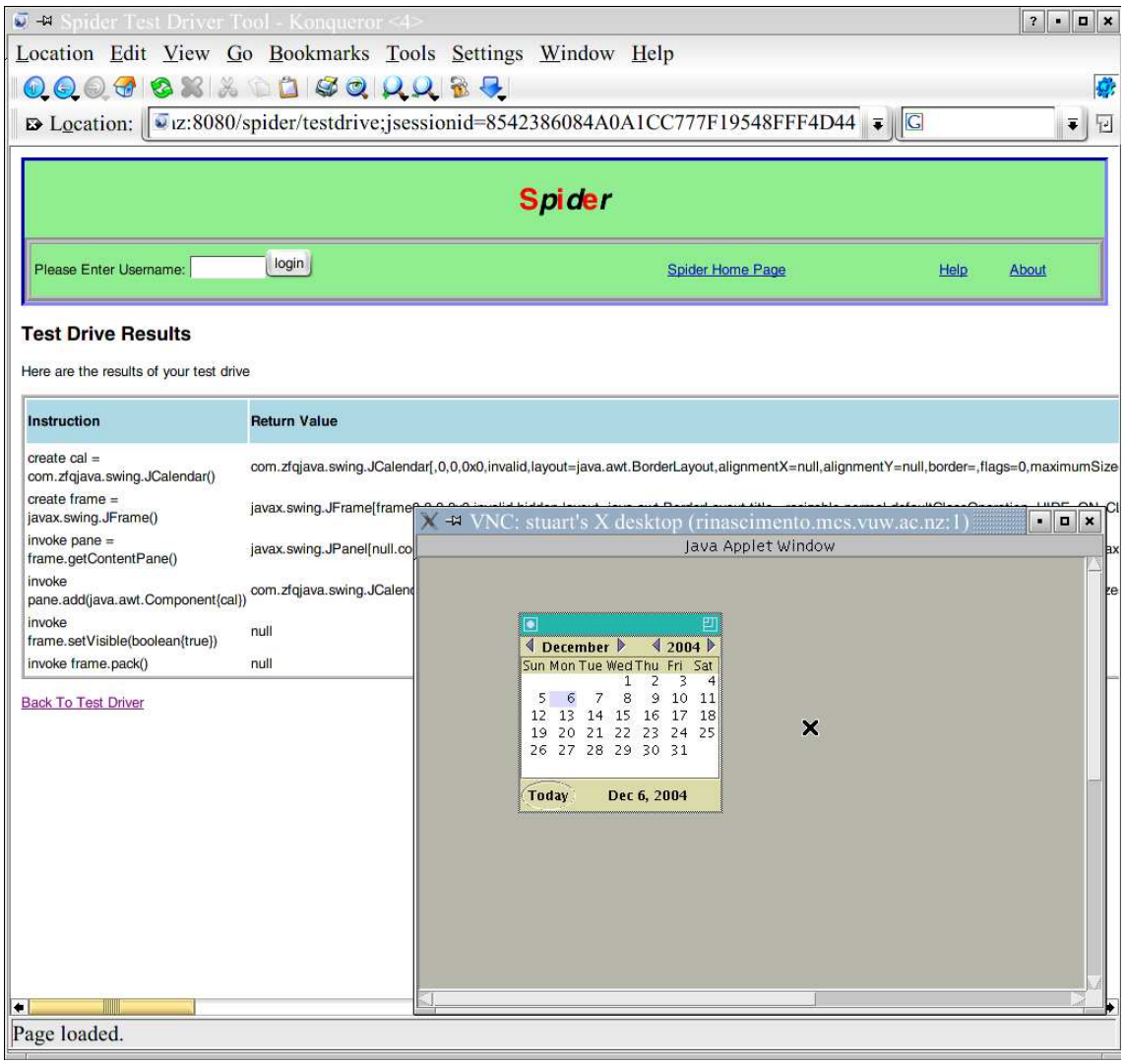


Figure 12: Spider, a test drive of the Java Calendar component.

