MultiView-Merlin: An Experiment in Tool Integration

Chris Marlin† Burkhard Peuschel‡ Michael McCarthy† Jennifer Harvey†*

† Discipline of Computer Science, Flinders University of South Australia, Adelaide, S.A. 5001, Australia;
(marlin | mccarthy | jenny)@cs.flinders.edu.au

‡ STZ – Gesellschaft für Software-Technologie mbH, Helenenbergweg 19, 4600 Dortmund 50, Germany;
peuschel@stzdo.de

* School of Computer and Information Science, University of South Australia, The Levels, S.A. 5095;
j.harvey@unisa.edu.au

Abstract

The experiment described in this paper involved the integration of a process-centred software development environment (Merlin) and a multiple-view integrated software development environment (MultiView). These two tools were developed quite separately from each other, with no expectation that they would ever be integrated into a single integrated software engineering environment.

The paper first briefly presents the separate environments and then describes the technique used to integrate them. This technique centres on the development of an adaptor process to mediate between the environments. It was first necessary to identify the point at which to connect the two environments, and then to design and implement an appropriate process to pass commands between them.

This work has resulted in enhancements to both of the separate tools and has created a combined environment which exploits the advantages of both of the original environments.

1 Introduction

An experiment in tool integration is described, involving the integration of a *process-centred software development environment* (PSDE) and an *integrated software development environment* (ISDE). Both of these environments were constructed on the assumption that they would be in complete control of their operating environment; neither was constructed with the idea of being integrated with any other tool.

Tool integration is concerned with the extent to which tools coalesce, that is, the extent to which and ease with which tools cooperate to achieve common goals. Included in this is the ease with which data is managed and shared, the ease with which the interaction between tools (including invocation of tools) is controlled, and the appearance of the result of the integration of two or more tools to the user.

Integration presents differently depending on the viewpoint. A user of an environment, for example, expects a well integrated environment to operate as a coherent whole, expecting the integration to result in a seamless meshing of the component tools which eases the task of software development and does not involve cognitive context switching as different tools are invoked. Indeed, automatic invocation of tools should be transparent to the user. However, to the environment builder, a well integrated environment represents:

- an environment which is easily extensible,
- an environment in which tools can be easily removed or replaced by others with similar functionality, and
- an environment which allows additional tools (new or third party) to be easily integrated.

Several themes of integration, initially identified by Wasserman [Was89], are discussed in the literature. Data, Control and Process Integration are of particular interest to this project.

Data Integration is instrumental in providing tool interoperability and relates to the sharing of data amongst tools and the management of the relationships between data objects produced by the tools. This usually implies a central data store, often with communication between tools occurring via the database in a tightly-coupled architecture, or via a message server in the case of more loosely coupled integration. Both Thomas [TN92] and Welsh [WY91] recognise the existence of persistent (primary) and non-persistent (secondary) data and the need for a tool to have access to both at the appropriate times.

Control Integration is defined as the ability to notify interested tools of significant events and the ability of a tool to react to such events, as well as the ability to automatically invoke tools in order to achieve a goal. Welsh notes that an environment may have control tools as well as function tools. Control tools organise and monitor the activation of other tools. Note the implied overlap with Process Integration (see below), which leads Thomas to state that Control Integration allows the combination of environment functions according to the preferences of a project, as directed by the underlying process.

0-8186-4460-5/93 $03.00 © 1993 IEEE

Process Integration is the support for a well-defined software engineering process. The implication is that the tools cooperate and interact effectively to support the process. Both Wasserman and Thomas believe that Process Integration is an integral part of an integrated environment, whereas Welsh argues that Process Integration has no specific implications for tool integration mechanisms and should be able to be implemented using facilities already provided by Data Integration and Control Integration. To some extent, this view is correct (and leads to the overlap between Control Integration and Process Integration); however, the process model employed by the environment will have obvious impact on aspects of Data Integration (e.g., the attributes of stored objects and the definition of and ability to manipulate the relationships between objects), and on Control Integration (for example, tool activation mechanisms, tool activation sequences, and notification of events).

Current research approaches to building *integrated software engineering environments* (ISEEs) focus either on:

(1) providing a set of highly integrated tools supporting a particular life cycle phase and hence the production of one type of document (such as a program, a structured-analysis diagram or an entity-relationship model), or

(2) loosely integrating various separate tools to provide support for a range of life-cycle phases and the corresponding development of a range of documents.

The MultiView ISDE is an example of the first approach: in its present form, it provides language-sensitive editor support for a programming language (Modula-2 is one of the languages supported, for example) via a variety of graphical and textual views on software components written in a language supported. The Merlin system is a loosely coupled PSDE and is hence an example of the second kind of ISEE.

Because Merlin and MultiView are complementary in their functionality, a straightforward way of investigating integration mechanisms and mutually extending their functionality presented itself with the chance to integrate the two environments. This opportunity arose during the course of wider cooperation between the corresponding research groups on the architecture of ISEEs, a discussion of which is beyond the scope of this paper.

The next section describes each of the separate tools (MultiView and Merlin) in isolation, to give some idea of their appearance and their separate implementations. The following section then describes the experiment of connecting these two tools into a single integrated environment. The final section presents some conclusions and discusses some possible future work.

2 The separate tools: MultiView and Merlin

2.1 MultiView

The MultiView project [AHM88, Mar90] at Flinders University is investigating the construction of multiple view ISDEs which are implemented in a distributed fashion.

Most software development environments provide access to the software components under development via only one editable representation. Typically, this is text. With the advent of cheaper workstations with high resolution displays, some experimental systems have been developed which make use of graphical depictions of programs. The principal motivation for the development of the MultiView environment has been the observation that software developers tend to make use of various representations during software development. Thus far, the work on the MultiView prototype environment has concentrated on the coding phase of the software life-cycle.

From a user's point of view, the MultiView environment provides multiple concurrent views of the software system under construction and is thus similar at the user level to the PECAN system [Rei84, Rei85]. The MultiView approach allows support for many of the representations employed by software developers, which range from textual descriptions (such as program listings) to various diagrammatic representations (flowcharts, Nassi-Schneiderman diagrams, and so on). The present MultiView prototype supports three kinds of view as a demonstration of the feasibility of the approach.

Figure 1 depicts a session with the current MultiView prototype. In the top left-hand corner of the figure is a ControlView window, which allows the creation of view instances and the loading of program components in the MultiView database. The remaining three windows in Figure 1 show instances of the view types mentioned above:

- the window in the top right-hand corner shows a flow-chart view (FlowView),
- the window in the bottom left-hand corner shows a textual view (TextView), and
- the window in the bottom right-hand corner shows an abstract syntax tree view (TreeView).

