Merlin: Supporting Cooperation in Software Development through a Knowledge-based Environment[1]

G. Junkermann*, B. Peuschel#, W. Schäfer*, S. Wolf#

* Informatik X
University of Dortmund
D - 44221 Dortmund
Germany

STZ - Gesellschaft für Software-Technologie mbH
Helenenbergweg 19
D - 44225 Dortmund
Germany

1 Introduction

Merlin[2] is a prototype Process-centred Software Development Environment (PSDE), developed within the context of the Merlin project carried out at University of Dortmund in cooperation with STZ, a Dortmund based software house. This prototype uses a rule-based technique to describe and enact a software process. Users of Merlin are either software developers or managers who are involved in a software process to produce a product. A further kind of user or group of users respectively called the process engineer(s) are responsible for defining a particular process in terms of Merlin rules (or rather a dedicated process modeling language as will be explained later), i.e. they customize a Merlin PSDE to a particular software process or project.

The major benefit of using an environment like Merlin for software production is sophisticated team support, i.e. support for coordinating access to shared information on different levels of granularity (e.g. from more or less complete systems of modules or documents down to a procedure definition in the export list of a single module), and dedicated message servers for broadcasting information about project states, (urgent) tasks to do, and getting feedback of completed work packages, etc. A further achievement of such an environment is the computer supported integration of development and management activities. For example, project managers are able to retrieve on-line information about the current project state at any time and developers are immediately informed about any necessary actions to be taken or any constraints applying to executing activities.

This paper describes the main concepts behind the implementation of the Merlin prototype and sketches current and further work within the research project evolving from the current experience with the prototype.

1. This work has been supported by the German Ministry for Research (BMFT) as part of the Eureka project ESF (Eureka Software Factory) and is supported by the Provincial Ministry for Research (MWF) of the state of Northrhine Westphalia.

2. Merlin is not an acronym. Merlin stands for the idea to support software construction based on a well-defined set of building blocks of software and their respective attributes and relations. Using this information, a process-centred software development environment like Merlin can provide maximum information to developers (or managers resp.) to support them to perform activities in the software process. This idea relates to the old english fairy tale about King-Arthur. The wizard Merlin is supposed to have built the Stonehenges in southern England (a set of well-defined building blocks and relations) and the wizard was able to foresee the future at least to a certain extent such that king Arthur could make reasonable plans for his further moves in ruling his country.

The novel features of Merlin described in this paper are: (1) the trisection of software process programs to simplify the process definition and to support changes of the software process (on the fly) in an easy manner, (2) a sophisticated graphical language which is mapped to the Merlin rule-language to specify processes and (3) a special transaction concept which is embedded into the Merlin rules to support cooperative work of multiple users.

In more detail the next chapter describes the Merlin user interface. Sections 3 and 4 respectively explain the Merlin support for the process engineer, i.e. the way how rules and more sophisticated language constructs are used to specify an executable software process. Section 5 sketches our non-standard transaction protocol supporting flexible cooperation and avoiding some of the disadvantages of standard transaction mechanisms. Section 6 describes the underlying architecture. Section 7 concludes with remarks on current and further work within the Merlin research project.

2 User Interface

Any process definition whose execution is supported by Merlin consists of the following entities:

- Activities: a collection of tasks which achieve a goal related to the production of a software product (e.g. specify, edit, compile or test a module);
- Roles: groups of activities which are logically highly related (e.g. project manager, technical leader, programmer);
- Documents: objects of any type that are produced during the software development process (e.g. modules, documentation, 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 document is bound to a set of activities able to be performed on the document and to a set of tools supporting these activities. For example, a module to be programmed is edited by a language-sensitive editor and a module to be tested is executed by a debugger. In addition, users are associated with one or more roles.

As a result of this view of the software process, users are assisted primarily by the display of all relevant information in a *working context* associated with their current *role*. The working context displays the *documents* to be manipulated, their *dependencies* to other documents and the *activities* able to be performed on each document. This varies from other PSDEs (like e.g. Marvel [BK92,PSL91]) in that the user is presented with all information needed to perform a task and can be confident that there is no more relevant information available (contrary to operating systems for example, where the user has to know which activities exists, which documents can manipulated by these activities and, often, where to locate the appropriate tools).

A working context in Merlin is displayed on the screen in the following way: documents are represented as boxes, the boxes have context-sensitive menus attached which contain the activities that can be and/or have to be performed on the corresponding documents. Labelled arrows between the boxes describe the inter-document relationships.

Selection of a menu item triggers the execution of an activity and hence the invocation of the corresponding tool. Possible menu items are displayed in Fig. 1 where a user has read access to the specifications *m1_spec* and *m2_spec* as well as *read/write* access to the documents *m1_c* and *m2_c*. After selecting the *ascii_editor* for *m1_c*, two additional windows appear on the screen, one for editing and the second to set the new state (see Fig. 2).

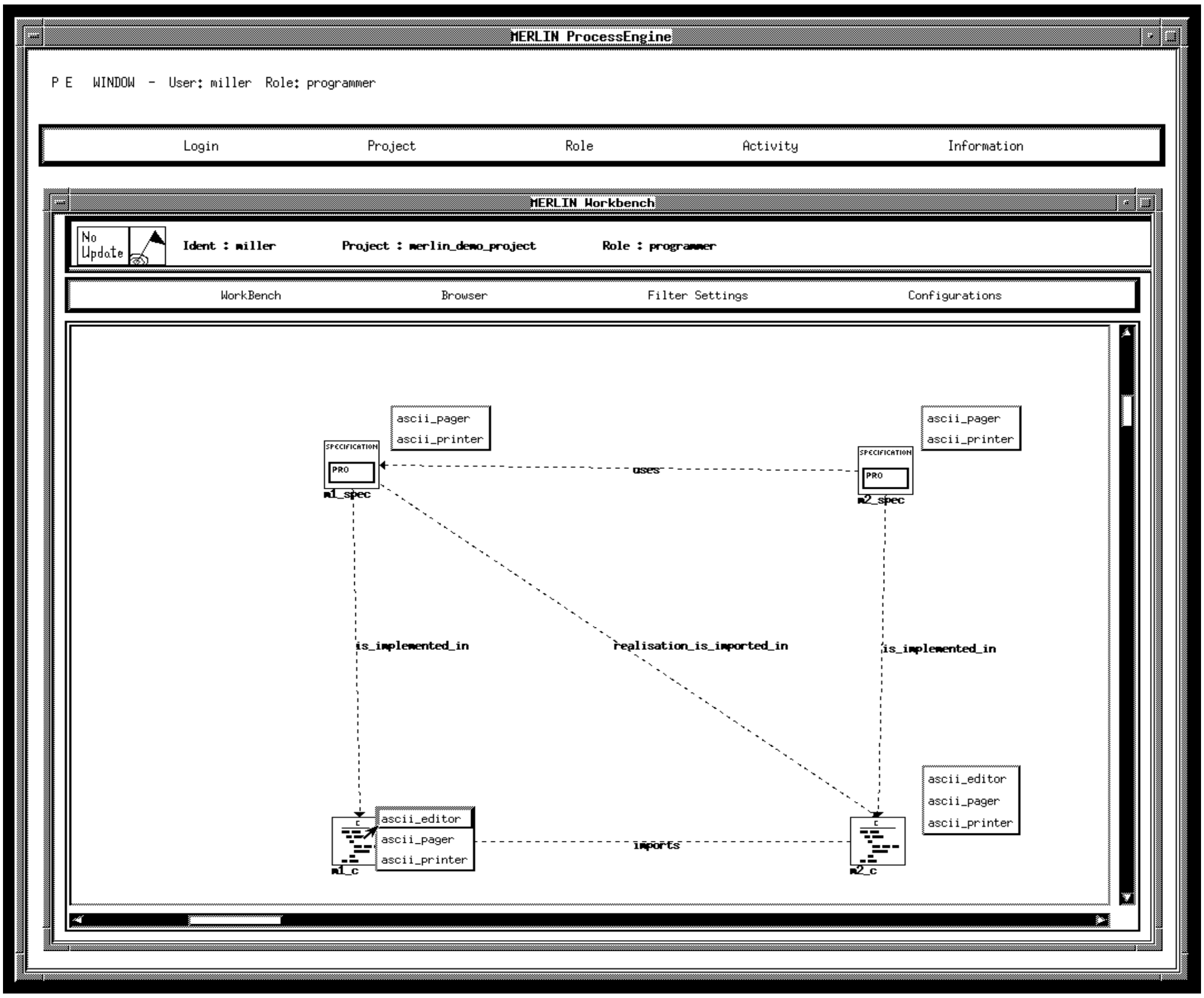

Fig. 1 A Merlin working context

The top line of the **ProcessEngine** window and the **WorkBench** window displays the user's name and the current role (in our example *miller* is currently in the role *programmer*). The menu items of the ProcessEngine window allow the user to login, select a project, select a role within the project, execute activities (e.g. to start the working context), and ask queries about the current project state. To query the project state provides information which is not automatically displayed within a user's working context and enables any user to get a complete overview of the project, including information not directly relevant to the current activities (e.g. all members of the project team). The menu items of the Workbench window allow the user to customize his working context. To customize the working context means, for instance, selecting a preferred tool to perform particular activities (e.g. a preferred editor).

