A Prolog-based Semantics of a Dedicated Process Design Language *

Gerald Junkermann
University of Dortmund, Informatik X, D-44221 Dortmund, Germany

June 27, 1994

1 Introduction

Like for the development of industrial products, the process underlying the development of software should be fixed in advance in order to achieve high productivity and high quality of the process and the resulting product. Currently, this process is specified through company internal development guidelines. These usually voluminous and informal documents are written in a "natural" language which make them difficult to use, because they allow different interpretations. To avoid different interpretations a language with a precise semantic is needed.

The use of formal languages and tools for the definition and enactment [FH93] of process programs [Ost87] enable a unequivocal interpretation of the specified processes. The process definitions can now be interpreted by machines. The consequence is, that there is no more a recommendation through guidelines then a guiding of software development through the definition and enactment of processes.

A further advantage is, that processes can now be analyzed and validated [Lon93]. Through validation, deficiencies and necessary process improvements can be detected. After change, the improvements will become an integral part of the process and will be used immediately.

Recently, several process modeling languages have been developed. Most of them are multi-paradigm approaches, as proposed in [DGS89], centered around one main paradigm like rules (e.g. Marvel [EFP88], Merlin [JPSW94], Oikos [AM92]), imperative programming languages (e.g. APPL/A [HSO90], IPSE 2.5 [War89]) or petri-nets (e.g. Melmac [Gru91], ProcessWeaver [Fer93], SLANG [BaAM91]).

Most of the above listed process programming languages are on the same level of abstraction as "traditional" programming languages and, therefore difficult to understand and unsuitable to explain the process defined. When writing a process program, the process engineer has to consider a lot of technical features of the used programming language, which hampers the process definition itself.

What is needed is a high-level process design language which offers a more intuitive representa-

*This work has been supported by the Provincial Ministry for Research (MWF) of the state of Northrhine Westphalia

tion of the process. A promising starting point is to apply the concepts of already known and accepted graphical languages, which have been used successfully for the design of databases and complex reactive systems.

This paper presents ESCAPE [1], a graphical design-language for the design of process-programs. The design-language will become part of the Merlin Process-centered Software Development Environment (PSDE) prototype [JPSW94].

In section 2 the aspects to be considered when modeling processes are worked out by an examination of the problem area of PSDEs. The section ends with a proposal for a high level process design language. The proposed design language is based on an EER-model extended by Statecharts. The EER-model and Statecharts are introduced in section 3 and an example process, is given. The example process shows, that application specific restrictions can be made on the introduced language to ease its use. Examples for necessary restrictions on the language as well as context-sensitive conditions and semantic conditions to be kept are introduced in section 4. The graphical design language is not enactable and only the front-end to the process-engineer. Therefore, it is mapped to an enactable language. The chosen enactable language is the PROLOG-like process programming language of the Merlin-prototype. In section 5 it is shown, how to map the graphical design language to the enactable language. In section 6 related systems are introduced and it is shown how they differ from ESCAPE. In section 7, a conclusion with a summary of results as well as further plans to extend the prototype functionality is given.

2 Problem definition

Documents, relationships between documents, properties of documents, tools which can be applied to documents and roles, are the ingredients (objects) which have to be considered when specifying a software development process.

Documents are manipulated in one of the phases of the underlying software life cycle (e.g. waterfall model, spiral model). For every phase, one or more documents can be determined. A *specification* document is, for example, part of the design phase, whereas the code of a module (e.g. *c_module*) is part of the implementation phase.

The documents which are part of the software production process are related to each other by **relationships** (e.g. a *uses* relationship between two *c-modules*, an *is_implemented_in* relationship between a *specification* and its *c_module*). Relationships are needed, to propagate changes to related documents or to get information which is needed for the development of the document of concern (e.g. (1) if an operation name is changed, this change must be propagated to all related documents using this operation, (2) a module can only be compiled if all imported modules have been compiled successfully).

