ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

B Novikov, University of St Petersburg, Russia and J W Schmidt, University of Hamburg, Germany. (Eds)

Advances in Databases and Information Systems, Moscow 1996

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS '96). Moscow, 10-13 September 1996

Extending the Gemstone Smalltalk Interface Debugger to Cope with Active Database Components

Ian S. Thomas and Andrew C. Jones

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Extending the Gemstone Smalltalk Interface Debugger to cope with Active Database Components

Ian S. Thomas and Andrew C. Jones

University of Wales College of Cardiff, Dept. of Computer Science, PO Box 916, Cardiff.

email: {scmist,andrew} @ cs.cf.ac.uk

Abstract

In this paper we introduce GOAD - an active database we have developed on top of GemStone - concentrating upon the implementation of our debugger prototype. We show how we extended the standard debugger provided with the GemStone-Smalltalk Interface to cope with the added complexity which the active parts of our system present. These extensions provide the means to trace the execution of rule code in the same straight-forward manner as standard class code. In addition we show how the GOAD visibility-toggling mechanism can be used by the debugger to provide different representations of an active application which are tailored to the needs of different types of users. We give an example of how, in conjunction with more abstract, graphical tools, we can use our debugger to help deal with errors in the functioning of the rule base. We conclude by evaluating the usefulness of our tool and the suitability of Smalltalk and GemStone as an underlying system for the development of tools such as the one we describe.

1 Introduction

There has been a large amount of work recently in the area of Active Databases [3, 10, 12, 15, 16, 17, 18]. Adding active functionality to a system inevitably involves changing the execution model of that system in some way, and we must provide tools to manage these extensions to ensure that users will be able to use them effectively. The use of wrapper methods or the implanting of event-raising code means that instead of merely executing the method of one's choice, one is now adding extra things to the model such as events and rules, and we must enable the users of the system to ensure that these extra parts of the system work correctly. There is a need, therefore, for users to be able to trace the execution of this active layer to ensure that their rules are acting as intended. However, the extra code that deals with the raising of events and the firing of rules - in our case wrapper methods - can sometimes cause further confusion. Debugging a method now involves tracing the code which implements the event detection/handling mechanism in addition to the code we have implemented for the method. We therefore believe that we should be able to debug an active application in two ways: firstly where we can see this extra code being evaluated and secondly where we cannot. This would help to simplify the debugging of code which has activity attached to it.

This paper therefore describes work in which we extend the standard Smalltalk debugger to enable us to debug our active database system, GOAD, which has been implemented using GemStone. It allows us to:

- trace through the rules which make up the active part of an application;
- use the visibility-toggling mechanism of GOAD to provide 'activity-visible' and 'activity-invisible' debugging sessions to present the system at different levels of detail; and
- examine all message sends that occurred during a high-level trace to help with the discovery of non-terminating behaviour

The rest of the paper is split up as follows: firstly we describe the state of the art and our motivation for the research described in this paper. Secondly we shall give a brief overview of our active database, GOAD, before proceeding to describe the implementation of our debugger prototype. Following this we will give an example to show that abstract,

rule-level, tracing tools are not sufficient on their own for the debugging of active database applications and how a debugger such as ours can be of great benefit in tracking down the actual code that causes the undesired behaviour indicated by these higher-level tools. Finally we will present our conclusions and discuss future work.

Throughout the paper we use a method called `name` : for illustration.

2 State of the Art

The debugging of active database systems is an important area, but thus far there have been few proposed approaches [2, 4, 5, 6, 9, 13, 14]. The work that has been done is divided between relational and object-oriented databases, with work centring around:

- static analysis techniques [2, 4, 6]
- dynamic analysis [5, 6, 9, 13]

The majority of work in the object-oriented context (which GOAD falls into) has been carried out in DEAR [13], the Sentinel rule debugger [9] and VITAL [6]. These trace rules at a high level, illustrating graphically which events and rules are interacting and how. Chakravarthy argues that a rule tracer should be at a high level such as this and that traditional debuggers are not suitable for this task. The debugger that comes with the Smalltalk system, however, is not just used to report errors but is also the traditional way of testing Smalltalk applications. Therefore whilst we also provide a high level trace tool to help us pin-point errors in the rule base, we claim several benefits for the extension of the debugger to allow it to be used with rule code:

- We can set breakpoints in rule code and trace execution at code level. This allows us to see exactly how the code is affecting objects within the system, rather than just giving us an abstract view which simply tells us that the rule has been fired.
- The tracing of the rule code will be seamlessly integrated with the rest of the system allowing us to follow message-sends from rule code to class code and back again. Additionally, we can inspect the values of all of the objects involved at each stage.
- Use of our visibility-toggling mechanism means that we can debug the system at two levels, making for more straight-forward debugging.
- By integrating an extended debugger into a broad toolset which includes high-level, more abstract graphical tracers, we can gain information about the active system at differing levels of granularity.

3 The GemStone-Smalltalk Interface

The GemStone OODBMS [Maie86] is an extension of the object-oriented language Smalltalk [Gold89]. There are certain elements of its architecture that we need to briefly cover here in order to be able to talk about their usage later in the paper.

All objects and classes in GemStone reside within the database, and the database system itself is already 'active' in the sense that we are able to send messages to the objects within the database and have them executed *in situ*. (i.e. There is no need to fetch them to an external application on the client machine in order to execute methods of the objects). Therefore the Smalltalk interface that we are using in conjunction with GemStone can be used as a way of merely sending and receiving messages from objects in the database. This means that we need not ever put any application code outside of the database system itself, with the exception of interface tools (since GemStone has no user interface classes). The interface tools within Smalltalk can be used to provide a framework in which to communicate with the objects within the database, and for presenting them to the user in an appropriate way. As an example, when an error occurs within the database this gets passed on to the Smalltalk interface. The interface then opens a debugger which allows us to inspect and issue commands to objects which are *in the database*. Therefore the application resides and is executed within the database, whilst the interface classes merely provide visualisations of what is happening.

4 GOAD - GemStone Object-Oriented Active DataBase

Our active database system, GOAD [22], is an extension of the commercial OODBMS Gemstone [19], adding the kind of behaviour normally associated with active databases. All classes in our system which need to provide active behaviour inherit from a class named `ActiveObject`, which contains all of the extensions needed to support the use of events and rules. We use the following mechanisms within this class:

Method wrapping Each method that needs to raise events is wrapped. The original method name: is recompiled under the selector `uname:` and the method name: becomes code to raise events and call the original method.

Event dictionaries Each active object type has event dictionaries associated with it that hold references to the events declared on it.

Automatic parameter name generation When a rule is declared on a particular event, the number and names of the parameters that are passed to the rule are automatically generated and put into place.

Visibility-toggling Visibility-toggling [23] aims to simplify the browsing and debugging of active systems. It is described in a later section, since it is relevant to the work within this paper.

5 Implementation

5.1 Breakpoints in Gemstone

In GemStone only code belonging to a class or metaclass is able to have breakpoints set. We cannot set breakpoints within individual instances. If we are to debug rules then we must have facilities for setting breakpoints within rule code to ensure that we can stop the execution of the rule at the required point and trace its execution.

5.1.1 Setting Breakpoints in Classes

There are two methods within Gemstone that support the setting of breakpoints. These are:

1. `setMethodBreak:stepPoint:` and
2. `methodBreakObject:stepPoint:`.

The first method checks that the parameters it has received are correct and that the step point at which the break is to be set actually exists. It then calls the second method to create a *break object* that it can register with the system. Here is the standard GemStone code that creates such a break object, an understanding of which will be necessary to follow the discussion in the next section.

```
methodBreakObject: aSelector stepPoint: anInt

|breakObject|

breakObject := InvariantArray new: 5.
"create an array to store the information"
breakObject at: 1 put: 0.
"0 represents a method break"
breakObject at: 2 put:(methodDict at: aSelector).
"put the compiledMethod in which we wish to set the break into the breakObject"
breakObject at: 3 put: anInt.
"places the stepPoint at which we wish the break to occur into the array"
breakObject at: 4 put: (breakObject at: 2)
```

```

_selector.
"selector of the method that we are placing the breakpoint into"
breakObject at: 5 put: self.
"places the class into the breakObject to allow it to be inspected when the break occurs"
^breakObject

```

This break object is then registered with the system and the next time the method is run a breakpoint will occur. When this happens a debugger will open for the inspection of the object that ran the method and allow viewing of both the contexts that were passed through in the run-up to the break and to trace through the system from this point onwards.