When an activity is performed whose result influences another or the same working context, (e.g. document states have changed or new documents have been created), the developer is informed about this situation by the update flag in the top left corner of the Workbench window. If he clicks this flag the changes are propagated in his working context (e.g. a new object or a number of new objects are added or a menu is extended by a further item indicating new activities which can be performed on the corresponding objects). This avoids to have highly userunfriendly, more or less permanent refreshes of the working context which are outside the user's control.

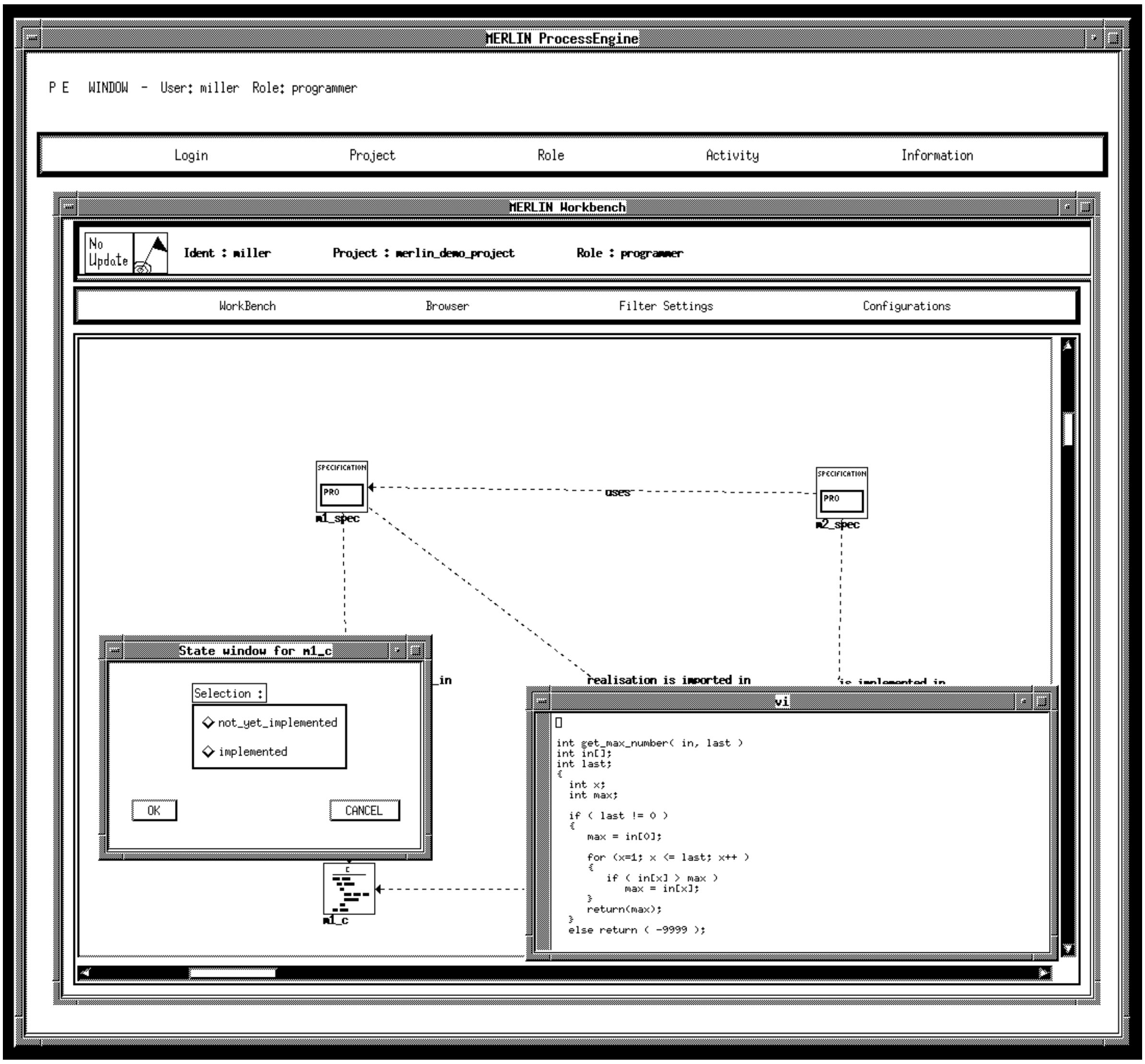

Fig. 2 Editing a module in the Merlin working context

As a brief example how Merlin supports cooperative software development, we take an excerpt from a Merlin instance which supports the process as defined in the ISPW6/ISPW7 example [KFF+91]. For a more elaborate description how that process is modeled and supported by Merlin we refer to [PSW92].

Assume two developers Smith and Miller performing two roles, namely a *quality assurance engineer* and a *programmer* who are responsible for testing modules and for coding/reviewing modules respectively. The process is defined as follows: The *quality assurance engineer* performs extensive testing of a module based on a predefined test plan, after a module has been coded, reviewed and briefly tested before by the responsible *programmer*.

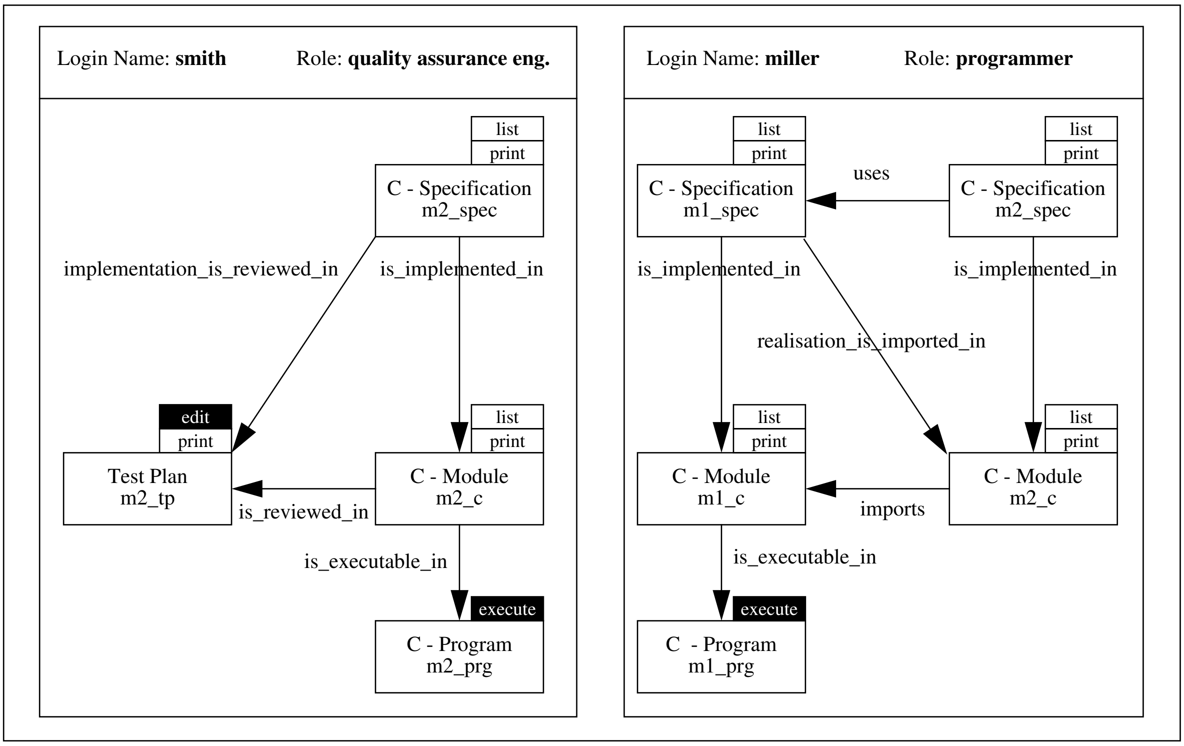

Fig. 3 Cooperation between a programmer and a quality assurance engineer

In our concrete example, we further assume that Miller could currently work on testing module *m1_c* and that Smith could work on creating (editing) the test plan *m2_tp* for module *m2_c*. The corresponding working contexts for these two developers are given in Fig. 3 (For reasons of saving space, we do not give complete screen dumps, but just the excerpts from a screen which are of concern for this example). We now assume that Miller has finished coding and testing module *m1_c*. His working context becomes empty. Smith's working context is refreshed, because he is responsible for testing module *m1_c*, i.e. all information needed for testing *m1_c* is displayed additionally in his working context. This part of our scenario is given in Fig. 4.