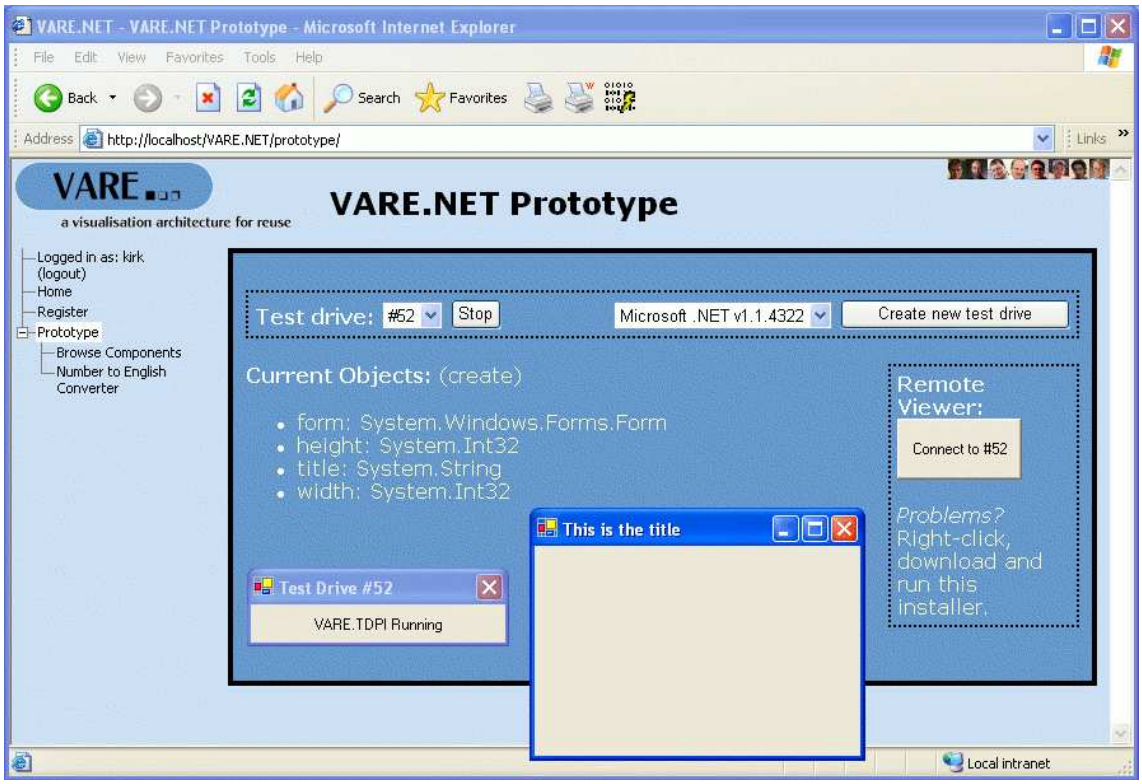


Figure 13: VARE.NET, remotely testing Microsoft.NET's System.Windows.Forms.Form component.

The program traces are filterable for extracting relevant information to create a visualisation. We query our program traces using the defacto XML query language, XQuery [29]. Sample queries include retrieving the classes in a program along with the classes methods and fields, or all the events where an object interacts with other objects.