5.1.2 *Setting Breakpoints in Rules*

Rules in GOAD are first class objects, and consist of:

- a rule name;
- an event on which the rule is fired;
- a condition;
- an action;
- a priority for the firing order; and
- a state of activation.

The only parts that concern us here, however, are the condition and action code as this is where we wish to set the breakpoints.

Conditions and actions in GOAD are represented by GemStone Blocks. Blocks in GemStone are pieces of compiled code that do not belong to any class and which can be executed by the sending of the message value. Because these pieces of code are not methods of any class, however, we cannot ordinarily set breakpoints in them. Nevertheless, what they do possess, in common with methods belonging to classes, is a `compiledMethod`. We can use this fact in a revised method which belongs to the class `Rule` and which sets breakpoints in the code blocks belonging to rules. The checking method `setMethodBreak:stepPoint:` now receives a string that is either 'condition' or 'action' and which is sent in place of the selector name. This allows the method to find the correct `compiledMethod` (i.e. either that belonging to the condition or the action) before passing this information on to the method that creates the break object. Similarly we have expanded and altered the signature of the `methodBreakObject:stepPoint:` method to become `methodBreakObject:in:stepPoint:` as it needs to receive and deal with different information from before.

```

methodBreakObject: aMethod in: aString stepPoint: anInt

|breakObject|

breakObject := InvariantArray new: 5.
"Create the array as before"
breakObject at: 1 put: 0.
"Tell the system that this is a method break as technically it is, as it is in a compiledMethod"
breakObject at: 2 put: aMethod.
"Put the compiledMethod that we earlier obtained from the block representing the condition or action into the breakObject"
breakObject at: 3 put: anInt.

```

”State the stepPoint within the compiledMethod where we wish the break to occur”

```
breakObject at: 4 put: aString.
```

”Put the string ‘condition’ or ‘action’ into the breakObject (depending on which it is) so that when the break occurs the context will be represented in the debugger by the string”

```
breakObject at: 5 put: self.
```

”Place the rule that is requesting the breakpoint into the breakObject for inspection later”

```
^breakObject
```

We can now set breakpoints in individual instances of `Rule`. Therefore any single rule that we wish to trace the working of can now be broken into at the required point. Next we needed to refine the debugger to ensure that rules were properly represented and seamlessly integrated with the rest of the system so we can trace every type of object in the debugger in exactly the same manner and move effortlessly between rules and classes.

5.2 Updating the Debugger to Allow the Viewing of Rules

Figure 1: The standard debugger context list

In the event of an error or a method break, a list of contexts is passed to the Smalltalk interface which represents the list of incomplete messages. Ordinarily, the designation `executed code` is used to describe contexts which do not belong to a method of a class (Fig. 1). This behaviour needed refining so that rules were represented differently from ordinary executed code. The problem is that contexts provide a description of themselves to the debugger and we need to provide a different description when the context in question belongs to a rule. We therefore had to create a class which, when we ask for a description, returns a string describing the rule rather than just `executed code`.

We then needed to change the `contextList` method which builds up the list of contexts for display in the debugger to ensure that any contexts that belonged to rule code were described correctly. To do this we used a class named `GSRuleActivation` which accepts a context as an argument as well as a string which describes how we wish the context to be represented. When we send messages to a `GSRuleActivation` they are merely passed on to the context that it is wrapped around, the exception being the message which asks for a description. This method returns the string we have provided in place of `executed code` (i.e. how the context would describe itself). This is illustrated in Fig. 2.

All contexts belonging to rules, therefore, are placed into an instance of this new class before being put onto the list. This ensures that rule code contexts are represented as `ruleCode>aRule` rather than `executed code` (Fig. 3).