If module *m1_c* does not pass the quality tests performed by Smith (which is indicated by the corresponding status information given by Smith), Miller's working context is displayed again as pictured in Fig. 3.

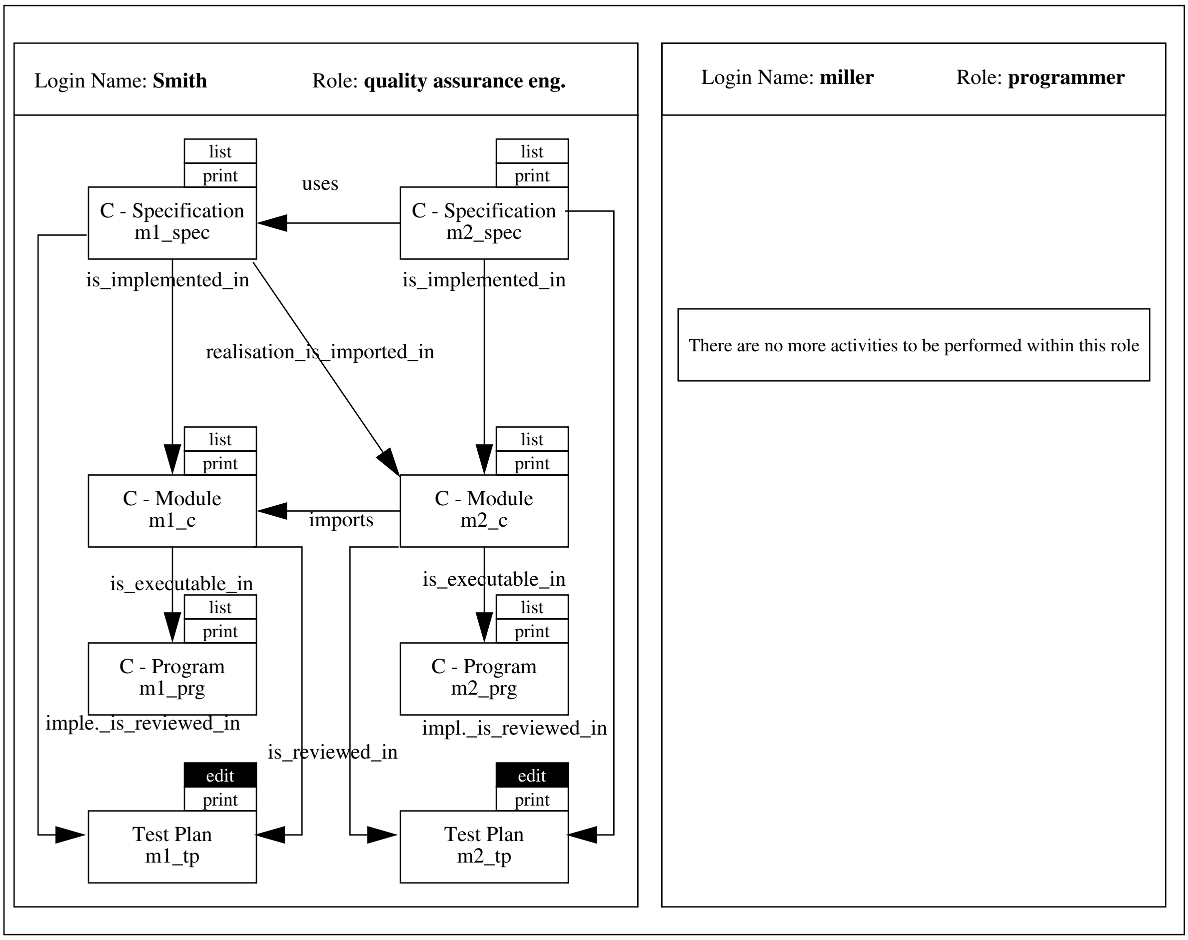

Fig. 4 **Updated working contexts**

3 Process Definition and Enactment

The process engine is that part of the Merlin environment which builds and refreshes user's working contexts by executing the given process definition. In principle, constructing and refreshing working contexts are based on the definition of activities and corresponding documents which are manipulated by performing the activities. Each activity has a number of associated preconditions which define e.g. the roles who could perform the activity, the person who is responsible for it, the corresponding access rights to the appropriate (set of) document(s), or the dependency with other activities. An activity whose preconditions are all true could be carried out (and is thus displayed in a working context). Performing an activity could result in new preconditions becoming true (based on the user input of status information, cf. Fig. 2) and consequently new activities to be displayed. A process engine's major job is thus to evaluate all preconditions and to refresh all displayed working contexts accordingly.

In addition, the process engine should not only display working contexts but it should also be able to explain current and previous process states on request. For instance, the environment should be able to answer questions from a user like *Why should I code module m1_c?*, *Who is involved in the project?*, *Who changed the specification for module m1_c? What are the time constraints for coding m1_c?*, i.e. questions which could be asked by selecting the information button (cf. section 2).

Furthermore, an important requirement is support for changes on the fly which even happen frequently. A process can usually not be fully determined in advance either because of unforeseen events (e.g. sickness of staff, lack in skills, break-down of machines) or because parts of the process depend on decisions during the course of the process (e.g. introduction of a new test strategy, new test cycles, a new experimental development path). For a more detailed and systematic discussion of what kind of changes may happen cf. [MS91]. Note that the requirement of changes on the fly requires an interpretative approach as the basis for the process engine and makes a compiler unusable.

Finally, the language used to define a software process should have a clearly defined semantics and should be easy enough such that a programm can be interpreted efficiently.

These requirements led us to select a rule-based language, namely a PROLOG-like language as our process implementation language. Thus, the construction of a working context or other questions about current and previous process states are uniformly handled as PROLOG goals which can or cannot be satisfied by the corresponding process definition.

A major disadvantage of rule-based languages (like PROLOG) is the lack of structuring mechanisms for facts and rules and mechanisms to keep the facts and rules consistent. This problem can, however, be avoided to a large extent in the special context of process modeling by using the special Merlin strategy, which we describe in detail in the sequel of this section and the next section.

In Merlin, a software development process is described on three different levels, namely the **Kernel**, the **Process** and the **Project**. The Kernel provides a predefined set of rules, which manage the ProcessEngine and WorkBench windows and their contents and which build and update the working context information. The Kernel has only to be changed when the user interface paradigm (as explained in section 2) is changed or new features like configuration management are added. It is thus the most stable part of the whole description.

Having these three levels of modeling, changes of the software process do not require changes of the Kernel (the only level using rules) but only changes of facts (which can be held much more easily in a consistent state). Therefore, the Kernel acts as an inference machine for software processes using the facts describing the Process and the Project as input.

The Kernel rule set is structured in five so called rule clusters namely **StartUp**, **WorkingContext**, **ChangedStates**, **TransactionManager** and **LockManager**. StartUp contains all rules managing the login of a user, the project, role, activity and information selection of the ProcessEngine window. WorkingContext contains all rules to select the information needed to display a working context. ChangedStates contains the rules to update the process information, if states of documents have been changed. The TransactionManager and LockManager are explained in section 5.

As an example consider a part of the rule defining the existence of a working context (see Fig. 5). Preconditions include that a person (identified by the parameter *Ident*) in a project (parameter *Project*) performing a role in this project (parameter *Role*) has some responsibilities in this project. Responsibilities is a fact explained below. In addition, after having identified his responsibilities, the corresponding documents must be in a state allowing to manipulate them according to the defined responsibilities. This is checked by executing rule documents which is for the sake of simplicity not explained in full detail here. Fact *roletype_document_work_on* is however explained below.

```
working_context (Ident, Project, Role, Menu_Activity_List, Document_List, Relation_List,):-
        responsibilities (Ident,Role,Responsibility_List),
        documents (Role, Responsibility_List, Document_List),
        ...
documents (Role, Responsibility_List, Document_List) :-
        ...
        document_state(Doc,State),
        roletype_document_work_on (Role, Type, State, Rdoctypes),
        ...
```

Fig. 5 Definition of rule *working context*

If an activity in the development process changes the state of a document this new state is inserted to the knowledge base. Changes of a document state usually cause state changes of other documents. This is defined by consistency conditions and automation conditions (see below).

Document states are changed using so called envelopes (cf. [PSW92,PS92]). An envelope encapsulates either an interactive tool or a batch tool. In case of a batch tool the envelope receives the new state directly from the tool, in case of an interactive tool the envelope demands a new document state from the developer.