All of these view instances are being used to view the same program component (a Modula-2 unit called "test3.mod"); a user can use any of the view instances to edit the component. It would have been possible to have other view instances being used to view different program components at the same time. The interested reader is referred to [Mar90] for a more detailed discussion of the operation of an earlier version of the MultiView environment.

The implementation of the MultiView environment is distributed and is designed to exploit the kind of coarse-grained parallelism to be found in multiprocessor workstations. This style of implementation is motivated by the

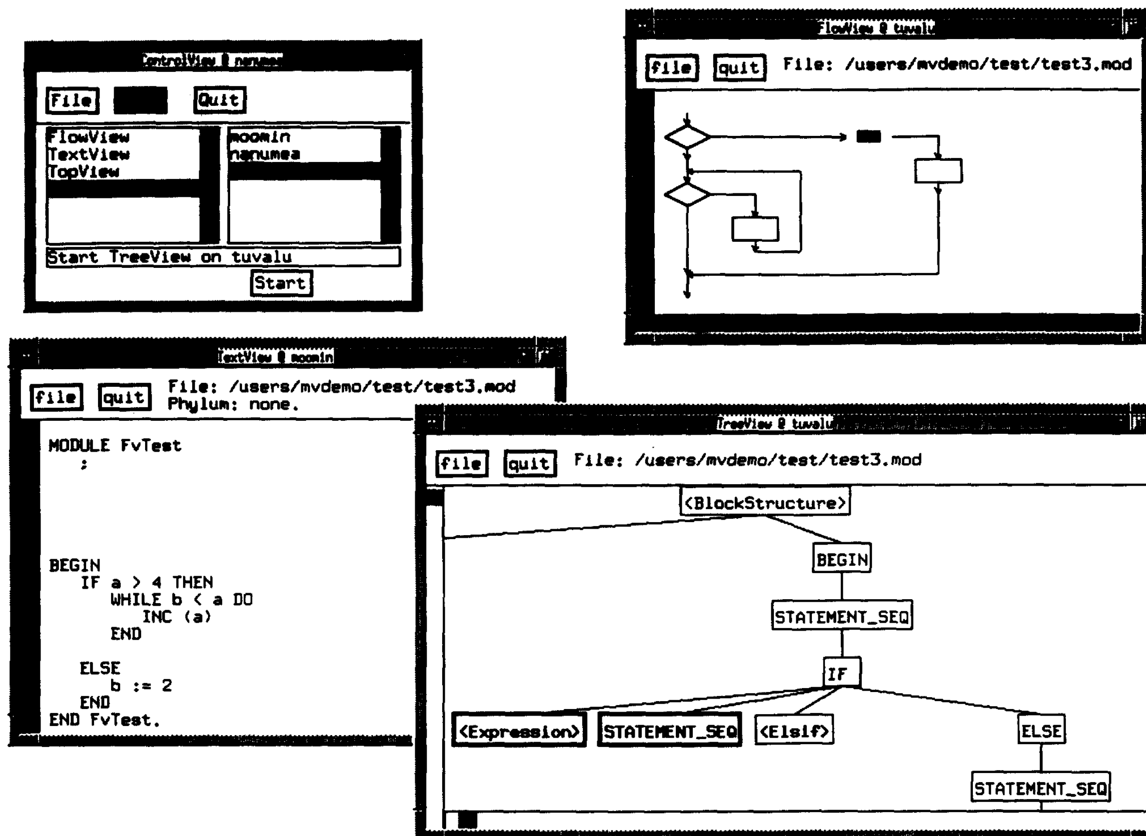

Figure 1: A session with the current MultiView prototype.

desire to improve the performance of sophisticated software development environments, particularly with regard to tasks such as incremental code generation. Experience with other kinds of computer-aided engineering environments (say, those for assisting with circuit design) has shown that, as the environments become more sophisticated, the most expensive resource – the engineer – is forced to remain idle for longer periods of time, waiting for response from the environment.

The software architecture of the MultiView implementation is illustrated in Figure 2: the implementation consists of a collection of concurrently executing processes communicating via message-passing, which has been implemented in terms of UNIX sockets. The exploitation of parallelism that occurs within the MultiView implementation depends on the concurrent execution of these processes.

At the heart of the implementation is the database, controlled by the *database process* shown at the bottom of Figure 2. This database holds abstract syntax trees for the collection of compilation units currently being operated upon by the user. The database process receives notification of changes to the compilation units held within the database and broadcasts information about modifications when necessary.

Each view instance is managed by a *view process*; if the user is employing a view instance to perform editing on a compilation unit, the corresponding view process receives and interprets the input from the user. Every view process holds a view-specific copy of a single compilation unit; this is an abstract syntax tree which is decorated with information relating to the particular kind of view. When the user input has been interpreted, the corresponding modifications are made to the local abstract syntax tree and the visible representation updated. Once this has been done, an appropriate description of the changes desired are sent to the database process, which then broadcasts notification to any other view processes holding a copy of the relevant abstract syntax tree; these view processes then update their abstract syntax trees and visible representations.

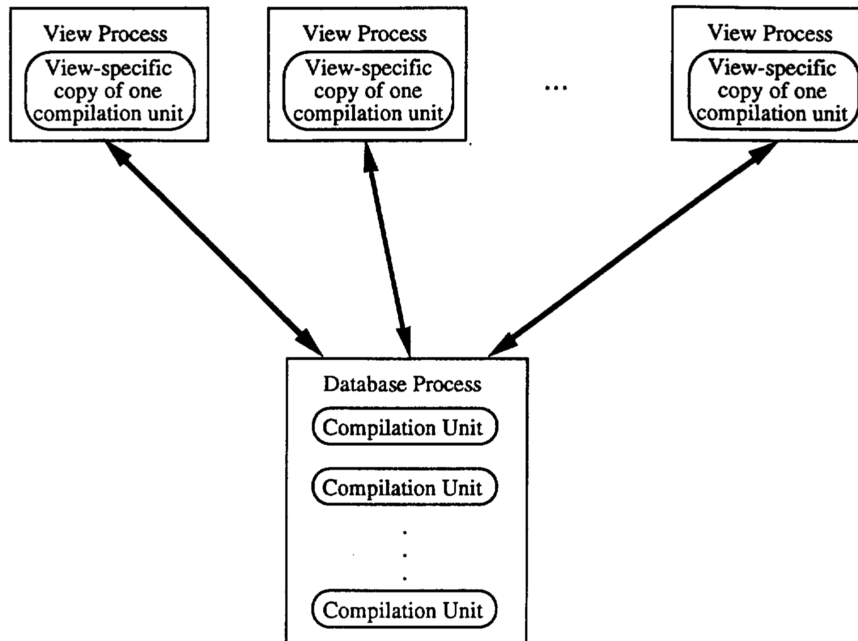

Figure 2: The software architecture for the MultiView implementation.