We can now, therefore, debug rules. However, we need to address the issue of whether it is really appropriate to present this information to all users at all times.

Figure 2: GSRuleActivation

Figure 3: The refined debugger context list

5.3 Tailoring the Representation

The ability to trace the active layer increases the complexity of system use. For users not involved in the maintenance of the rule base there is not necessarily any need to trace through the wrapping code or the rules that make up the active extensions of GOAD. Tracing through wrapper code and rules greatly increases the time it takes to debug an application. We have therefore implemented a system which can tailor the level of complexity presented to the user. Users tracing and maintaining the rule base can have the active layer visible and debuggable (activity-visible). A user with no current interest in the active layer, however, can be given a representation which hides the extra complexity, allowing them to just debug their own methods with the active layer working invisibly in the background (activity-invisible). This system is visibility-toggling [23] and we will give a brief description of it here before relating it to our discussion on debugging rules.

5.3.1 Visibility-Toggling in GOAD

Visibility-toggling allows the programmer to choose whether we are interested in viewing the active layer. If not then the active layer works in the background and we are never aware that it is there.

ActiveObject subclasses (and their metaclasses) store a variable `visible` which indicates whether a class is activity-visible. Whether a class is activity-visible or activity-invisible influences how some methods belonging to the class react. To display a class the Gemstone Browser uses several methods common to all classes:

- `selectorsIn:` - return a list of the class's selectors

- `sourceCodeAt:` - return the source code for method
- `compileMethod:` - accept changed method text for re-compilation

These methods are overridden in `ActiveObject` to do different things in the event of a class being activity-visible or activity-invisible.

If a class is activity-visible then there is no need to return anything different (we are not hiding anything). As a result each method executes the standard system method that all classes use.

If the class is activity-invisible, however, it returns different information.

Figure 4: An activity-visible debugging session

- `selectorsIn:` returns a list which does not include the selector of any wrapped method. So `uname:` would be removed from the list of selectors before it is returned.
- `sourceCodeAt:` returns the user's method rather than the wrapper code. Therefore if we clicked on `name:` in the browser we would actually see the method code for `uname:` (only with the 'u' taken off the front). This is because we are hiding the fact that there is any activity attached to the selector `name:`.
- `compileMethod:` receives as a parameter a changed method which will have a selector `name:` (since we removed the 'u' before displaying it). The compilation method knows however, that this method is a user one and so will actually compile it to `uname:`.

The debugger makes use of these facts to present different views of the system depending on whether we wish the active layer to be visible or not. When the active layer is visible we get to see the entire set of classes and rules that are being evaluated (Fig. 4).

When we wish to debug the system without the distraction of having to look at the active part, however, debugger presents a view of the system such as that in Fig. 5.

In this instance, exactly the same code has been evaluated as in Fig. 4, but we have hidden the active part of the system to save us from the distraction and time of tracing through it. To enable this kind of usage, however, we had to refine the debugger in several further ways.

5.3.2 Providing Debugging Sessions with Different Levels of Complexity

Allowing the debugger to work with a hidden activity model means refining its behaviour. There are three things that need to be altered:

Figure 5: An activity-invisible debugging session

- what is displayed in the debugger's list of contexts;
- what happens when we step 'into' a message send (i.e. trace the execution of a message within its receiver); and
- what happens after stepping through a user's method.

The context list The list of contexts displayed in the debugger is altered so that all contexts belonging to the active layer are removed and all contexts belonging to wrapped methods (i.e. user methods) are described as `name:` rather than `uname:`. This hides the fact that the method has been wrapped. We do this in a similar way as with rules, using a class that alters the description of the context.

This means that when we have designated a class as activity-visible, we can see all contexts, including both wrapper methods and wrapped methods (as in Fig. 4), but when the class is activity-invisible we appear to have only one method (as in Fig. 5).

Step 'into' method If the class of the method we are stepping into is activity-invisible then we need to hide the fact that the method is wrapped and merely show the user's method (Fig. 6). We do this by:

- stepping into the wrapper method without updating the display;
- automatically stepping through the wrapper method until we get to the call to the wrapped method (i.e. `uname:`); and then
- stepping into the user's method and updating the display.