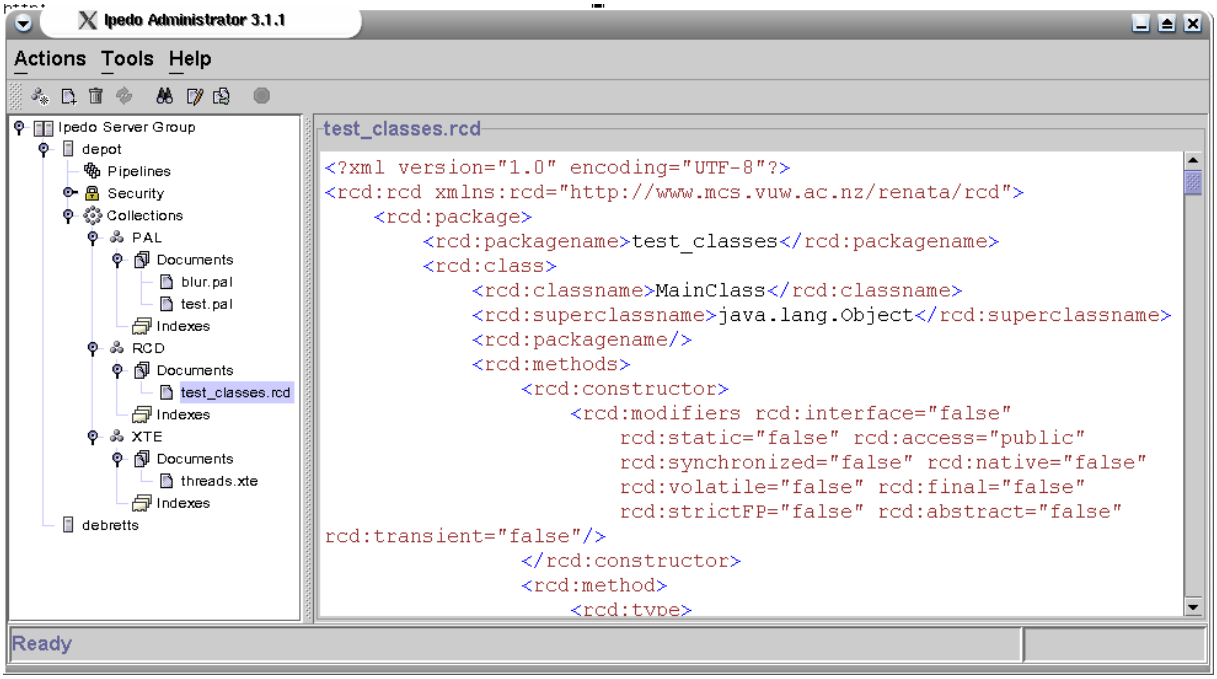


Figure 14: XDSE, a query which produces an RCD file.

Blur [6] is a prototype for producing SVG visualisations. Blur is an implementation of a transformer from VARE. Blur takes a program trace as input and transforms it into an SVG visualisation for viewing over the web. Blur is implemented as a Java Servlet running a version of Apache Tomcat.

Figure 15 shows an SVG interactive UML class diagram from a PAL program trace generated by Blur. When the mouse covers a piece of code in the right hand side frame, the left hand side highlights the appropriate class or method in the UML class diagram. This is a helpful tool for developers, because it shows where the code is located in a file, and how it is associated with other classes in a program.

Figure 16 shows an SVG interactive UML sequence diagram generated by Blur from dynamic information found in the same PAL program trace that we show in figure 15. The sequence diagram is interactive and allows the user to navigate, zoom-in-out, and fold and unfold call sequences to better understand the behaviour or interactions between the classes.

VET (Visualisation of Execution Traces) [15] is a prototype visualisation application that lets users interact with and understand execution traces (also known as program traces). VET follows the methodology of show all data of an execution trace, then let users filter out unwanted data, and provide details on demand. VET uses filters so that users can directly adjust the information being displayed by the visualisations. VET is a plugin based architecture which allows users to design their own visualisation plugins, so that a user can decide what to draw and when particular events occur. Users can also design their own filters.