The interface between view processes and the database process consists of a protocol which is common to all kinds of view process. This protocol is independent of the kind of view concerned because it deals entirely in terms of the abstract syntax tree operations.

The implementation structure depicted in Figure 2 allows view instances to be updated in parallel, which contrasts with the round-robin scheduled updating of views which occurs in the PECAN implementation. The current MultiView prototype is the third implementation of the system and current work is focussed on adding incremental semantic analysis and parallel incremental code generation, in preparation for the development of run-time views.

2.2 Merlin

Merlin is a prototype PSDE, developed by the Merlin project at the University of Dortmund. This work has been carried out in cooperation with Gesellschaft für Software-Technologie mbH (STZ), a Dortmund-based software house. The Merlin prototype uses a rule-based technique to describe a software process model [PS92, PSW92]. The emphasis of the Merlin work is on executing (or enacting) such process definitions.

Through this enactment, the environment is able to provide information concerning the current and previous states of a software project, to inform users about activities to be performed and to provide information concerning time and other constraints for specific activities. One major achievement of such an environment is the computer supported integration of development activities with management activities. Project managers, for example, are able to retrieve on-line information about the current project status and developers are immediately informed about any additional activities to be undertaken and of constraints applying to existing or new activities.

A process definition to be enacted by Merlin consists of the following entities:

- *Activities:* a collection of tasks which achieve some goal related to the production of a software product (e.g., specifying, editing, compiling or testing a module);
- *Roles:* groups of activities which are logically highly related and which represent a subset or view of the software process (e.g., activities representing a project manager, a technical leader or a programmer);
- *Software objects:* objects of any granularity that are produced during the software development process (e.g., modules, documentation and test plans);
- *Resources:* people who participate in the production of software and technical resources such as tools supporting the software development activities (e.g., editors and debuggers).

A software object has a related set of activities which can manipulate the object and a related set of tools which support these activities. For example, a module to be pro-

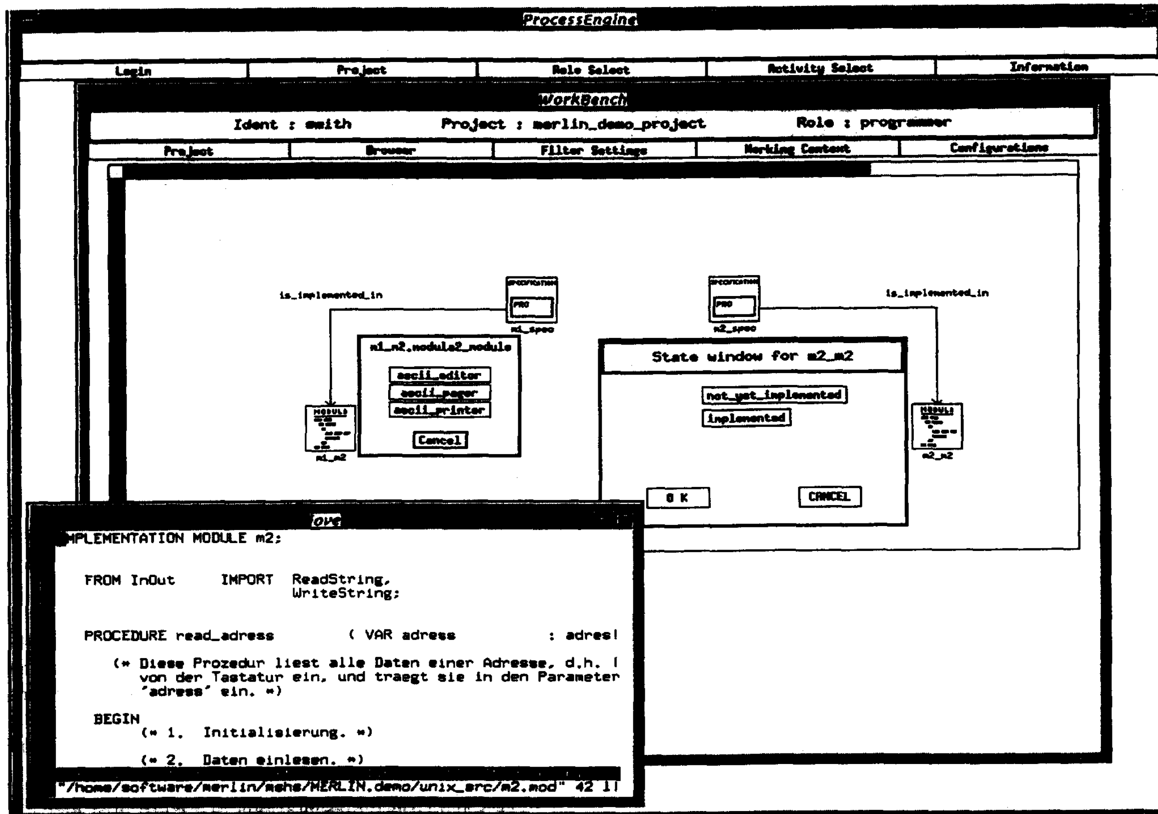

Figure 3: A session with the Merlin process-centred software development environment.

grammed can be edited by some editor and a module to be tested can be executed under the control of a debugger.

Furthermore, users are associated with one or more roles, each role presenting a view of the software process. As a result of this view mechanism, users are assisted by the Merlin environment; all relevant information (and no more) associated with their current role is displayed in a working context. The display includes objects currently being manipulated, their dependencies to other software objects and the activities which can be used to manipulate each object. This approach varies from that taken by many other PSDEs; the user is presented with all information needed to perform a task, the user can be confident that there is no more relevant information available, and that no unrelated information is displayed (avoiding information overload). This is in contrast to operating systems, for example, where the user has to know which objects exist, which activities can manipulate these objects and (often) where to locate the appropriate tools.

A Merlin user sees a *working context* displayed on the screen in a hypertext-like manner; in this working context, software objects are represented as boxes and labeled arcs between the boxes describe the relationships between software objects. For example, the working context in Figure 3 (the window labelled "WorkBench") shows four software objects and two arcs. The boxes have attached menus which detail the manipulations or activities which can be performed on the software objects. In Figure 3, one of the software objects is a Modula-2 module labelled "m1_m2"; this module has associated with it a menu of three activities which can be performed on it: "ascii_editor", "ascii_pager" and "ascii_printer". Selecting a menu item will cause the invocation of the appropriate tool or tools to perform the activity. The window in the lower left-hand corner of Figure 3 shows the result of selecting the "ascii_editor" menu item associated with the other Modula-2 module shown in Figure 3: the implementation module "m2" (denoted by "m2_m2" in the working context). Such tool invocation and any underlying cooperation between multiple tools to complete the selected activity is transparent to the user.