Rules *changed_states* of rule cluster ChangedStates define the goal to look for further document states to be changed by an envelope. The first and second rule define the state changes depending on a consistency or automation condition (see Fig. 10 and Fig. 11). The third rule defines the invocation of batch tools (this is defined by the built-in CALL which implements the tool envelope). The last rule defines state changes of related documents.

The rules *working_context* and *changed_states* are called in a rule which is the main rule of cluster StartUp (Further rules in StartUp calculate menu items of the ProcessEngine window and react on the user inputs (login, project selection etc.)). This rule is responsible to create the working context and to wait for state changes initiated by an activity. Then, rule *changed_states* is executed and a new working context is calculated for the for the developer.

```
changed_documents (Ident, Project, Role, Document, State) :-
        document (Document, Type, Path),
        consistency_condition (Type, State, New_state, Condition_list),
        ...
        changed_documents (Ident, Project, Role, Document, New_state).
changed_documents (Ident, Project, Role, Document, State) :-
        document (Document, Type, Path),
        automation_condition (Type, State, New_state, Condition_list),
        ...
        changed_documents (Ident, Project, Role, Document, New_state).
changed_documents (Ident, Project, Role, Document, State) :-
        document (Document,Type,Path),
        document_type_tools (Type, Call, Access, State, [New_state_failure,
                                                        New_state_success]),
        document_call (Document,Call,Program_to_call),
        CALL (program_call, Document, Program_to_call, New_state_failure,
                                                  New_state_success, New_state),
        REMOVE (document_state(Document, Old_state),
        INSERT (document_state(Document, New_state),
        document_state (Document, New_state),
        changed_documents (Ident, Project, Role, Document, New_state).
changed_documents (Ident, Project, Role, Document, State) :-
        act_on_related_documents (Ident, Project, Role, Document, State).
```

Fig. 6 Definition of rule *act_on_document*

The main idea to simplify process- and project-descriptions is that the Kernel predefines the predicates to be used for describing the Process level and the Project level. Defining new processes and projects basically means to fill in values of parameters of these predicates, as we will explain now.

The first step of modelling a Process is to define the document types that are used in the software process. For example, the fact *document_type_states* defines all possible states for a particular document type. As an example, the definition of the states of document types *specification* and *c_module* as used in section 2 is given in Fig. 7.

```
document_type_states (specification,[incomplete, not_yet_designed, designed, complete]).
document_type_states (c_module, [incomplete, not_yet_implemented, implemented, not_yet_compiled,
        compiled, not_yet_tested, complete]).
```

Fig. 7 Definition of document_type_states

Fact *document_type_tools* defines which tools can be used in which states and which access rights have to be granted, i.e. this information is the basis for computing the menu lists attached to documents in the working context display (cf. section 2). As an example see Fig. 8. The first fact of this example must be read as: If a *c_module* can be accessed with *readable* access an *ascii_pager* is presented. The second fact is to be read as: If a *c_module* can be accessed with *readable* access an *ascii_printer* is presented. The third fact must be read as: If a *c_module* can be accessed with *writeable* access an *ascii_editor* is presented; the tool is only presented if the document is in the state *not_yet_implemented*; the possible new state of the document after having it manipulated is either *not_yet_implemented* or *implemented*.

```
document_type_tools (c_module, ascii_pager, readable, Any_state, []).
document_type_tools (c_module, ascii_printer, readable, Any_state, []).
document_type_tools (c_module, ascii_editor, writeable, not_yet_implemented, [not_yet_implemented,
        implemented]).
document_type_tools(c_module, c_compiler, executable, not_yet_compiled, [compiled_with_errors,
        compiled]).
```

Fig. 8 Definition of *document_type_tools*

Fact *document_relation_type* specifies the relation type between document types. This information is required by Merlin to be able to draw the lines between documents in the working context and to change document states depending on changes of related documents. Some of the document relation types used in our example are listed in Fig. 9. For example, the first fact is to be read as: Between a document type *specification* and a document type *c_module* a relationship called *is_implemented_in* exists.

```
document_relation_type (is_implemented_in, specification, c_module).
document_relation_type (uses, specification, specification).
document_relation_type (realisation_is_imported_in, specification, c_module).
document_relation_type (implementation_is_reviewed_in, specification, test_plan).
document_relation_type (imports, c_module, c_module).
document_relation_type (is_executable_in, c_module, c_program).
document_relation_type (is_reviewed_in, c_module, test_plan).
```

Fig. 9 Definition of *document_relation_type*

State changes of documents are triggered by so called consistency conditions and automation conditions. They describe how states of related documents have to be changed if either the user or a batch tool have changed the document's state. For example, if quality assurance engineer Smith in Fig. 4 detects errors in module *m1_c* which require the change of the specification, the state of the corresponding specification *m1_spec* is set to *not_yet_specified* (i.e. the specification) and the state of module *m1_c* is set to *incomplete*.

Consistency conditions are used to preserve the consistency of the process, if the change of a document state demands resetting of states of other documents, e.g. a *c_module* has to be re-edited because the test executed after the first editing of the *c_module* demands changes in the *c_module*.

Automation conditions change states of documents to enable new activities to be performed if the proceeding activity has been completed correctly. These conditions support the automation of the process (invocation of batch tools and setting of states to enable activities to be carried out in further phases of the process (e.g. after having finished the specification phase the coding phase can start)).

Consistency conditions and automation conditions are defined by facts *consistency_condition* and *automation_condition*.

```
consistency_condition (c_module, Any_state, incomplete, [[source, is_implemented_in, not_yet_designed],[source, realisation_is_imported_in, not_yet_designed]]).
```

Fig. 10 Definition of *consistency_condition*

Fact *consistency_condition* in Fig. 10 have to be read as follows:

The state of a *c_module* is changed to *incomplete* independent of the current state (*Any_state*), if either the state of a source document of the relationship *is_implemented_in* is changed to *not_yet_designed*, or if the state of a source document of the relationship *realisation_is_imported_in* is changed to *not_yet_designed*. That means a *c_module* cannot be accessed anymore if one of the corresponding specifications has to be changed

```
automation_condition (c_module, incomplete, not_yet_implemented, [[source, is_implemented_in,
        complete], [source, realisation_is_imported_in, complete]]).
automation_condition (c_module, implemented, not_yet_compiled, [[destination, imports, complete]]).
```

Fig. 11 Definition of *automation_condition*

The two *automation_condition* facts in Fig. 11 have to be read as follows:

The state of a *c_module* is changed from *incomplete* to *not_yet_implemented* (i.e. the *c_module* may be programmed now), if the states of all source documents of the *is_implemented_in* relation have state *complete* (i.e. the *specification* of the *c_module* has been *completed*) **and** the states of all source documents of the relation *realisation_is_imported_in* have state *complete* (i.e. the *specification* of all imported *c_modules* have been *completed*).

Next, the state of a *c_module* is changed from *implemented* to *not_yet_compiled*, if the states of all destination documents of the *imports* relationship are *complete*, i.e. a *c_module* can only be compiled, if the work on all imported *c_modules* has been finished[1].

Facts *consistency_condition* and *automation_condition* are used by the Kernel as follows: The first rule given in Fig. 6 checks the existence of a consistency condition for the current document. If a consistency condition does exist, the state of the document is changed (following the definition in Fig. 10) and rule *changed_documents* is called for the new state again. If there does not exist a consistency condition the second rule *changed_documents* searches for an automation condition. If a automation condition does exist, the state of the document is changed (following the definition in Fig. 11) and rule *changed_documents* is called for the new state again. In the third rule *changed_documents* the invocation of a batch tool is checked. If a tool can be invoked the rule *changed_documents* is called again. If all conditions for the current document have been checked the last rule *changed_documents* is called again to perform necessary changes for all related documents.

Fact *roletype_document_work_on* specifies the role name, and the documents as well as the access rights to these documents to be presented in the working context of a user performing this

1. Those examples are not meant to indicate that Merlin only supports waterfall-like production processes. In fact, any kind of incremental production process can be defined by the Merlin approach. It is just the case that the example used here, namely the ISPW6/ISPW7 example prescribes a mainly waterfall-like process.

role. As an example consider the definition of the role programmer and the corresponding documents.

```
roletype_document_work_on (programmer, c_module, not_yet_implemented,
        [[specification, complete], [c_error_report, with_errors], [review, review_rejected]]).
roletype_document_work_on (programmer, c_program, not_yet_executed,
        [[c_module, not_yet_tested]]).
```

Fig. 12 Definition of *roletype_document_work_on*

The two facts defining the role of a programmer have to be interpreted as follows:

A programmer gets a *c_module* in his working context, if this module has the state *not_yet_-implemented*. In this case, the corresponding *specification* is displayed within his working context with read access if the *specification* has the state *complete*, and an *c_error_report* is displayed within the working context with read access if this report has the state *with_errors*. Furthermore, the corresponding *review* is displayed if it has the state *review_rejected*. Note that the documents specified in the list are only displayed, if there exists a document relation between these documents and the *c_module*.