This means we skip the wrapper code and as far as the user is concerned it does not exist. The user sees the method actually written for `name:` and does not see the wrapper code at all. So when the message `name:` is sent, the user will not be shown the code that is actually stored in `name:` (i.e. the wrapper code) but the user's method (`uname:`).

Step 'over' method If we get to the end of a method which is wrapped we want to automatically step through the end of the wrapper code that called it and return to the method that called the wrapper code before updating the displays (Fig. 7). This again hides the fact that the method was wrapped and allows us to only show the methods that represent the non-active parts of the system. It works as follows:

- Step over the message without updating the display.

Figure 6: Stepping through wrapper code before and after execution

- Check whether the context that is now top of the stack is the same as before we stepped. If so then update views and highlight the new step point.
- If the context has changed then automatically step through the wrapper code and return to the method that called it before updating the displays.

Figure 7: Stepping through a method

These techniques allow us to present both activity-visible and activity-invisible views of the system. There is a further problem with rules, however. When we set a breakpoint in a rule and this breakpoint then occurs when a user is viewing the system in an activity-invisible mode, the user will still be informed that a breakpoint has occurred. This means that our visibility-toggling mechanism no longer works as it should, since we are only giving the user a representation whereby the active layer is partially hidden. We therefore need a way to ignore breakpoints that occur in code that we have designated as invisible.

5.4 Handling Breaks in 'Invisible' Code

In order to trap the breaks generated by Gemstone we needed to use the Smalltalk exception handling mechanism.

5.4.1 Smalltalk Exceptions

Smalltalk Exceptions can recognise different errors and deal with them if we have defined a specific course of action for the error. By default, method breaks are dealt with by the standard emergency handler. This notifies the user of

Figure 8: Trapping breaks with an exception handler

the break and allows the user to proceed (continue execution) or debug (trace the code). We have defined a Static Exception that recognises method-breaks and deals with them as follows:

- If the code the breakpoint was encountered in is not designated as invisible then the normal system procedures for handling breakpoints take control.
- If the code is designated as invisible, however, we take control away from the normal system procedure and resume execution of the application without the user ever being informed that the break has been encountered.

(This model is illustrated in Fig. 8).

We therefore need some way of knowing when we are executing invisible code and disabling breakpoints.

5.4.2 BreakSavingObject Class

BreakSavingObject keeps track of whether breakpoints are enabled or not. It counts the number of times that breaks have been disabled and enabled, only answering that breaks are enabled when the level of disables is zero. We needed to do this since it is possible to call another wrapped method from within a wrapped method.

We then use BreakSavingObject within our Static Exception to continue execution if breakpoints are disabled and we are executing code that is designated as invisible. If we are executing visible code, however, the normal system procedures take over and a debugger is opened for inspection of the breakpoint. This means that the active part of our system can now be wholly visible or invisible, since we are ensuring that breakpoints are ignored when they occur in invisible code.

6 Our Debugger in Context

Our debugger was implemented as just one part of a broader set of tools for the development of active database applications. At the present time these tools include:

- a textual rule browser/designer;
- a rule logging mechanism;
- a graphical static analysis tool to aid in the discovery of non-terminating rule behaviour;
- a graphical active schema design/browser tool;
- a graphical run-time tracing tool for the visualisation of event and rule execution;
- our extended debugger; and
- an extended Class Hierarchy Browser to help in the design of active classes.

An extended discussion of this toolset is outside the scope of this paper. (A description of our run-time tracer tool can be found in [24]). However, since we wish to demonstrate the usefulness of our debugger as part of this toolset, we will need to demonstrate the use of the static analyser and run-time tracer.

6.1 Our Example Application

The application we use here for illustration is a simplified version of the Starburst power station application discussed in [8]. Besides simplification, differences are also introduced as a result of the conversion of the schema from a relational to an object-oriented paradigm.

6.2 Debugging the Application

6.2.1 Cyclic Rule Calling

One of the biggest problems of active database development is the potential for creating non-terminating behaviour [7, 13, 11, 20]. This problem becomes ever more difficult to detect as the size of a rule base increases, since it becomes increasingly difficult to foresee the possible interactions amongst rules [21].