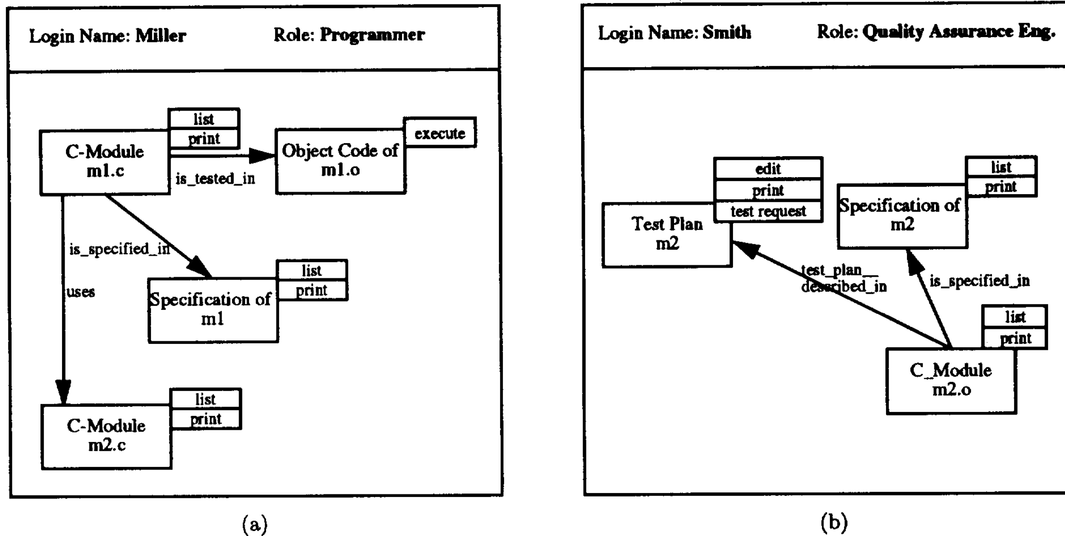

Figure 4: The working contexts of a programmer and a quality assurance engineer.

The line at the top of the WorkBench window displays the current user's name, the project concerned and the corresponding current role (e.g., designer or programmer). The line below this is a menu bar which provides facilities for customising the working context display. It allows, for instance, selection of preferred tools for specific activities (such as a preferred editor). It is also possible to filter the display to obtain an overview of the working context information; for example, the user can hide all software objects of type *specification* or hide software objects without attached relationships. In addition, browsing through the network of software objects is facilitated by the ability to centre the displayed collection of software objects on a selected software object.

The software objects displayed in a user's working context are determined dynamically by the current system state. System state is a snapshot of the individual software object states; as the states of software objects alter, so the system state and hence the contents of various working contexts are altered. Software object status information is requested from the user whenever an activity terminates. For example, when the editing of module "m2" in Figure 3 is complete, a state selection window appears, as shown in the middle right of the figure. This window requires the user to specify the current status of the software object concerned; the status information displayed is determined by the underlying process model.

As a brief illustration of how Merlin supports cooperative software development, consider a situation involving two developers: Miller and Smith, who are performing the roles of programmer and quality assurance engineer, respectively. Specifically, Miller has responsibility for coding and reviewing modules, and Smith is responsible for testing them. The software process involved is defined in the following terms: the quality assurance engineer performs extensive testing of a module based on a predefined quality plan, but only after a module has been coded, reviewed and briefly tested by the programmer.

The schematic representations in Figure 4(a) and (b) show the working contexts for Miller and Smith, respectively, at some point in the software development process. (These schematic representations will now be used in place of complete screen snapshots, to save space and to make the essential aspects clearer.) At the point depicted in Figure 4, Miller is working on executing module "m1", while Smith works on creating a test plan for module "m2".

When Miller finishes coding and testing module "m1" (and so has altered the software object state), the working contexts are refreshed and Miller's working context no longer contains the module "m1" or its associated software objects, as shown in Figure 5(a); in fact, Miller has no further activities to perform in the role of programmer at this point. Since Smith is responsible for testing "m1", this module and its associated software objects now appear in Smith's working context, as illustrated in Figure 5(b).

If module "m1" does not pass the quality tests being carried out by Smith in Figure 5(b), as would be indicated by the appropriate software object status information, Miller's original working context – that in Figure 4(a) – will be regenerated.

A more detailed example and discussion of Merlin's support for cooperative work can be found in [PSW92]. The example discussed in this paper is based on the ISPW6/7

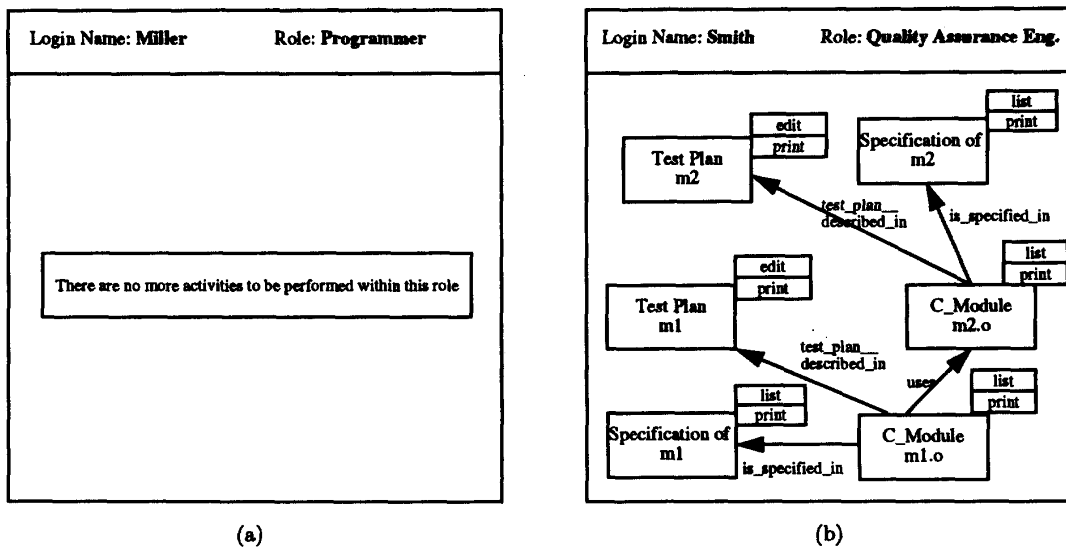

Figure 5: The refreshed working contexts of a programmer and a quality assurance engineer.

scenario, defined in [KFFK91].