The second fact must be read as: The programmer can get a *program* in his working context, if the *c_program* has the state *not_yet_executed*. In this case, the corresponding *c_module* is displayed within his working context with read access if it has the state *not_yet_tested.*

The definition of real projects on the project level now becomes very simple. The main information to be provided is the names, roles, and responsibilities of persons participating in a project, and the types and names of documents to be produced. As examples we describe the facts *project*, *has_roles*, and *responsibilities*. The example in Fig. 13 describes a project called *merlin_demo_project* with two participants called *smith* and *miller* who perform the roles of a *programmer* and *quality_assurance_engineer* respectively. Miller is responsible for *m1_c* and smith is responsible for *m2_tp*.

```
project (merlin_demo_project, [smith, miller]).

has_roles (miller, [programmer]).
has_roles (smith, [quality_assurance_engineer]).

responsibilities (miller, programmer, [m1_c]).
responsibilities (smith, quality_assurance_engineer, [m2_tp]).
```

Fig. 13 Definition of a project

Of course, this information is usually defined by a project management tool and is not necessarily given by the process engineer just as other facts on the Project level. In addition, it is worthwhile to note here again that the above informally given semantics of the various facts is formally and precisely defined by the rules of the Kernel.

The Merlin *Forward Chaining Rules* proposed in [HJP+89,PSW92,PS92] have been skipped in this new modelling approach for the following reason:

Because of the Merlin user interface paradigm (more than one activity can be executed in one working context and changes of document states trigger new activities in other users' working contexts) the evaluation of preconditions and postconditions of activities has become very complex, because a hierarchy of conditions has to be evaluated. Of course, it is possible to specify the evaluation of this hierarchy of preconditions with production rules (which have been used as Merlin forward chaining rules). The result is a break down of the hierarchy into a lot of artificial (technical) rules which do not have any activities in their bodies as for example happens in Marvel [KFP88] or Matisse [GP93]. In contrast to Marvel which allows only one activity to be performed by one user at a certain point in time, the working context paradigm in Merlin requires more than one activity to be executed by one user at a certain point in time. The evaluation of the hierarchy of preconditions and postconditions is much better defined in a PROLOG-like style and, therefore, defined by PROLOG goals in the Merlin Kernel (cf. Fig. 5 and Fig. 6).

4 A Dedicated Process Modeling Language

For the specification of the Process level, Merlin provides a graphical design-language based on the concepts of **Extended-Entity-Relationship(EER)-diagrams**, **finite-state-machines** and **predicate-logic.**

The predicates being part of the Process level can be subdivided in two groups: (1) those specifying the static part of the process (i.e. *document_type_states*, *document_relation_type* and *roletype_document_work_on*) and (2) those specifying the dynamic part of the process (i.e. *document_type_tools, automation_condition* and *consistency_condition*).

The EER-model is based on the definition given in [EGH+93] and offers as modeling concepts entities, attributes and relationships as well as refinement (not explained here) and inheritance for structuring purposes.

The EER-diagram in Fig. 14 shows an example, how the predicates *document_type_states* and *document_relation_type* are specified and how we exploit inheritance.

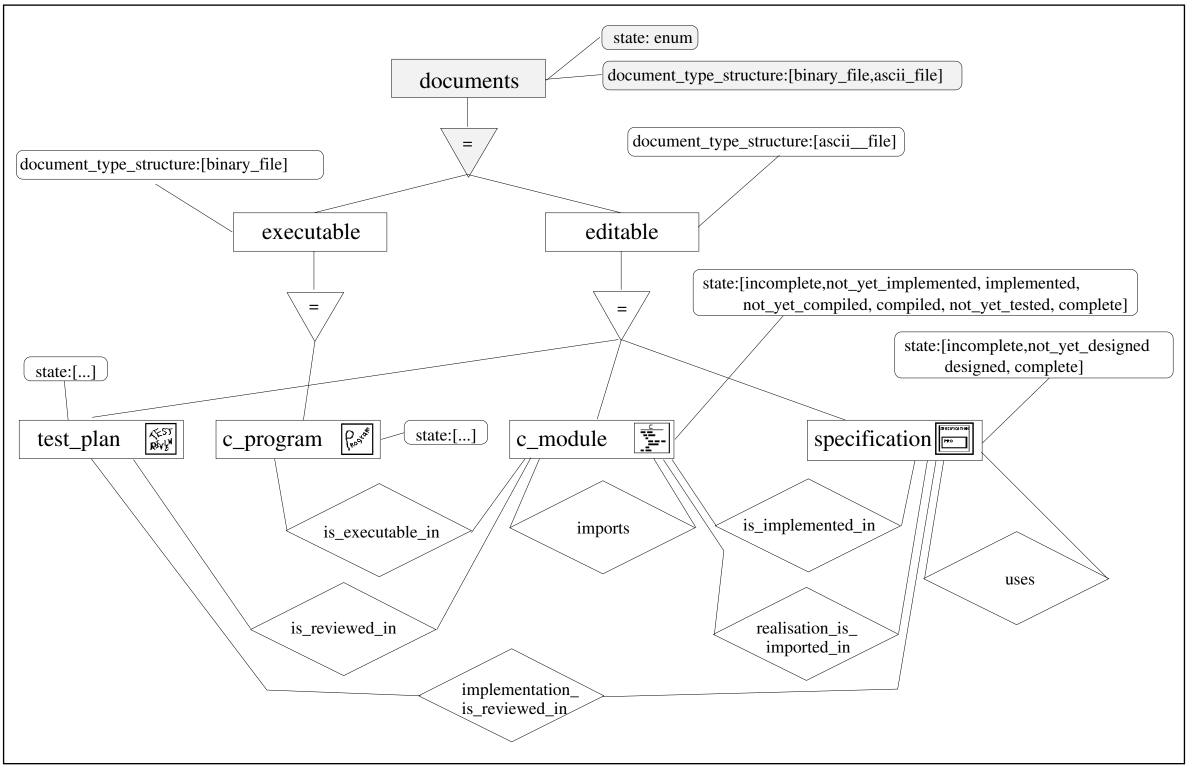

Fig. 14 Document-hierarchy

Predicate *document_type_states* is specified in Fig. 14 by an entity (graphically represented by a rectangle) and an attribute (graphically represented by an oval).

The first fact of the example in Fig. 7 is reflected by the rectangle which contains the name *specification* and has an attribute *state* of type enumeration (with values *[incomplete, not_yet_-designed, designed, complete]*). The second fact is reflected by the rectangle which contains the name *c_module* together with its attribute *state* (with values *[incomplete, not_yet_implemented, implemented, not_yet_compiled, compiled, not_yet_tested, complete]*).

Predicate *document_relation_type* is specified in the diagram by a relationship (graphically represented by a diamond).

The first fact of the example in Fig. 9 is reflected by the diamond which contains the name *is_implemented_in* and is connected by edges to the entities named *specification* and *c_module*. The other six facts are reflected similarly.

Each document type is specified on the Process level by a predefined set of predicates. This is reflected in the EER-diagram by an invariant part (represented with a dark background). The attributes of the invariant part are inherited by all new introduced document types and can be further specified. As variant part, the process engineer can only specify new document and relationship types.

A finite-state machines is attached to each entity which has an attribute of type enumeration.

An example finite-state machine for the attribute *state* of the document type *c_module* is given in Fig. 15.