Both, documents and relationships can have **properties**. Examples are the behaviour of a document or the EBNF correspondence between two documents. The EBNF correspondence is used to relate the contents of one document, or only a part of it, to the contents of the related document (e.g. the operation name in the specification and implementation of a module must be the same).

Tools correspond to activities and are used to access and manipulate the document contents

[1] **E**ER-models and **S**tatecharts **C**ombined for **A**dvanced **P**rocess **E**ngineering

or its management information (e.g. a time stamp). Tools can be distinguished into those which require user interaction (e.g. editor, debugger) and those which can be started and no further user interaction is required (e.g. compiler). In the following, tools of the first category are named **interactive tools**, and those of the second category **batch tools**.

In addition to the technical aspect, there is also a managerial aspect. If a large number of developers are involved in a project, the whole work is divided into smaller parts. Each part is assigned to one of the developers who is now responsible for it. To support specialization and by that increase productivity, these parts are centered around activities which are highly related (e.g. project management, design, test). A group of activities which are highly related is named **role**. One or more roles (e.g. manager, programmer, tester) can be assigned to a developer to define his task and to fix the work which can and must be performed by him (e.g. a developer having as role programmer is responsible for writing the code but has no responsibilities for managing the development activities).

From the structural perspective the above listed ingredients are sufficient for the specification of a software development process. What has not been considered is, that the development activities must be perfomed in a certain order to get the desired output. It is for example not possible to test a program before it has been written. Furthermore, development activities may be perfomed in parallel and have to be coordinated. To solve this problem, synchronization is needed. Through synchronization, the work of several developers working cooperatively on a common project is coordinated and the desired output is guaranteed. Synchronization can only be realized, if the development state of a document is known and if the effects of modifications are propagated to all objects affected by this modification.

The development state of a document is fixed by the **document state**. For each development state, one unique value is defined. The set of all values valid for the document is the domain of the document state.

An example for the development states of a *c_module* is: (1) the specification of the module is not complete or it is currently modified, implementation can not start or an old implementation has to be adapted to new requirements, (2) the specification is complete and the implementation of the module can start, (3) the module is implemented as specified and all imported modules are compiled successfully, (4) the compilation is successful. The corresponding state values are: (1) *incomplete*, (2) *not_yet_implemented*, (3) *implemented*, (4) *compiled.*

A change of state for a document can have two reasons: (1) a document is modified by using a tool and a new state is set by the user or system, or (2) the effect of a change of state for the document of concern must be propagated to related documents.

Propagation of changes concern the aspect that a change of state for one document might cause a change of state for possibly an arbitrary number of related documents. If, for example, the state of a *specification* changes from *complete* to *incomplete*, the state of the related *implementation* must be changed too (to *not_yet_implemented*). The change of state for the *implementation* is necessary to guarantee, that the changes made for the *specification* will also be considered for the *implementation.* The change of state for the *implementation* may also cause further state changes to other related documents which have to be handled in the same way.

If a document is modified by a tool it has to be guaranteed, that the tool can only be applied to a document, if it makes sense in the current state of development, i.e. some **preconditions** must be fulfilled. A precondition for a compilation is for example, that the developer has choosen the role programmer and that coding has been finished. Analogously, the result of tool application should be important for further activities to be performed, i.e. a **postcondition** must be fixed. An example for a postcondition is, that the document is ready for test if the compilation was successful.

Summarizing, the development of a process-design language is highly influenced by the following aspects to be considered when modeling processes.

- Which documents are manipulated during software development?
- Which are the relationships between documents?
- Which are the properties of a document/relationship?
- Which tools/activities can be invoked on a document?
- Which roles are supported?
- What are the states a document may be in?
- How does the invokation of a tool influence the state of a document?
- How does a change of state for one document influence the state of related documents?
- What are the preconditions and postconditions for tool invokation?

The above given problem definition has shown, that two different perspectives have to be considered for process-design. The structural perspective and the behavioural perspective.

The structural perspective includes the specification of document types, relationship types, tools that can be applied to documents of a specific type, states a document may be in, and the supported roles.

The behavioural perspective captures those aspects of the process that are concerned with change. Every activity performed on one document may change the state of the document of concern or related documents.