As previously mentioned, Merlin is a loosely coupled integrated environment which currently uses Unix tools (such as *vi* and *make*) exclusively. It provides high level support for process integration; a process model is described using rule-based techniques and mechanisms are provided for the enactment of such a description. Data integration is achieved through the Unix file system with relationships between software objects, and role, activity and file access information stored as a dynamic, persistent graph structure. Control integration is bound into each *WorkBench*. The WorkBench is a structure which includes the user's working context and mechanisms for the automatic and transparent invocation of tools (e.g, the invocation of a compiler at the point that a code component becomes complete) and notification of significant events (such as the alteration of software object status).

The implementation architecture for Merlin is shown in Figure 6. The Merlin environment supports multiple users by providing a *working context* to each user. Depending on changes in the states of software objects made by a user, this working context and the working contexts of the other users are refreshed. Each user is supported by one ProcessEngine and one working context. The *ProcessEngine* is responsible for calculating the working context by interpreting the facts and rules which define the software process. The facts and rules are stored in the *process database*. The *WorkBench* realizes the user interface (i.e., display of menus and the working context. Therefore, the WorkBench Control needs to have access to the underlying window management system (X Windows, in this case). Furthermore the WorkBench is responsible for invoking tools; this is realized by the ToolCaller, which implements envelopes to invoke the tools and to receive the return codes (e.g., if a compiler has been called). Thus, the ToolCaller needs access to the software object store which is part of the process database. Communication between the ProcessEngine and the WorkBench is realized by the Communication component, which is based on UNIX sockets.

Multiple user support is achieved by sharing the process database: all process engines involved share the facts and rules, and the WorkBench instances share the software objects. If a process engine has changed some process relevant information, the other process machines are notified that they have to update the working contexts for their users.

3 Approach to tool integration

3.1 Introduction

As mentioned in Section 1 and as illustrated in Section 2, Merlin and MultiView provide completely different functionality. A useful integrated environment could be obtained if the sophisticated editing facilities provided by MultiView were to be used by Merlin at the point that a software object consisting of programming language source code is about to be edited by a Merlin user. Thus, the goal of our integration experiment was to attempt to have the two tools cooperate in this fashion.

This style of integration involves more than simply *calling* MultiView from the Merlin environment, since MultiView is a self-contained environment with its own data, control and process integration mechanisms, and since

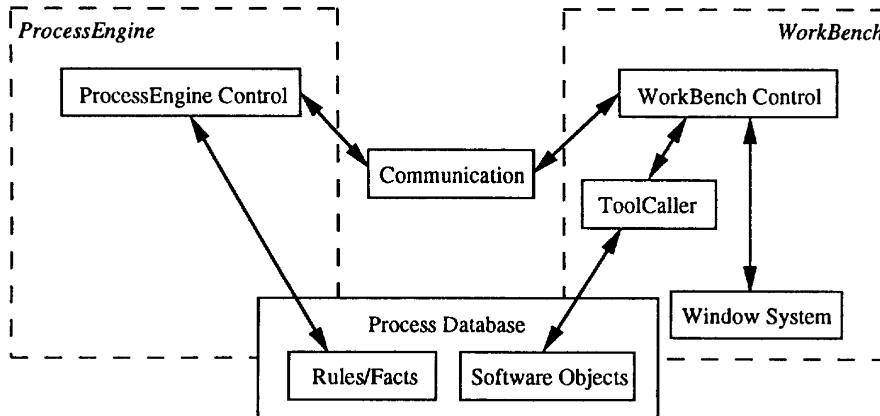

Figure 6: The Merlin implementation architecture.

MultiView can manipulate more than one software object at a time. Furthermore, when a user commits changes in a Merlin working context, Merlin closes the corresponding tool by killing the tool window without first saving the software object or checking for changes. Merlin can react to the exit codes of an invoked tool (e.g., a compiler); however, this behaviour does not work with complex tools such as MultiView which use their own data management. For example, if Merlin were to close MultiView in the same manner, it could lead to corruption of the MultiView database.

Therefore, further concepts were needed to achieve an appropriate level of cooperation between Merlin and MultiView (or any other tools to be integrated). This section focuses on these concepts, describes a standard protocol between Merlin and external tools, a standard library to realize the protocol, and the results of the experiment to integrate Merlin and MultiView.

3.2 Integration architecture

To achieve the integration of Merlin with MultiView (or indeed with any external tool), a facility was required to manage both the data and control integration aspects of the integration, including the transfer of information between Merlin and the tool (e.g., which component is to be manipulated). From the point of view of the Merlin project, it was clearly advantageous to use the MultiView-Merlin integration experiment to obtain a generic facility that could be used to integrate further external tools (such as design or requirement analysis tools).

The facility mentioned above we call an *adaptor* between Merlin and the tool; this adaptor must provide for two-way communication between Merlin and the other tool. Note that the adaptor amounts to a "control tool" in the terminology of Welsh introduced earlier. The general architecture of the adaptor is depicted in Figure 7. Since our goal was partially to explore the nature of a generic facility to integrate Merlin with external tools, some functionality has been provided in the adaptor (e.g., mechanisms to process software objects created or deleted by the external tool) not required immediately for the MultiView-Merlin integration experiment.

Neither of the original architectures for MultiView and Merlin were totally suited to the kind of integration that we required, even given the presence of the adaptor. Thus, a number of changes and enhancements to both were required. These changes not only enabled the construction of the adaptor, but also individually enhanced the two systems.

From the Merlin viewpoint, the starting point for the integration was in terms of Merlin's interaction with other tools. As explained earlier, in Section 2.2, Merlin provides editing facilities through the invocation and termination of discrete Unix tools, such as the *vi* editor, on a particular file. When the editing is complete, Merlin detects the fact that the tool is exiting and acts as determined by the process model. Communication between the Merlin WorkBench and the tool is via the Unix command line and the exit status of the process. The lifetime of the external tool is assumed to be the duration of the editing session. In addition, Merlin assumes that a given tool invocation operates on a single component. This interaction, while appropriate for simple tools such as *vi*, is inadequate for more powerful tools such as MultiView. A MultiView session will typically handle the editing of a number of components at the same time and the lifetime of the session should not be restricted to the editing of a single component. These limitations in the Merlin viewpoint were overcome by introducing the concept of external tools.

In order to facilitate the kind of interaction required with external tools, the *Merlin external tool control protocol* was defined. This protocol includes commands which allow Merlin to instruct the external tool to load components for editing or reading, change the access rights to

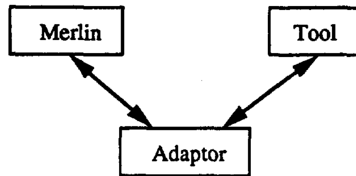

Figure 7: The general architecture for tool integration with Merlin.

components, unload components and terminate the external tool. Merlin can also use the protocol to query an external tool for the modification status of any loaded components. The full set of external tool control protocol commands is listed in the appendix. The lifetime of the external tool is not bound to an editing session and the external tool can operate on any number of components during its lifetime.