Figure 17 shows two visualisations, the top one is a sequence diagram while the bottom one is an association diagram. The two visualisations are synchronised at the same location within the execution trace and also the information that is displayed. VET is a useful tool

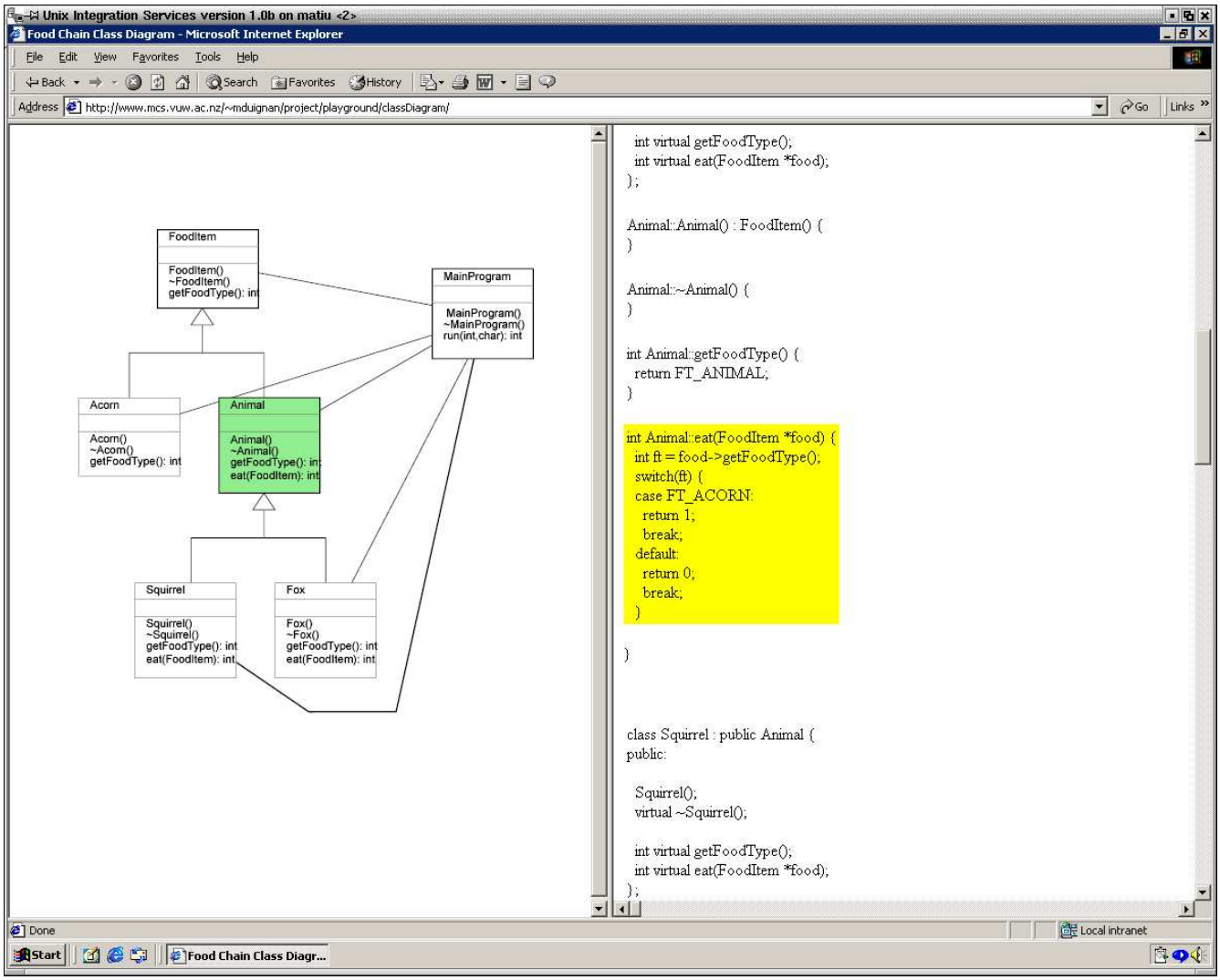


Figure 15: Blur, an SVG interactive UML class diagram. When the mouse covers a piece of code in the right hand side frame, the left hand side highlights the appropriate class or method in the class diagram.