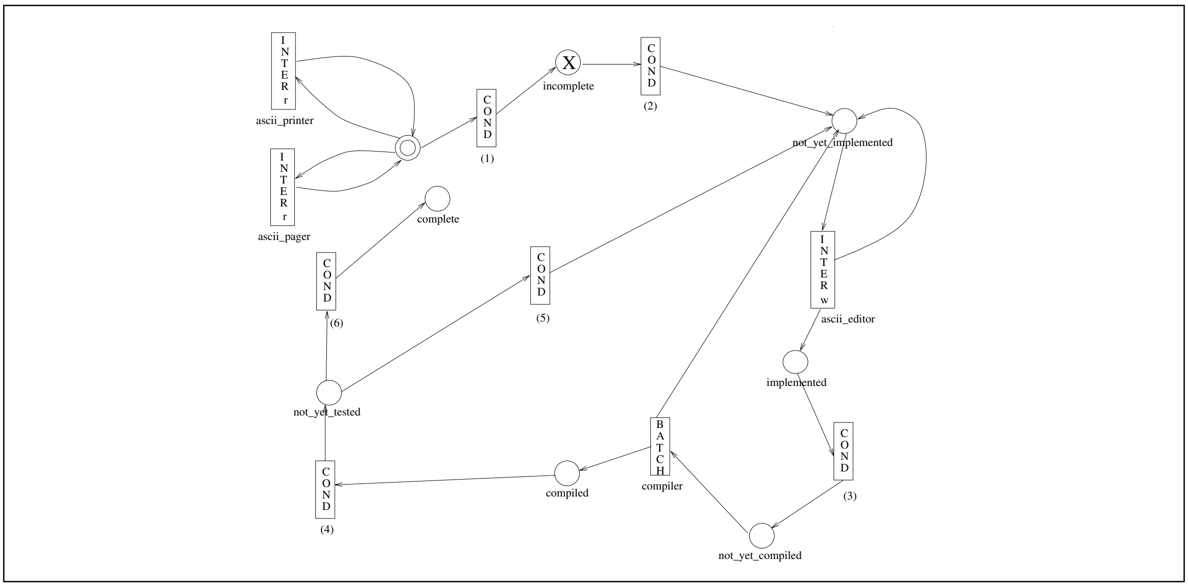

Fig. 15 Finite-state machine of *c_module*

The predicates belonging to the second group specify the conditions under which a transition which is represented as rectangle within the finite-state machine becomes true. Predicates specifying the behaviour which results from the use of interactive tools are represented by rectangles labelled INTER (i.e. *document_type_tools*). Predicates specifying the behaviour which results from the use of a batch tools are represented by rectangles labelled BATCH (i.e. *document_type_tools*). Predicates specifying the behaviour which results from changes on related documents are represented by rectangles labelled COND (i.e. *automation_condition* and *consistency_condition*).

The first fact of the example in Fig. 8 is reflected in the finite-state machine as follows. Parameter *ascii_pager* is given below the rectangle (the second rectangle in the top left corner) and parameter *readable* is given by the character *r* below *INTER*. In the case of *writeable* or *executable* the characters *w* or *x* respectively are used. The precondition allows tool invocation in any state (given by *Any_state*). This is represented in the finite-state machine by the arrow starting from the double-circle. Double-circle is a placeholder for all states of the document type. The empty postcondition *[]* is represented by the reverse arrow pointing to the double-circle. This means the documents' state remains unchanged.

The second fact is reflected similarly. The third fact represents an *ascii_editor* and has as precondition *not_yet_implemented* and as possible postcondition *not_yet_implemented* and *implemented*. The corresponding rectangle is shown in the finite-state machine in the middle of the right side. The precondition is reflected by the source state of the incoming arrow (written below the circle) and the postcondition is reflected by the two target states of the outgoing arrows.

Batch tools are reflected in the finite-state machines similar to interactive tools.

For the specification of the predicates *automation_condition* and *consistency_condition*, formulas in predicate-logic (based on the definition in [Mey88]) are attached to the *COND* transitions in the finite-state machine. The formulas used in Fig. 15 (referred to by numbers in the finite-state machine) are given in Fig. 16.

(1) ∃ x,y ∈ specification • ((is_implemented_in(x,self) ∧ =(x.state, "not_yet_designed")) ∨ (realisation_is_imported_in(y,self) ∧ =(y,state, "not_yet_designed")))

(2) ∀ x,y ∈ specification • ((is_implemented_in(x,self) ∧ =(x.state, "complete")) ∧ (realisation_is_imported_in(y,self) ∧ =(y,state, "complete")))

(3) ∀ x ∈ c_module • (imports(self,x) ∧ =(x.state, "complete"))

(4) ∀ x ∈ test_review • (module_is_reviewed_in(self,x) ∧ =(x.state, "incomplete"))

(5) ∃ x ∈ test_review • (module_is_reviewed_in(self,x) ∧ =(x.state, "test_review_rejected"))

(6) ∀ x ∈ test_review • (module_is_reviewed_in(self,x) ∧ =(x.state, "complete"))

Fig. 16 Formulas attached to conditional-transitions

Predicate *automation_condition* specifies, that a condition must be fulfilled for all documents related to the document of interest by a given relationship. This is reflected in the formula by using the quantor ∀.

The counterpart of the first fact of the example given in Fig. 11 is formula (2) of Fig. 16. This formula belongs to the COND transition between the states *incomplete* and *not_yet_implemented* (see Fig. 15). The *automation_condition* can be mapped to the formula as described now. Parameter *incomplete* is the source state of the incoming arrow connected to the *COND* transition. Parameter *not_yet_implemented* is the target state of the outgoing arrow. The last parameter is a list of conditions which must be fulfilled. The first condition expresses, that all documents related to the *c_module* by an *is_implemented_in* relationship must be in the state *complete*. This condition is reflected in the formula by *is_implemented_in(x,self) ∧ =(x.state, "complete").*

Predicate *consistency_condition* is specified by formulas having the quantor ∃, because the conditions must be fulfilled for only one related document to become true. The fact given in Fig. 10 is reflected by formula (1) in Fig. 16. *Consistency_conditions* are mapped similar to *automation_conditions*.

For a detailed description of the design-language we refer to [Jun93].

5 The Transaction Concept

As already mentioned above the process engine must manage the effects of the parallel work on each developer's working context, i.e. presentation and update of a working context may happen in a distributed fashion. The main problem occurs if a document is accessed by different developers in their working contexts which is a very natural requirement for performing complex tasks. For instance, the documentation of a subsystem might be written cooperatively by all programmers responsible for modules of the subsystem.

Providing the same document in different working contexts at the same time causes concurrency problems like read/write or write/write conflicts on documents. Additional conflicts may arise because the process engine performs the before mentioned batch jobs. To synchronize the concurrent access to documents either by the users or by the process engine a transaction mechanism is required which avoids or solves the conflicts.

A lot of approaches already exist each having its specific advantages and disadvantages (for a detailed discussion c.f. [BK91]). But the open problem still is how to integrate the described transaction mechanisms into a PSEE (or even SEE), and how far can the knowledge about a process be used for deciding which concurrency control policy should be used in which situation?

In Merlin we develop a transaction model which is adopted to the working context based user interaction model. Five different transaction types are supported in Merlin. They can be distinguished in the transactions triggered by the user by performing activities in the working context (user transactions) and the transactions triggered by the process engine by performing batch jobs (process engine transactions).

The user transactions differ first in the number of objects they access and second in their synchronization policies. A user transaction in Merlin either controls the execution of a single activity on a single document (e.g. editing a test plan) or it includes and controls all activities performed within a working context (e.g. editing several source code modules, compiling and linking them). Single activity transactions, in the following called **activity transactions**, could be **s**ynchronized either in a **pessimistic** or **optimistic** way. Transactions controlling all activities within a working context, in the following called **working context transactions**, are only synchronized in a pessimistic way because the possible rollback of those transactions in the optimistic case after having performed many (time consuming) activities is not acceptable.

User transaction types enable to support long transactions (working context transactions) as well as short transactions (activity transactions), the latter ones providing either exclusive and secure document access in the pessimistic case or concurrent (write) access in the optimistic case. The pessimistic case, of course, decreases the number of concurrent accesses whereas the optimistic policy may result in a merge of concurrently performed modifications.

The process engine transactions differ with respect to the semantics of the activities performed under the control of those transactions. Either the process engine follows some automation conditions in the process definition, and therefore changes states of documents and invokes tools (e.g. compilation of a module after the imported modules have been finished), or states and documents are accessed triggered by some consistency conditions (e.g. re-compilation of an already compiled module because an imported module has been modified). Respectively, the process engine transaction types in Merlin are called **automation transactions** or **consistency transactions** (c.f. the separation between automation chains and consistency chains in Marvel [Bar92]).

The type of transaction to be applied is prescribed by the role definition, task definition, the users' access rights, and also by the current project state. For instance, if the programming of a module has to be finished very soon in order to meet a project milestone, the system will not allow to access the module by an optimistic activity transaction anymore (in order not to risk the completion by concurrent accesses).

If still alternative transaction types are applicable for a task (or a set of tasks) the system supports the user in choosing the correct transaction type by providing him information about currently active working contexts, running transactions related to a user's working context, transactions requested by other developers (e.g. requested locks), and changed states of related documents (as an extension of the communication modes proposed in [SZ89]). For instance, a user performing an optimistic transaction then might decide to abort the optimistic transaction and to restart it with pessimistic control or even to contact other developers to discuss the planned modifications and to prepare the necessary merge activity.

The synchronisation algorithm for controlling the concurrent execution of Merlin transactions is based on the definition of priorities for the five transaction types. In case of a conflict the access right is granted to the transaction with the higher priority. For instance a consistency transaction has the highest priority and the pessimistic synchronized user transactions have a higher priority than an automation transaction. A conflict resolution strategy, which distinguishes the access to documents and the access to document states, allows the *Transaction-Manager* (see below) to resolve specific conflict situations not only by aborting one of the two conflicting transactions based on the priority definitions, but also by withdrawing a lock on a document state from one of the conflicting transactions (e.g. a user editing a module then still can continue his editing activity but cannot access and change the module's state).