Exchanges of messages between the external tool and the Merlin WorkBench are synchronous with respect to the WorkBench. All messages are initiated by the WorkBench and the external tool is expected to respond. The protocol does not include any messages or commands initiated by the external tool to which the WorkBench must respond.

For Merlin to enact a given software process model, it demands complete control over the interaction of the external tool with the external environment (e.g., the relevant file system). The external tool must only load components when instructed, it must unload them when instructed and terminate when instructed. Otherwise, the external tool could be used to circumvent the constraints of the process model.

The above Merlin external tool facility was not only necessary for the MultiView-Merlin integration project, but is also likely to be a useful facility in future attempts to integrate Merlin with other tools. The integration of these tools with Merlin will involve the realization of the external tool control protocol in the tool.

From the MultiView point of view, the integration starting point was ControlView. As indicated earlier in Section 2.1, this is a window which is used by the MultiView user to load files, start views, unload files or terminate a MultiView session. These operations, however, were actually realized within the MultiView database process and were initiated by a ControlView process transmitting control messages to the database. In particular, ControlView is not implemented by a view process of the kind depicted in Figure 2. Unfortunately, the set of control messages provided within the MultiView implementation was not rich enough to fully implement a Merlin external tool. In addition, any view could issue a control message, thus violating Merlin's requirement for complete control over the tool. These limitations were overcome by designing a set of control messages, called the *MultiView control protocol*, and introducing the concept of a designated controller process.

The MultiView control protocol messages instruct the MultiView database process to load files into the database as structured representations of compilation units, delete units from the database, save units into files and terminate the MultiView session. The full set of MultiView control protocol messages is given in the appendix. In addition, the database process can be queried about the modification status of a unit. All other messages that initiate interaction with the file system were removed from the MultiView protocol. Controller processes are designated processes within a MultiView session; the database process will only honour control messages which come from the designated controller and allow only one controller process to be connected at any time.

The MultiView control protocol provides the mechanism required by the MultiView side of the integration experiment. It also led to a new design and implementation of ControlView for MultiView. When MultiView operates in isolation, the ControlView process is the designated controller process and interacts with the database using the control protocol.

Given these enhancements to MultiView and Merlin, the integration experiment consisted of the design and implementation of the *MultiView-Merlin adaptor*. The role of this adaptor can be seen in Figure 8, which is a specialisation of Figure 7 for the case of the specific integration experiment described in this paper. The adaptor is both a MultiView controller process and a Merlin external tool. It receives instructions from Merlin, encoded in the Merlin external tool control protocol and controls the MultiView session by exchanging MultiView control protocol messages with the database process.

3.3 Implementation

Although the focus of the implementation of the integration was the design and coding of the *adaptor*, it was necessary to first implement the external tool control protocol within Merlin and the MultiView control protocol within MultiView.

Enhancing Merlin involved both the specification of the external tool control protocol, and the implementation of this protocol both within the WorkBench and as a library to be linked into external tools. The protocol was implemented as messages transmitted and received over Unix sockets. The WorkBench was modified to interact with tools that could manipulate more than one file and could not be terminated by merely killing the associated process.

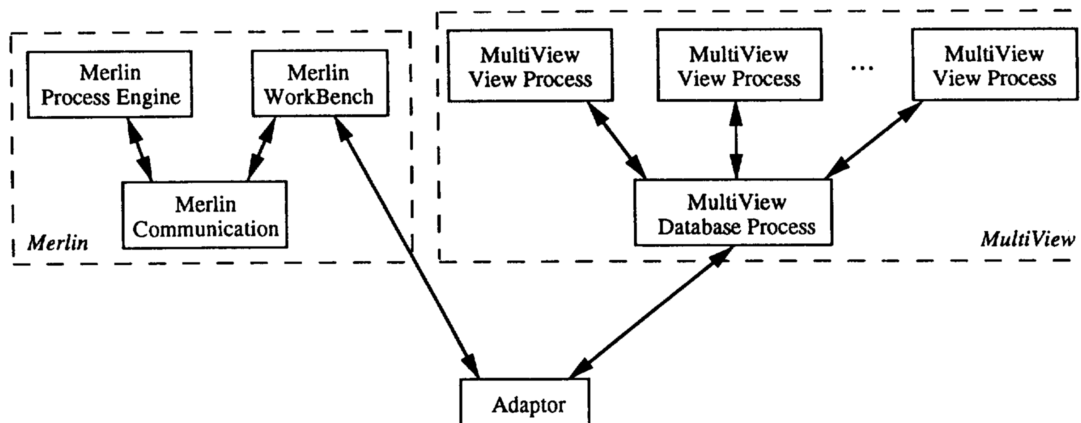

Figure 8: The architecture of the MultiView-Merlin integration.

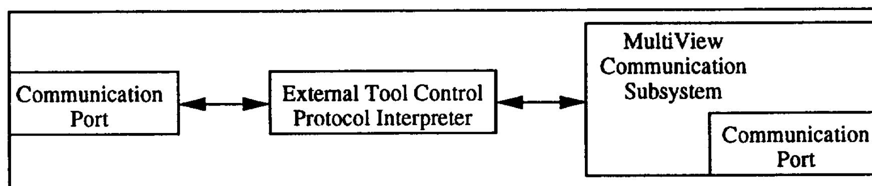

Figure 9: The MultiView-Merlin adaptor.

This was achieved by modifying the ToolCaller component of the WorkBench. Merlin has to decide whether a UNIX tool or an external tool is being called; this is defined by the process model. If a external tool is to be invoked, the ToolCaller has to communicate with the adaptor to send a message to start the external tool. Other messages, such as starting and finishing a component, and changing the access rights of a component, are also provided by the ToolCaller. The implementation of the invocation of UNIX-based tools remains unchanged.

In addition, the Merlin user interface was enhanced to provide user feedback about adaptor time-outs and to inform the Merlin user of problems emanating from MultiView.

Implementing the MultiView control protocol required some minor changes to MultiView's internal communication protocol and the resulting changes in the database to support the new commands. Communication between the MultiView database and view processes is implemented within the MultiView *Communication Subsystem* (or *CSS*). All MultiView view processes include an instantiation of the CSS. The CSS encapsulates queries and commands from views, and transmits them as data over a Unix socket to the corresponding CSS within the MultiView database process. Adding the MultiView control protocol commands to the existing MultiView protocol required extending the CSS to support the new message types. At this stage, the existing ControlView was modified to interact with the database using the new commands.

The adaptor was then implemented as a MultiView controller process that also accepted commands from the Merlin WorkBench. Figure 9 shows the structure of the adaptor, at the heart of which is an interpreter that maps the Merlin external tool commands into MultiView control messages; the adaptor also contains an instantiation of the MultiView CSS and a port to communicate with the Merlin WorkBench. Since it is a controller process, the adaptor is able to start a MultiView database and take control of its operation. Hence, when Merlin invokes the adaptor, the adaptor in turn invokes the MultiView database to initiate a MultiView session.