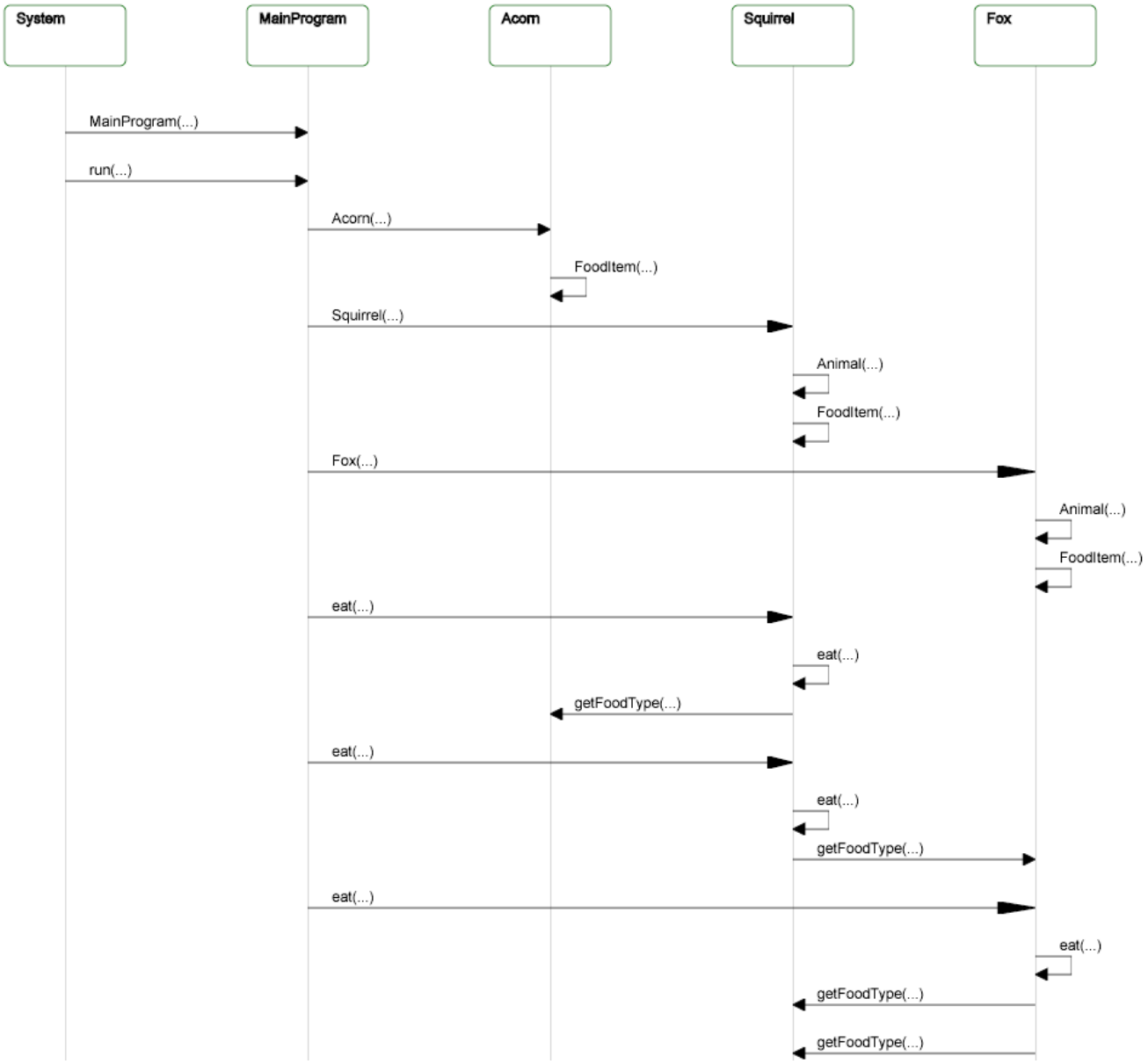


Figure 16: Blur, an SVG interactive UML sequence diagram that allows the user to navigate, zoom-in-out, and fold and unfold call sequences.

as it can show mulitple visualisations of the same execution trace, hence developers can see the trace from different perspectives.