As mentioned before, the selection of the right user transaction type is based on the process definition and on the project's state. The decision to initiate a transaction of one of the process engine transaction types depends on the semantics of the activities performed by the process engine. On the database level information like role definitions, user access rights, the project's state or the semantics of activities is not known. Therefore the selection of a transaction type, the initiation of a transaction of a selected type and the completion of that transaction is integrated in the Kernel. Fig. 17 shows the integration of working context transactions in the rule *working_context* which was explained in section 3.

```
working_context (Ident, Project, Role,Menu_Activity_List, Document_List, Relation_List):-
        working_context_transaction(Ident, Project,Role),
        start_transaction(Ident, Project,Role, wc_transaction, T_id),
        responsibilities (Ident,Role,Responsibility_List),
        locked_documents(Role,Responsibility_List,Document_List,T_id),
        ...
locked_documents(Role,Responsibility_List,Document_List,T_id):-
        document(Doc,Type),
        document_state(Doc,Doc_State),
        roletype_document_work_on(Role,Type,State,Rdoctypes),
        document_lock(Doc,T_id),
        state_lock(Doc,T_id),
        ...
working_context (Ident, Project, Role,Menu_Activity_List, Document_List, Relation_List):-
        responsibilities (Ident,Role,Responsibility_List),
        documents(Role,Responsibility_List,Document_List,None,None),
        ...
```

Fig. 17 Invocation of a working context transaction in the rule *working_context*

When the existence of a working context is checked, the rule *working_context_transaction* checks whether the working context has to be (or the user wants it to be) controlled by a working context transaction. In that case, a working context transaction is started by invoking the rule *start_transaction* which is a rule of the *TransactionManager* described below. After having identified the user's responsibilities, the rule *locked_documents* checks the documents' states w.r.t. to the role definition, and locks the documents and their states using the rules *document_lock* and *state_lock* of the TransactionManager.

If the working context is not controlled by a working context transaction, the second rule with the name *working_context* is executed which checks the existence of a working context without any transaction control and without locking any document or state. In this case, each user access to a document in the working context is controlled either by an optimistic or pessimistic activity transaction. Activity transactions are integrated in other rules in the *WorkingContext* cluster of the Kernel whereas the process engine transactions are integrated in the *ChangedStates* cluster.

```
working_context_transaction_required (programmer, c_module, c_module, imports).

prohibit_transaction_type_near_deadline (opt_activity_transaction, c_module, 10d).
```

Fig. 18 Defining selection criterion for transaction types by additional facts

The selection of a transaction type for a specific task as discussed above is based on additional process facts and project facts. For instance, fact *working_context_transaction_required* in Fig. 18 defines that a programmer's working context has to be controlled by a working context transaction if the working context enables the programmer to modify at least two modules with an *import* relation. By that it is assured that the modification of related documents is controlled in one transaction.

The second fact, *prohibit_transaction_type_near_deadline,* prohibits to control the access to a *c_module* by an optimistic activity transaction if the deadline to complete the task of programming a *c_module* is reached in 10 days or less. Additional rules in the *WorkingContext* and the *ChangedStates* cluster interpret those facts whenever a transaction is required to control an activity (e.g. the rule *working_context_transaction* in Fig. 17 interprets facts of type *working_-context_transaction_required*). These rules define the semantics of the facts used to influence the selection of a transaction type.

The TransactionManager and LockManager are implemented as rule clusters in the Kernel instead of using a conventional programming language. This achieves tight integration of the transaction control mechanism and the process control mechanism. For instance a transaction might be aborted because a lock request is rejected by the lock manager or because the process engine detects that an activity performed within a transaction cannot be continued because some preconditions have changed and unrecoverable inconsistencies would arise. In both cases, the backtracking mechanism of Merlin is used either by the transaction manager or by the process engine to abort and undo a transaction.

The TransactionManager consists of rules to start transactions, commit transactions, validate optimistic transactions, abort transactions or to request and release locks. Information about all active transactions, i.e. the transactions' identifiers and types, the names and roles of the users performing the transactions etc. is described by facts. The TransactionManager*'s* decisions to reject a lock request and to abort or even to modify a transaction is triggered by the conflict detection which is done by the lock manager.

The LockManager consists of rules to manage all lock information like existing locks on documents and states as well as the compatibility definition of the existing lock modes. If a transaction requests a lock from the TransactionManager, the TransactionManager requests that lock from the LockManager who checks for incompatibility with already existing locks. In case of a conflict the transaction manager is informed about the conflict and decides how to resolve the conflict.

For a detailed description of this transaction concept we refer to [Wol94].

6 Architecture Overview and Implementation Issues

Fig. 19 represents an overview of the Merlin architecture. In this diagram boxes denote modules or subsystems. Arrows denote a use-relationship, i.e. a module or subsystem respectively imports resources provided by a "target" module or subsystem respectively. This use-relationship is very much like the Modula-2 import-relationship or the EIFFEL client-server relationship.

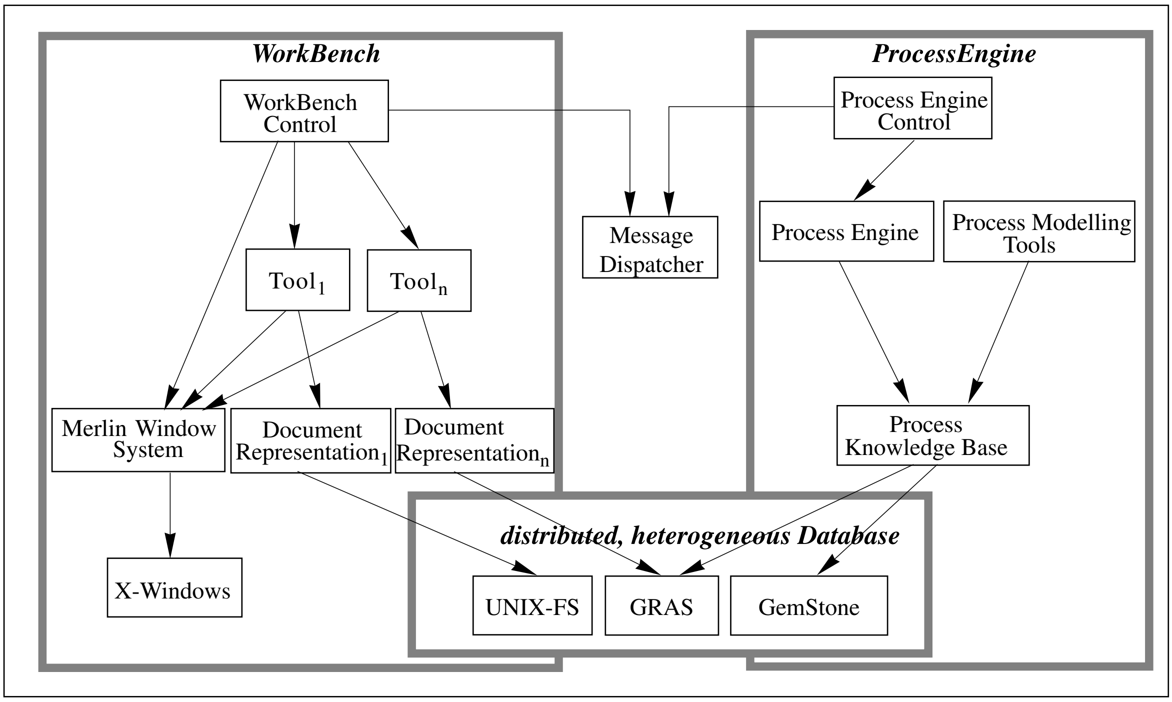

Fig. 19 Merlin architecture

The architecture is based on a distribution model which assumes several WorkBenches where each one is associated with a Process Engine. The diagram in Fig. 19 just gives the overview about one such pair. WorkBench and Process Engine communicate with each other via the Message Dispatcher's facilities based on TCP/IP. The subsystem Process Engine Control transmits working context updates based on the Process Engines's computation (see below) to the WorkBench. Subsystem WorkBench Control is in charge of initiating the corresponding screen display and transmitting the updated state information, which results from the execution of tools, back to the Process Engine.

Each pair WorkBench/Process Engine in a Merlin environment communicates also via a message dispatcher with each other (not shown in the diagram). It is worthwhile to note here that the process data which is stored in the underlying data base is not (yet) stored in a distributed fashion, i.e. only the process interpreter exists in several instances but not the facts of a particular process and project description.