Unfortunately, it was not possible to use the external tool library implemented when Merlin was extended with the tool control protocol, because of the assumptions made about the form of interprocess communication to be used. Instead, the external tool side of this protocol had to be reimplemented to function within the context of a MultiView view process.

The close relationship between the external tool control protocol commands and the MultiView control protocol commands made the coding of the interpreter a simple task. This interpreter receives the tool control pro-

tocol commands and emits, via the CSS, the appropriate MultiView control messages. For example, a Merlin `start_component` message is mapped into a MultiView `load_unit` message. The `load_unit` message returns a unit identifier to the adaptor. A subsequent `finish_component` message, for the same component, is mapped onto a MultiView `delete_unit` message for the unit.

4 Conclusions and future work

This paper has described an experiment in tool integration, involving the integration of two very different tools whose integration was never anticipated. The tools were, in fact, developed on opposite sides of the world in isolation from each other. The two tools are a loosely coupled process-centred software development environment and a closely coupled integrated software development environment.

A general architecture for the integration of foreign tools was designed for the process-centred environment (Merlin). This architecture involves the use of an adaptor process. The architecture was then used in integrating an integrated software development environment (MultiView) with the process-centred environment.

The result was an integrated software engineering environment which retains the strengths of its individual components:

- Merlin's enactment of software processes represents an excellent approach to the problem of controlling cooperative software development, and
- MultiView provides very strong support for the manipulation of software objects at the point that a software developer is ready to perform such manipulation.

The resulting integrated software engineering environment also addresses weaknesses in each of the separate tools, weaknesses arising from issues which were outside the scope of the research projects concerned:

- Merlin only provides access to text editing utilities for editing software objects, instead of more specific editing facilities.
- MultiView does not address the issue of coordinating the work of multiple software developers and, in particular, provides no controls over the copying of software objects into the databases of several MultiView users.

The enhancements of both MultiView and Merlin, and the subsequent implementation of the adaptor, illustrated a number of points about designing and implementing tools intended to be integrated into integrated software engineering environments. Mechanisms to control the operation of the tools should be considered when the tool is designed. However, no assumptions should be made about how a tool may be invoked; instead the mechanisms should be primitive enough to allow the realisation of a range of tool control policies. Elucidation and refinement of these ideas is the subject of ongoing work.

Future work will also now extend the library of operations so far implemented to cover a message server which can connect to multiple, different tools.

The work described in this paper is also being used in the development of a general model of tool integration. This model will be used to define tool integration mechanisms such as those represented by the adaptor and the two protocols, and as the basis of automatically generating tool integration mechanisms (i.e., adaptors).

Acknowledgements

The development of the MultiView environment has been supported by the Australian Research Grants Scheme, the Australian Research Council, the Australian Computer Research Board, the Australian Telecommunications and Electronics Research Board, the Defence Science and Technology Organization, the University of Adelaide and the Flinders University of South Australia. The Merlin environment has been developed at the University of Dortmund and STZ, a Dortmund-based software house, and has been supported by the German Ministry for Research and Technology (BMFT), as part of the Eureka project ESF (Eureka Software Factory). The work of the fourth author was facilitated by a release-time scholarship from the University of South Australia.

The work in this paper has been carried out as part of an ongoing programme of cooperation between Australia and Germany in the area of software engineering; this cooperation has been funded under the auspices of the Department of Industry, Technology and Commerce (DITAC) of the Australian government and the German Ministry for Research and Technology (BMFT).

References

[AHM88] R. A. Altmann, A. N. Hawke and C. D. Marlin, "An integrated programming environment based on multiple concurrent views", *Australian Computer Journal*, Vol. 20, No. 2 (May 1988), pp. 65–72.

[KFFK91] M. I. Kellner, P. H. Feiler, A. Finkelstein, F. Katayama, L. J. Osterweil, M. H. Penedo and H. D. Rombach, "ISPW6 software process example", *Proc. First Int. Conf. on the Software Process*, M. Dowson (Ed.) pp. 176–186 (I.E.E.E. Press, Los Angeles, 1991).

[Mar90] C. D. Marlin, "A distributed implementation of a multiple view integrated software development environment", *Proc. Fifth Conference on Knowledge-Based Software Assistant* (Syracuse, New York, 1990), pp. 388–402.

[PS92] B. Peuschel and W. Schäfer, "Concepts and implementation of a rule-based process engine", *Proc. Fourteenth International Conference on Software Engineering* (Melbourne, Victoria, 1992), pp. 262–279.

[PSW92] B. Peuschel, W. Schäfer and S. Wolf, "A knowledge-based software development environment supporting cooperative work", *Int. J. of Software Engineering and Knowledge Engineering*, Vol. 2, No. 1 (1992), pp. 79–106.

Command	Parameters	Result
start_tool	tool	boolean
start_component	tool, component, unit-mode	boolean
finish_component	tool, component, unit-mode	boolean
finish_tool	tool	boolean
change_access	tool, component, unit-mode	boolean
is_changed	tool, component	boolean
are_deleted	tool	component-list
have_been_created	tool	component-list

Figure 10: The external tool control messages for Merlin.

[Rei84] S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems", *Proc. A.C.M. SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments {A.C.M. SIGPLAN Notices*, Vol. 19, No. 5 (May 1984)}, P. Henderson (Ed.), pp. 30–41.

[Rei85] S. P. Reiss, "PECAN: program development systems that support multiple views", *I.E.E.E. Transactions on Software Engineering*, Vol. SE-11, No. 3 (March 1985), pp. 276–285.

[TN92] I. Thomas and B. A. Nejmeh, "Definitions of tool integration for environments", *I.E.E.E. Software*, Vol. 9, No. 2 (March 1992), pp. 29–35.

[Was89] A. I. Wasserman, "Tool integration in software engineering environments", *Proc. Software Engineering Environments: An International Workshop on Environments* (Chinon, France, September 1989), *Lecture Notes in Computer Science*, Vol. 467, pp. 138–149 (Springer-Verlag, Berlin, 1989).

[WY91] J. Welsh and Y. Yang, "Tool integration techniques", *Proc. Sixth Australian Software Engineering Conference* (Sydney, July 1991), pp. 405–418.

Appendix

The Merlin external tool control messages

An instantiation of the Merlin working context is able to control the operation of tools that manipulate software objects that fall into the domain of the process model. As shown in Figure 10, the Merlin external tool control protocol defines messages that provide this control over an external tool. Each message has an associated acknowledgement message that indicates whether or not the operation has been successfully carried.