The application involves connecting power cables from power plants to the users of the power. We have set up some of the rules in our application so that they form infinite cycles; this will enable us to demonstrate how our tools could be used to find such a problem. The rules in question are those where a `PowerPlant` triggers an addition or removal of a cable from a `PowerUser` or vice-versa. The non-termination occurs because after we have added a cable to a `PowerPlant` we trigger an addition of the cable in the `PowerUser`. After the `PowerUser` has added the cable it then triggers an addition in the `PowerPlant`. The same thing happens when we remove a cable. This results in an infinite cycle, with each object continually triggering the other. We will show how we can use our tools to find problems such as this.

6.2.2 Static Analysis of a Rule

We wish to test whether or not the rule `powerUserTriggeredUpdate` (the rule which triggers an addition of a cable within a power user) will terminate. The first thing we can do to test this is to use our static analyser to examine the message-sends that occur within the rule and see whether any of them will trigger more active behaviour. If another rule is triggered by the first then we check this rule to see whether or not that rule triggers any more behaviour. This continues until either a rule does not trigger any more active behaviour or we find a call to the original rule. In the second case we have found a potential cycle which needs to be investigated further by the designer.

Figure 9: Results of static analysis

In the resulting browser which displays the possible execution tree of the rule (Fig. 9) the events `PowerPlant>before >addCable:to:` and `PowerPlant<after<addCable:to:` are highlighted to show that they have been triggered more than once by the same rule in the analysis. (The far end of the tree ends with the same two events highlighted after they have been found for the second time). This tells us that the rule `powerUserTriggeredUpdate` *may* not terminate and thus we *may* have cyclic calling of rules. To ascertain whether or not this is really the case in practice we need to trace the execution of the rule as it is running. This allows us to see whether the possible outcome discovered by the static analyser ever actually occurs or not.

6.2.3 Run Time Tracing

The static analysis phase has indicated that the rule `powerUserTriggeredUpdate` may not terminate. We therefore need to trace the execution of the rule at run-time to see how the rule is interacting with other rules in the system.

Fig. 10 shows an open tracer in which we have followed the execution of the rule `powerUserTriggeredUpdate`. It shows the end part of the trace tree, highlighting the event `Rule>before>condition` to show that it has been executed twice by the same rule. This means that we have now discovered that the possible scenario outlined by the static analyser (i.e. that the rule was non-terminating) was in fact a viable one.

Figure 10: Results of dynamic analysis

6.2.4 Using the Extended Debugger

Besides the obvious benefits of allowing the debugging of rule code, our extended debugger also helps us, in conjunction with our other tools, to find more subtle problems. In this case, our high-level graphical tracing tools have drawn our attention to the fact that there is a problem somewhere within the active part of our application which is causing non-terminating behaviour. They do not, however, tell us where this problem is occurring. By their nature these tools are abstracted away from the code level in order to present a more natural and easily understood view of the interactions between rules and events. The actual source of the non-terminating behaviour, however, lies within the code that the rules are executing. We therefore need some way of examining the flow of message-sends that have occurred during the execution that the graphical tracer has been following. Our debugger can provide such a view, displaying all of the contexts that have been passed through as the code was executed. Furthermore, each new context

in the list will have been triggered by a particular message-send in the previous one, and the debugger automatically highlights these messages in each of the contexts in the list. This means that we can follow the exact flow of messages by following the stack of contexts from bottom to top, allowing us to see which message-sends are causing the non-terminating behaviour. Fig. 11 shows the message that initiated the non-terminating behaviour in our example, namely the message send `addCable:` in the action code of rule `powerUserTriggeredUpdate`. As a final aid, we are also able to inspect any of the objects that were involved at each stage of the execution to examine their state and how this may have contributed to the problem (Fig. 12).

Figure 11: Examining the context list

In this example we merely need to expand the condition of the rule to prevent it being fired more than once. This can be done by checking whether the `PowerUser` already has the cable that we are planning to add. If so then there is no need to trigger the update.