We currently investigate the use of a message server, namely the HP Softbench, to improve the services provided by the current message dispatcher. This would result in a architecture which is very similar to the SPADE-1 architecture as described in [BFG+93]. In this sense we suggest here a possible break-down of their course-grained "three component"-architecture which consists of the user environment (corresponding to our WorkBench), the process engine environment (corresponding to our Process Engine) and the filter (corresponding to our WorkBench Control and Process Engine Control respectively). The further difference is that SPADE-1 uses the DEC FUSE message server instead of HP Softbench. A more thorough investigation and comparison of those approaches is subject to further research.

The Process Engine-subsystem is in charge of storing, retrieving, and interpreting the specification of a development process as explained in section 3. In particular, it implements the transaction synchronisation as mentioned in section 5.

Process Modelling Tools provide the edit and analysis facilities to specify a process by using the graphical modeling language as explained in section 4.

The underlying data base management system used to store documents and process information is heterogeneous. Some documents are stored as UNIX-files, because the corresponding tools are UNIX-tools like vi, cc, etc. Some documents are manipulated by more sophisticated tools and their contents is stored in the form of an abstract syntax graph. To store a massive amount of fine-grained data we exploit the features of the non-standard data base system GRAS [LS88]. That paper explains in detail why GRAS is a highly suitable storage mechanism for abstract syntax graphs.

Module Process Knowledge Base just provides the special operations to access facts and rules based on the schema definition as given in [PS92]. This schema definition is basically an abstract syntax graph definition of a PROLOG-like program and thus a storage mechanism like GRAS is needed to give adequate support for storing this fine-grained data. In particular, the tools supporting the modeling of processes are all sophisticated syntax-directed tools and thus, as mentioned above, need support like GRAS offers.

Process data, namely the facts and rules of the Kernel, Process and Project levels are stored partly in GRAS (the rules) and partly in the fully object-oriented data base system (OODBMS) GEMSTONE (the facts). Facts are stored in GEMSTONE rather than in GRAS, because the transaction mechanism as explained in the last section requires to lock facts. GRAS only provides locking of complete graphs whereas facts are small subgraphs which cannot be locked separately in GRAS. In an OODBMS like GEMSTONE this granularity problem does not exist, as locking can be performed on any level of granularity, i.e. on small objects representing facts as well as on larger granular like a collection of objects representing a complete abstract syntax graph.

7 Current and Further Work

The problem with at least the currently available version of GRAS with respect to multi-user support and transaction management in general as well as the good availability and performance of OODBMSs in that respect kicked off a new research direction in the Merlin project, namely the investigation of OODBMSs as dedicated software engineering platforms. First results indicate that this is a very promising way to go but that existing OODBMSs still need some improvement to fulfil all software engineering requirements [ESW93], [EKS93].

A further major piece of work is to integrate a configuration management component in Merlin, i.e. to improve Merlin‘s support for cooperation by enabling to specify a particular version and corresponding configuration model within the Merlin modeling language. This again means to extend the Kernel level by rules which fix a basic VM/CM model and then provide a number of predefined predicates which are used to define a particular process or project-specific VM/CM model [Sac93].

Finally, a major effort is geared towards improving the architecture with respect to distribution as indicated in [PW93]. The main idea is to have, in addition to distributed process interpreters, also partly distributed process data. This, of course requires much more sophisticated synchronisation policies which keep the process data in a consistent state. The communication which is needed to keep this consistency should be supported by a sophisticated message server which is also under investigation.

An important part of the Merlin research project is technology transfer. Thus, some of the mentioned concepts are or have already been used in the commercial production of CASE tools together with our industrial partners. The transfered concepts includes the exploitation of GRAS and the idea of an explicit definition of a software process as the major part of tools which support multi-user cooperation.

Acknowledgements

We are indebted to the other members of the Merlin project, namely Wolfgang Emmerich, Olaf Neumann and Sabine Sachweh who contributed a lot to the state of the project as described here through a lot of exploitive and fruitful discussions.

We gratefully acknowledge the implementation work of a number of master students which only brought the current version of the prototype into existence. Those theses include the work of Frank Buddrus, Oliver Gritsch, Jens Jahnke, Thomas Leppek, Rainer Kruschinski, Klaus Marquardt and Michael Nippel.

References

[BK91] N.S. Barghouti, G. Kaiser, "Concurrency Control in Advanced Database Applications", *ACM Computing Surveys*, Vol. 23, Nr. 3, 1991.

[BK92] N.S. Barghouti, G. Kaiser, “Scaling Up Rule-based Software Development Environments”, in: *International Journal of Software Engineering and Knowledge Engineering* (March 1992), Vol. 2, No. 1, pp. 59-78.

[Bar92] N.S. Barghouti, "Concurrency Control in Rule-Based Software Development Environments*", Ph.D. Thesis*, Columbia University, Technical Report CUCS-001-92, New York, USA, 1992.

[BFG+93] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, Luigi Lavazza, "An Overview of the SPADE Project", this volume.

[EGH+93] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, H.-D. Ehrich, "Conceptual modelling of database applications using an extended ER model", in: *Data & Knowledge Engineering* (1992/1993), Vol. 9, No. 2, pp. 157-204.

[EKS93] W. Emmerich, P. Kroha, W. Schäfer, "Object-oriented Database Management Systems for Construction of CASE Environments", in: *Proc. of the 4th Int. Conf. on Database and Expert Systems Application*, Prague, Czech Republic, 1993 (to appear), Springer LNCS.

[ESW93] W. Emmerich, W. Schäfer, J. Welsh, "Databases for Software Engineering Environments --- The Goal has not yet been attained", in: *Proc. of the 4th European Software Engineering Conference*, Garmisch-Partenkirchen, Germany, 1993 (to appear), Springer LNCS.

[GP93] P.K. Garg, T.Q. Pham, "Process Modeling in Matisse, A Team Programming Environment", in: *Proc. of the 8th Int. Software Process Workshop*, Dagstuhl, Germany, 1993.

[HJP+89] H. Hünnekens, G. Junkermann, B. Peuschel, W. Schäfer, K.J. Vagts, "A Step Towards Knowledge-based Software Process Modeling" in: Madhavji N., Schäfer W., Weber H. (ed.): *Proceedings of the First Conference on System Development Environments and Factories (SDE&F 1)*, Pitman Publishing, London, 1990.

[Jun93] G. Junkermann, "A design method to improve process-programming in Merlin" (submitted for publication).

[KFF+91] M.I. Kellner, P.H. Feiler, A. Finkelstein, F. Katayama, *et. al.*, ISPW6 Software Process Example, in: M. Dowson (ed.), *Proc. of the 1st Int. Conf. on the Software Process* (IEEE Press, Oct. 1991) pp. 176-186.

[KFP88] Gail E.Kaiser, Peter H. Feiler, Steven S. Popovich, "Intelligent Assistance for Software Development and Maintenance", *IEEE Software*, 1988, pp. 40-49.

[LS88] C. Lewerentz, A. Schürr, "GRAS - a Management System for Graph-like Documents", in: Beeri, Schmidt, and Dayal (eds.), *Proc. of the 3rd Conf. on Data and Knowledge Bases* (Morgan Kaufmann, 1988) pp. 19-31.

[Mey88] B. Meyer, "Introduction to the Theory of Programming Languages", Prentice Hall, 1988.

[MS91] Madhavji N.H., Schäfer W., "Prism - Methodology and Process-Oriented Environment", *IEEE Transactions on Software Engineering*, Vol. 17, No. 12, December 1991, pp. 1270-1283.

[PS92] B. Peuschel, W. Schäfer, "Concepts and Implementation of a Rule-based Process Engine, *Proc. of the 14th Int. Conf. on Software Engineering*, Melbourne, May 1992, pp. 257-276.

[PSL91] Programming Systems Laboratory, “Marvel 3.0 User’s Manual”, *Technical report CUCS-033-91*, Columbia University Department of Computer Science, October 1991.

[PSW92] B. Peuschel, W. Schäfer, S. Wolf, "A Knowledge-Based Software Development Environment", in: *International Journal of Software Engineering and Knowledge Engineering* (March 1992), Vol. 2, No. 1, pp. 79-106.

[PW93] B. Peuschel, S. Wolf, "Architectural support for distributed process centered software development environments", in: *Proc. of the 8th Int. Software Process Workshop*, IEEE Press, Dagstuhl, Germany, 1993.

[Sac93] Sabine Sachweh, "A Proposal for Configuration- and Version-Management in Merlin", Technical report, University of Dortmund, Department of Computer Science, Software-Technology (to appear).

[SZ89] A.H. Skarra, S.B. Zdonik, "Concurrency Control and Object Oriented Databases", in: W. Kim, F.H. Lochovsky (ed.) *Object Oriented Concepts, Databases and Applications*, pp. 395-421, ACM Press, New York, 1989.

[Wol94] S. Wolf, "Supporting Cooperation in Process-Centered Software Development Environments (in german)", (in preparation: Ph.D. thesis, University of Dortmund, Dep. of Computer Science, 1994).