The following are the external tool control messages used in the MultiView-Merlin integration experiment. In all but the last, the boolean value being returned represents only the acknowledgement for the completion of the command.

start_tool(tool):
returns boolean

tool: The name of the tool to be started (e.g., MultiView).

boolean: Acknowledgement of command completion.

This message is used to start the external tool indicated. It is invoked at most once for a given tool within a working context session, namely when the user first requests the use of the tool by clicking the edit/list button in the tool selection window. The **boolean** value indicates whether or not the tool has been started successfully. Merlin needs this information to issue further messages, reinvoke the tool, or to calculate a time-out (to inform the developer that the tool is currently not available).

start_component(tool, component, unit-mode):
returns boolean

tool: The tool which is to receive the current message.

component: The name of the component to be opened.

unit-mode: Either "read-write" or "read-only", indicating the appropriate access rights to the component.

boolean: Acknowledgement of command completion.

This message starts the indicated tool on the specified component. For the MultiView-Merlin integration, the **tool** is bound to "MultiView". The **unit-mode** determines whether the tool may modify the component (read-write) or merely browse it (read-only). The **boolean** value informs Merlin whether or not the invocation has been executed successfully.

finish_component(tool, component, unit-mode):
returns boolean

tool: The tool which is to receive the message.

component: The name of the component to be closed.

unit-mode: Either "do-save", "optional-save" or "do-not-save", indicating the disposition of the component being closed.

boolean: Acknowledgement of command completion.

Terminates processing on a particular component. If any references or copies of the component exist within the external tool, they must be purged. In the MultiView context, this implies that views must be terminated which relate to the component and the corresponding unit must be removed from the database. Because of the Merlin concept of transactions, it may be necessary to prohibit or to demand the saving of the component; the **unit-mode** parameter determines the actions that the tool may take if the component has been edited, but not yet saved. If a value of "true" is returned, indicating that the component has been successfully closed, Merlin can read the new state for the component from the document state window and start the processing required to refresh the working context.

finish_tool(tool):
returns boolean

tool: The name of the tool to be closed (e.g., MultiView).

boolean: Acknowledgement of command completion.

Command	Parameters	Result
register_controller		
deregister_controller		
start_database	controller-address	
load_unit	file, unit-mode	unit
save_unit	unit	
delete_unit	unit	
change_access	unit, unit-mode	
is_changed	unit	boolean
terminate_multiview		

Figure 11: MultiView control protocol messages.

This command is used to teminate the external tool, usually just prior to exiting the Merlin workbench. The saving of modified components must be initiated by Merlin prior to issuing the **finish_tool** message, by using **finish_component** for each component accessed by the tool. Merlin must also initiate the closing of all components which have been opened under read access. The **boolean** value indicates whether or not the message has been executed completely.

change_access(tool, component, unit-mode):

tool: The tool which is to receive the message.

component: The name of the component whose access rights are to be modified.

unit-mode: Either "read-write" or "read-only", indicating the new access rights for the component.

boolean: Acknowledgement of command completion.

This command is used to change the access rights of the indicated component. This message is needed, for example, after the working context has been refreshed and the access rights to the component differ from those applying previously. The possible changes are from "read-write" to "read-only", or vice versa. The **boolean** value indicates whether or not the change has been executed successfully.

is_changed(tool, component):
returns boolean

tool: The tool which is to receive the message.

component: The name of the component whose status is required.

boolean: Acknowledgement of command completion.

This command queries whether a given component has been modified. This query may be used when exiting to determine what action must be taken by Merlin when terminating the processing of a tool. The onus is on Merlin to initiate any saving of components. The **boolean** value indicates whether or not the component has been changed.

In addition to the control messages listed above, there are two further messages which have not been used in the MultiView-Merlin integration but which will be needed for other external tool integration.

are_deleted(tool):
returns component-list

tool: The name of the tool receive this message.

component-list: A list of the components deleted by the tool.

This command queries which components have been deleted by the indicated tool.

have_been_created(tool):
returns component-list

tool: The name of the tool receive this message.

component-list: A list of the components created by the tool.

This command queries which components have been created by the indicated tool.

The MultiView control protocol messages

The MultiView control protocol is implemented as a subset of MultiView's internal communication protocol. This internal protocol defines a set of messages, consisting in each case of a command field and a set of parameters (the nature of which is determined by the command). For each command, a corresponding acknowledgement message is defined. The acknowledgement message may also contain data fields, such as the boolean result in the acknowledgement to the **is_changed** command. The CSS converts the commands and their parameters into a linear bit stream that is written onto the Unix socket connected to the corresponding socket in the MultiView database process. Once the database has processed the command, an acknowledgement message is constructed and transmitted back to the view. This acknowledgement always contains a status field indicating whether the operation succeeded or, if it failed, the reason for the failure.

Figure 11 summarizes the commands that make up the MultiView control protocol. Some details of each command are presented below.

register_controller:

Attempt to register the view originating the message as the designated controller process. This command is honoured only if no other view is currently registered as the controller.

deregister_controller:

Deregisters the originating view as the designated controller process.

start_database(controller-address):

controller-address: The IP address and socket number of the communications port within the controller process.

Starts a database process communicating with the controller through the port addressed by **controller-address**. This command cannot be implemented by the controller purely in terms of message passing. Instead, a database process has to be created and communication establised. However, it is conceptually neater to consider it as a message. The **controller-address** parameter is transmitted as a command line parameter to the database process.

`load_unit(file, unit-mode)`:
`returns unit`

file: The name of the Unix file containing the source of the component.

unit-mode: Either "read-write" or "read-only", indicating whether the MultiView user may make changes to the component.

unit: The identifier allocated by the database that identifies the new unit.

Reads a software component (e.g., a Modula-2 module) from the file given by `file` and creates a unit in the MultiView database in structured form (abstract syntax tree) for the component. The returned `unit` value is used to identify the new unit in subsequent commands. The command will fail if either the file cannot be accessed or if its contents could not be parsed as a valid component in the language.

`save_unit(unit)`:

unit: The name of the unit to be saved.

Save the unit denoted by `unit`. The database saves the unit into the file from which it was originally read. The command fails if the file cannot be written.

`delete_unit(unit)`:

unit: The name of the unit to be deleted.

Delete the unit denoted by `unit` from the database.

`change_access(unit, unit-mode)`:

unit: The name of the unit to be modified.

unit-mode: Either "read-write" or "read-only", indicating the new access that MultiView has to the component.

Changes the access mode of the unit denoted by `unit` to the given value.

`is_changed(unit)`:
`returns boolean`

unit: The name of the unit about which information is required.

This comand queries whether the unit denoted by `unit` has been modified. A boolean value is returned indicating the status.

`terminate_multiview`:

Terminates the current MultiView session. On receipt of a `terminate_multiview`, the database shuts down all the connected views and exits.