Figure 12: Inspecting a variable of the rule `powerUserTriggeredUpdate`

6.3 Conclusions

We have shown how to extend the Gemstone Smalltalk interface debugger in two ways:

1. To enable the seamless tracing of rule code with other parts of the system
2. To use visibility-toggling to debug the system at different levels of detail, ensuring that breakpoints in rule code do not compromise the visibility mechanism

In isolation the debugger has proved to be a useful tool for:

- trapping run-time errors within incorrect rule code and allowing the designer the opportunity to ascertain exactly what went wrong;
- setting breakpoints within rule code, allowing us to trace rule executions at code level. This has enabled us to see - at a finer granularity than is possible in our higher-level tracer - exactly what effect the rule is having upon the system; and
- debugging active class code without the added complication of wrapper methods and events/rules when required.

However, the debugger becomes much more useful when integrated into our set of active database design/maintenance tools, since there it is able to supplement the information provided by our higher-level tracing tools. As well as the benefits cited above we can now also use the debugger, in conjunction with our other tools, to aid in the discovery of non-terminating behaviour. We can first pin-point errors in the working of the rule base by using an abstracted, graphical tool before switching the granularity to code level in order to find the train of message-sends that cause the undesired behaviour. This means that we are providing a quick way of moving between an abstract view of the problem and the behaviour that causes it.

The use of GemStone and the Smalltalk interface as the underlying platform for our work has provided us with several benefits:

1. The exception handling mechanism of Smalltalk has greatly eased the problem of trapping the specific errors that we defined and dealing with them in an appropriate manner.
2. The tight coupling between Smalltalk and GemStone meant that all concerns about communicating with objects in the database were taken care of for us
3. The fact that both the interface language (Smalltalk-80) and the database language (SmalltalkDB) were mostly the same greatly reduced the problems of 'impedance mismatch'.
4. The extensible nature of Smalltalk, as well as the availability of the code, meant that we were able to both understand how the original tools worked and to extend them in the way in which we wished. This meant that we were not faced with the task of re-writing a new debugger which possessed the functionality that we wished to support. We were therefore able to re-use all of the good features of the debugger, greatly easing our job and allowing us to retain a standard tool which current Smalltalk users are at ease with. It also meant that we were able to use the debugger to seamlessly move between rule code (which it was extended to handle) and class code (which it originally handled). Any new debugger that we had created for rule code alone may well not have had this important feature without a substantial implementation effort.

Therefore, GemStone and Smalltalk proved to be well-suited to the development of the tools that we wished to provide. The only real problem that we have come across is the speed at which the database executes. At present this is a problem which we are hoping to address at a later date through optimization.

6.4 Future Work

At present we have an integrated toolset which allows us to create, browse, debug and maintain active applications. Besides the problem of speed which we touched upon above there is one other outstanding problem with our interface tools. At present our static analyser is not very sophisticated and can only find simple cases of non-termination. This means that it is not very useful in some cases. We therefore wish to find a way of implementing a more thorough tool. The problem is that because of the untyped nature of Smalltalk we never know which class is going to be the receiver of a particular message. This means that it is difficult to ascertain whether a particular message-send will trigger any more active behaviour. We wish, therefore, to investigate the possibility of developing a type inference system such as that in [1]. This system was designed with the aim of finding all objects that are reachable within an execution of a given application. The aim of this is so that all of these objects can be removed from a larger environment and placed into a new one, thus reducing the size of the application. This 'shrink-wrapping' of applications is a similar problem to the one which we face, since we are also interested in knowing all possible objects that could be invoked during an execution. This would allow us to ascertain whether there are any loops in such a possible execution graph, thereby indicating that we may have some non-terminating behaviour.

As for other aspects of our active database, we are still looking at ways of extending our system with other kinds of events; most importantly composite and temporal event types.