VET is built using the Python scripting language and the Tk toolkit via Python's Tkinter module. A Python-based API is provided for VET plugins, providing abstract wrapers around all graphical operations and exceution trace information.

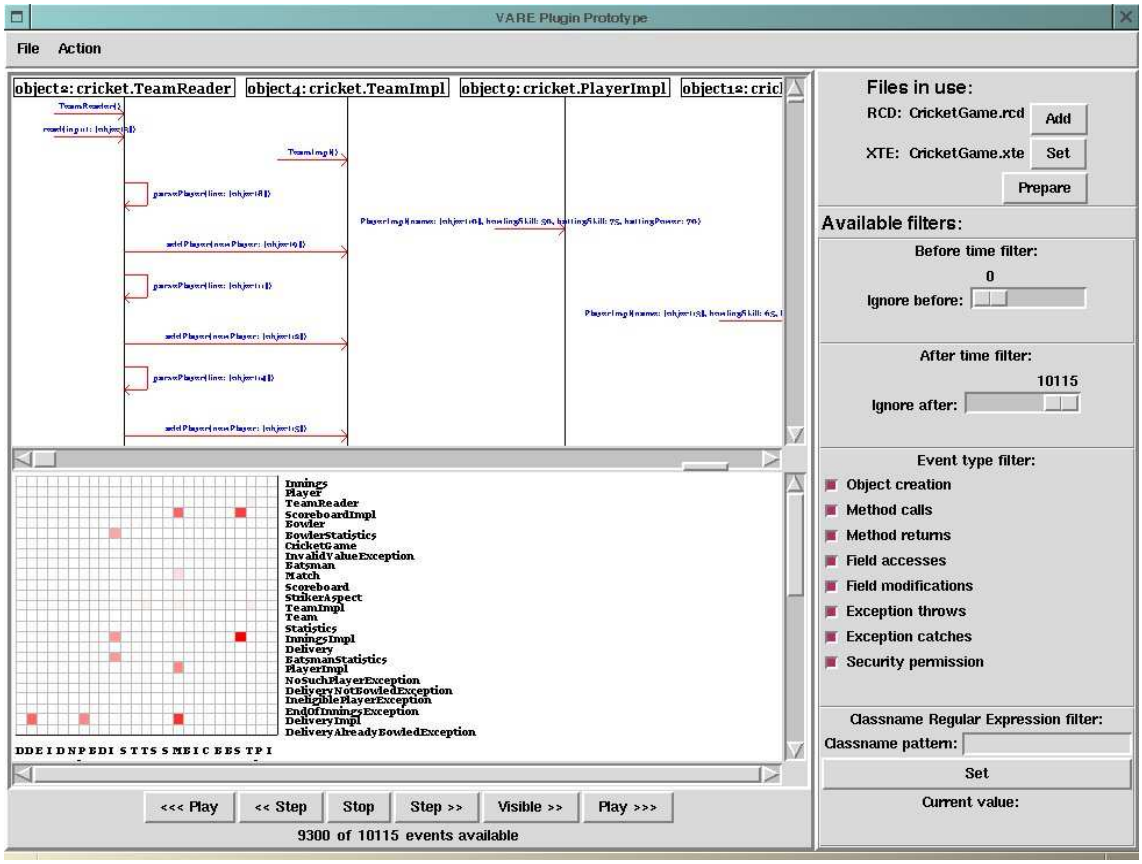


Figure 17: VET displays a sequence and an association diagram of the same data from an execution trace.

6 Related Work

Many software visualisation systems exist that are dedicated to creating visualisations using program traces, and several overviews are available [27, 5, 34, 7].