As proposed in [EG94], the process design language should support an explicit modeling of both perspectives to decrease the complexity of the modeling activity and to improve the changeability of software process specifications.

The similarities between the above given requirements and those for information modeling are obvious. Therefore, the three information modeling approaches Object Modeling Technique (OMT) [RBP+91], the Booch method [Boo94], and the method supported by STATEMATE [2] [HLN+90] have been examined to take advantage of knowledge already available.

Due to the fact that the object model is most fundamental, because it is necessary to describe what is changing before describing when and how it is changed it becomes the central part of the specification. It captures the structural perspective of the process. For modeling that part, the EER-model, as supported by OMT and the Booch method, is a suitable expedient. Using a specification technique which is build around object classes rather than around functionality, more closely corresponds to the real world and therefore modeling becomes easier.

[2] STATEMATE is a trademark of i-Logix, Inc., Burlington, MA

For modeling the behavioural perspective, the EER-model must be extended. In STATEMATE, which is an environment for the development of reactive systems, the activity charts and module charts are used to specify the structural perspective. These charts are combined with Statecharts to specify the behavioural perspective. Similarly, the above proposed EER-model is extended by Statecharts [Har87]. Statecharts are formally well defined and support the specification of states, transitions between states, and, through orthogonality, the specification of synchronization.

3 Process Modeling using an EER-model

In the previous section an EER-model was proposed as formalism for the specification of software processes. In this section the suitability of this approach is shown. The used EER-models and Statecharts are briefly introduced and an example process design is given.

3.1 Concepts of the supported EER-model

The supported EER-model is based on the definitions and graphical representations introduced by Rumbaugh et al. in [RBP+91].

The syntax of the used EER-models can be explained by the example given in figure 1. The

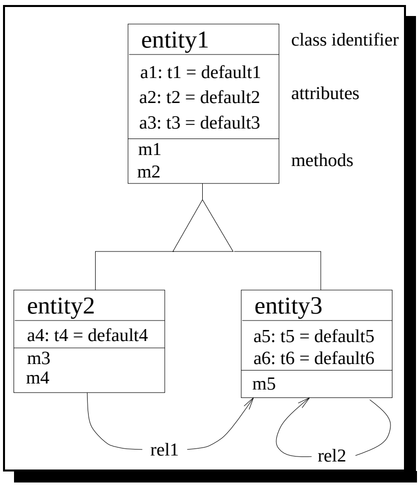

Figure 1: Example EER-diagram

specification of attributes and methods is shown for each of the three specified classes, named *entity1*, *entity2* and *entity3*. The number of attributes and methods is unlimited. Classes are graphically represented by a box separated into three parts. The first part contains the class identifier, the second the attributes and the third the methods. A class being refined is called a superclass and the refined version is called a subclass. Non-leaf nodes are called abstract classes and leaf nodes are called concrete classes.

For class *entity1* the attributes *a1*, *a2*, and *a3* are specified. For each of them, a type and default value are given. In the case of *a1*, the type is *t1* and the default value is *default1*. The methods specified for that class are *m1* and *m2*.

Two relationship types are specified, named *rel1* and *rel2*. Relationships are allowed between two concrete classes, between an abstract class and a concrete class, and between two abstract classes. Relationship types are graphically represented by an arrow having the relationship identifier attached to it.

For sharing similarities among classes they are organized in an inheritance hierarchy. All properties, methods and relationships are inherited.

3.2 Concepts of the supported Statecharts

The supported Statecharts are based on the definitions and graphical representations introduced by Harel in [Har87], [Har88]. Statecharts are based on state-transition-diagrams enhanced by orthogonality, depth and a broadcast-communication mechanism. They offer as modeling concepts *states* and *labeled transitions*. Labels are an *event/action* pair used to specify the dynamic behaviour. The supported broadcast mechanism guarantees, that all external and internal events are received by every part of the system, independent of the sender.