References

- [1] O. Agesen, D. Ungar. Sifting Out the Gold: Delivering Compact Applications From an Exploratory Programming Environment *Proc. OOPSLA '94*, pages 355-369, 1994.
- [2] A. Aiken, J. Hellerstein, J. Widom. Static Analysis Techniques for Predicting the behavior of Active Database Rules *ACM Transactions on Database Systems*, 20(1):3-41, March 1995.
- [3] E. Anwar, L. Maugis, S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases In *Proc. ACM SIGMOD*, pages 99-108, 1993.
- [4] E. Baralis, S. Ceri, S. Paraboschi. Improved Rule Analysis by Means of Triggering and Activation Graphs. *Proc. 2nd Intl. Workshop on Rules in Database Systems*, pages 165-181, 1995.
- [5] H. Behrends. Simulation-based Debugging of Active Databases *Tech. report TR-IS-AI5-94-02, Fachbereich Informatik Universitat Oldenburg*, 1994.
- [6] E. Benazet, H. Guehl, M. Bouzeghoub. VITAL: a Visual Tool for Analysis of Rules Behaviour in Active Databases. *Proc. 2nd Intl. Workshop on Rules in Database Svstems*, pages 182-196, 1995.
- [7] A. P. Buchmann. Active Object Systems In *Advances in Object-Oriented Database Systems*, pages 201-224, Springer Verlag, 1994.
- [8] S. Ceri, J. Widom Deriving Production Rules for Constraint Maintenance. *VLDB 16*, pages 566-577, 1990.
- [9] S. Chakravarthy, Z. Tamizuddin, J. Zhou. A Visualisation and Explanation Tool for Debugging ECA Rules in Active Databases In *Proc. 2nd Intl. Workshop on Rules in Database Systems*, pages 197-209, 1995.
- [10] U. Dayal, U. et al. The HiPAC Proiect: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1):51-70, 1988.
- [11] U. Dayal, E.N. Hanson, J. Widom. Active Database Systems In *Modern Database Systems: The Object Model, Interoperability, and Beyond*, Addison-Wesley, 1994.
- [12] O. Diaz, N. Paton, P., Gray. Rule Management in Object-Oriented Databases: A Uniform Approach *VLDB 17*, pages 317-326, 1991.

- [13] O. Diaz, A. Jaime, N. Paton. DEAR: a Debugger for Active Rules in an Object-Oriented Context. *Proc. 1st Intl. Workshop on Rules in Database Systems*, pages 180-193, 1993.
- [14] T. Fors. Visualization of Rule Behaviour in Active Databases *Tech report HS-IDA-TR-94-009, University of Skovde Dept. of Computer Science*, 1994.
- [15] S. Gatzju, K.R. Dittrich. SAMOS: an Active Object-Oriented Database System In *IEEE Quarterly Builetin on Data Engineering, Special Issue on Active Database Research*, pages 1-4, 1992.
- [16] N.H. Gehani, H.V. Jagadish. ODE as an Active Database: Constraints and Triggers. *VLDB 17*, pages, 327-336, 1991.
- [17] G. Kappel et al. TriGS: Making a passive object-oriented database system active *JOOP*, pages 40-51, July-August, 1994
- [18] A. Kotz-Dittrich. Adding Active Functionality to an Object-Oriented Database System - a Layered Approach In *Proc. GI Conf. Datenbanksysteme in Buro, Technik und Wissenschaft*, pages, 54-73, 1993.
- [19] D. Maier, J. Stein. Development of an Object-Oriented DBMS. *Proc. OOPSLA 86*, pages 472-481, 1986.
- [20] N.W. Paton, O. Diaz. Active Database Systems. *Submitted for publication*, 1994.
- [21] M. Stonebraker. The Integration of Rule Systems and Database Systems. *IEEE Transactions on Knowledge and Data Eng.*, 4(5), 1992.
- [22] I.S. Thomas, A.C. Jones Design and Implementation of an Active Object-Oriented Database Supporting Construction of Database Tools. *Proc. 2nd Intl Workshop on Rules in Database Systems*, pages 147-161, 1995.
- [23] I.S. Thomas, A.C. Jones Visibility Issues in an Active OODBMS. *Proc. 2nd Intl Conf. on Object-Oriented Information Systems*, pages 150-153, 1995.
- [24] I.S. Thomas, A.C. Jones OLAF: The GOAD Active Database Event/Rule Tracer. *To appear in DEXA 96*, Zurich, September, 1996.