BLOOM [23, 35] is a system for understanding software through visualisation and stores program traces in two files. One file contains the trace data in a compressed binary format. This file consists of a series of records indicating the following types of events: entry and exit of a method, the amount of memory allocated to an object, the amount of time an object takes to execute, the amount of time an object waits for a resource, when an object is created and deleted, and when memory is freed. The second file contains records describing the classes, methods, and objects accessed in the trace. XML files are used to store the analyses of the trace data.

Jinsight [9, 21, 22, 25] is a tool for visualising and analysing the execution of Java programs and provides instrumentation for making trace data which is captured in a proprietary format. The trace data is then saved to a file or sent over a network to a visualiser for live analysis.

Jinsight supports task-oriented tracing or drive-by analysis, which can trace details of a program task selectively.

STEP [3] is a system designed to provide profiler developers with a standard method for encoding general program trace data of Java programs. STEP uses the JVMPI and trace information is expressed in ASCII text, while visualisations are created with Evolve [28]. There is limited support for storing the program traces which will inhibit being able to create visualisations in the future.

JaVis [17] is a system for the debugging and visualisation of concurrent Java programs. The first part of the system creates a trace of events that occur during execution using JDI and the second part creates UML interaction diagram visualisations based on the trace or for each deadlock.

Alonso and Frakes [1] have developed a model and architecture for visualising reusable components which focuses on providing visualisations that are linked to a repository where components are stored. Queries are submitted to the repository where upon the results are stored in an intermediary XML file in the 3CML format. The XML files are used to create static visualisations of a component at compile time, no dynamic visualisations are available.

All of the systems mentioned in this section predominantly focus on a selection of problems to be solved, use similar debugging technologies to capture event information, and have some variant of a formal specification for the program trace format. They are limited to one programming language for the components, are not platform-independent, and do not focus on program traces being transportable to many visualisation tools.

7 Summary

It is important for developers to understand components before they can reuse them in another program. Software visualisations help to assist developers in understanding reusable components by visualising the internal structure and behaviour of components. However it is a difficult process to generate meaningful visualisations for comprehending.

In this paper we approach the problem by test driving components to store static and dynamic information in a formal, transportable, and filterable intermediary program trace format. We have created two XML program trace format languages. The languages, PAL and CTL describe the static and dynamic information of C++, Java, and C# reusable components and can be used to create visualisations at real time or in the future for a developer to understand.

References

[1] Omar Alonso and William B. Frakes. Visualization of reusable software assets. In *Proceedings of the 6th International Conerence on Software Reuse*, pages 251–265. Springer-Verlag, 2000.

[2] Craig Anslow, Stuart Marshall, Robert Biddle, James Noble, and Kirk Jackson. Xml database support for program trace visualisation. In Neville Churcher and Clare Churcher, editors, *Australasian Symposium on Information Visualisation, (invis.au'04)*, volume 35 of *CRPIT*, pages 25–34, Christchurch, New Zealand, 2004. ACS.

[3] Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren, John Jorgensen, Clark Verbrugge, and Qin Wang. Step: a framework for the efficient encoding of general trace data. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 27–34. ACM Press, 2002.

[4] Bruno Cabral, Paulo Marques, and Luis M. Silva. Il code instrumentation with rail. *.NET Developers Journal*, 2(1):34–35, January 2004.

[5] Stephan Diehl. Revised lectures on software visualization, international seminar, 2002.

[6] Matthew Duignan, Robert Biddle, and Ewan Tempero. Evaluating scalable vector graphics for use in software visualisation. In Tim Pattison and Bruce Thomas, editors, *Australian symposium on Information visualisation, (invis.au'03)*, CRPIT, pages 127–136, Adelaide, Australia, 2003. ACS.

[7] Peter Eades and Kang Zhang, editors. *Software Visualisation*, volume 7 of *Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, River Edge, NJ, USA, November 1996.

[8] Free Software Foundation. Gdb: The gnu project debugger. `http://www.gnu.org/software/gdb`.

[9] IBM and Alphaworks. Jinsight: A visual tool for optimizing and understanding Java programs, 2002. `http://www.research.ibm.com/jinsight`.

[10] Stuart Marshall. Understanding code for reuse. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, 1999.

[11] Stuart Marshall, Robert Biddle, and James Noble. Using software visualisation to enhance online component markets. In Neville Churcher and Clare Churcher, editors, *Australasian Symposium on Information Visualisation, (invis.au'04)*, volume 35-41 of *CRPIT*, page 35, Christchurch, New Zealand, 2004. ACS.

[12] Stuart Marshall, Kirk Jackson, Craig Anslow, and Robert Biddle. Aspects to visualising reusable components. In Tim Pattison and Bruce Thomas, editors, *Australian symposium on Information visualisation, (invis.au'03)*, CRPIT, pages 81–88, Adelaide, Australia, 2003. ACS.

[13] Stuart Marshall, Kirk Jackson, Michael McGavin, Matthew Duignan, Robert Biddle, and Ewan Tempero. Visualising reusable software over the web. In Peter Eades and Tim Pattison, editors, *Australian Symposium on Information Visualisation, (invis.au 2001)*, volume 9 of *CRPIT*, pages 103–111, Sydney, Australia, 2001. ACS.

[14] Mike McGavin. Extracting software reuse information for visualisation tools. Honours Report, School of Mathematical and Computing Sciences, Victoria University of Wellington, October 2001.

[15] Mike McGavin, Tim Wright, and Stuart Marshall. Visualisations of execution traces (vet): An interactive plugin-based visualisation tool. In Wayne Piekarski, editor, *Seventh Australasian User Interface Conference (AUIC2006)*, volume 50 of *CRPIT*, pages 153–160, Hobart, Australia, 2006. ACS.

[16] David Megginson. Simple API for XML (SAX), January 2002. `http://www.saxproject.org`.

[17] Katharina Mehner. Javis: A uml-based visualization and debugging environment for concurrent java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 163–175. Springer-Verlag, 2002.

[18] Sun Microsystems. Java debugger interface api, July 2003. `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdi`.

[19] Sun Microsystems. Java reflection api, July 2003. http://java.sun.com/j2se/1.4.2/docs/guide/reflection.

[20] James Noble. *Abstract Program Visualisation*. PhD thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington, 1996.

[21] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by analysis of running programs. In *Proceedings for Workshop on Software Visualization*, Toronto, Canada, May 2001. International Conference on Software Engineering.

[22] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the exeuction of java programs. In *Revised Lectures on Software Visualization, International Seminar*, volume 2269, pages 151–162. Springer-Verlag, 2002.

[23] Steven P. Reiss. An overview of BLOOM. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–5. ACM Press, 2001.

[24] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12), December 1993.

[25] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *Proceedings for TOOLS Europe 2001*, Zurich, Switzerland, March 2001. Technology of Object-Oriented Languages and Systems (TOOLS) Conference Series.

[26] John T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.

[27] John T. Stasko, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization, Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, USA, 1997.

[28] Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren, and Clark Verbrugge. Evolve: an open extensible software visualization framework. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 37–ff. ACM Press, 2003.

[29] World Wide Web Consortium (W3C). XQuery 1.0: An XML Query Language, November 2002. http://www.w3.org/TR/xquery.

[30] World Wide Web Consortium (W3C). Document Object Model (DOM), 2003. http://www.w3.org/TR/DOM-Level-3-Core.

[31] World Wide Web Consortium (W3C). Scalable Vector Graphics (SVG) 1.1 specification, 2003. http://www.w3.org/TR/SVG11.

[32] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) 1.2, June 2003. http://www.w3.org/TR/SOAP12.

[33] World Wide Web Consortium (W3C). eXtensible Markup Language (XML) 1.0 (third edition), February 2004. http://www.w3.org/TR/REC-xml.

[34] Kang Zhang, editor. *Software Visualization, From Theory to Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[35] Kang Zhang. *Software Visualization: From Theory to Practice*, chapter 11. Kluwer Academic Publishers, 2003.