The syntax of Statecharts can be explained by the example Statechart given in figure 2. Orthogonality is shown in state Z, which is composed out of the states G, H and I. The

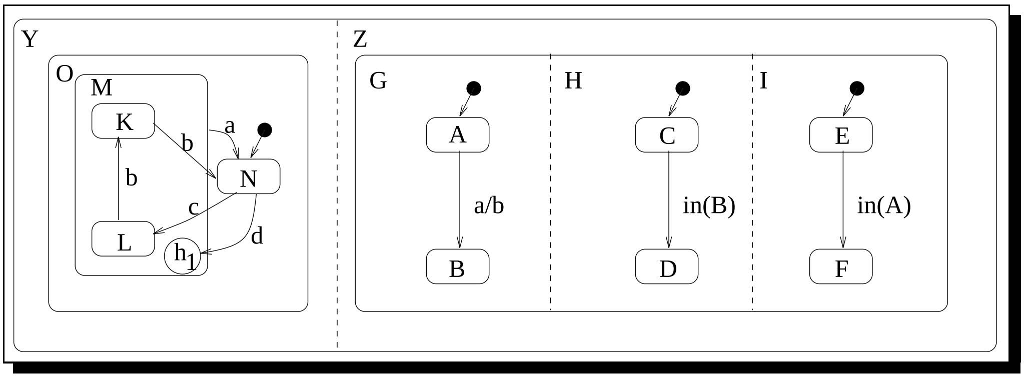

Figure 2: Example of a Statechart

notation is, that the box is split into components using dashed lines. When in being Z, the system is in all states of its components (e.g. $\{A,\ C,\ E\}$). Clustering and refinement is supported as shown for state M. M is used to cluster K and L. When clustering states, the system can only be in one of the specified components which are part of the cluster (i.e. for the given example K or L). If the event a occurs, the new state will be N, if one of the states K or L have been pre-states. If event b occurs, the new state will be N, if the pre-state was K, or it will be K, if the pre-state was L. A default-state, indicated by an arrow with a filled dot (e.g. A for the state G) and history, graphically represented by a circle, are also supported. The history h_1 in state M indicates, that the system will return to the state it has been in before leaving M. An example for a condition is *in(A)* as shown for the transition between states E and F. An example for an event/action pair is a/b as shown for the transition between states A and B. If event a occurs, the transition can fire and event b is send.

Through orthogonality, the Statecharts support the specification of concurrency. If, in the example given in figure 2, the system is in the state $\{K,\ A,\ C,\ E\}$ and the event a occurs, the new state will be $\{N,\ B,\ C,\ F\}$. The two components O and G have changed state simultaneously.

For further details concerning syntax and semantic of Statecharts the reader is refered to

[HPSS87].

3.3 Process modeling example

The aim of this section is to demonstrate by an example, that the structural as well as behavioural perspective of a process can be modeled using EER-diagrams and Statecharts.

To structure the explanation, the whole process is introduced in a step-wise manner. The steps give a first idea of the different steps to be performed to get a process design and reflect the aspects to be considered when specifying processes (see section 2). Although the steps are explained separately, incremental development, iteration, and refinement will be usual.

As mentioned in section 2, the specification should be build around object classes. Therefore, the first three steps are dedicated to the specification of the class hierarchy.

Step 1: Examine the documents manipulated in the different phases of the software life cycle and classify them. After identifying all document classes, its properties and the tools which can be applied to documents of this class are fixed. Properties are defined by attributes and tools correspond to methods.

Analogous to the separation of tools into interactive tools and batch tools, methods are also separated into interactive methods and batch methods. This separation on the specification level is needed to distinguish between those tools which must be invoked by the user (e.g. an editor) and those which can automatically be invoked by the process engine (e.g. a compiler). If the same tool can be applied interactively as well as automatically (e.g. a compiler could for example be used for a syntax check and for the final compilation), it has to be specified twice with different names, because the semantic when using the tool is different in both cases. Graphically, interactive methods and batch methods are separated by a dashed line.

Step 2: After the specification of attributes and methods, the similarities among document classes are identified. Similarities can for example be the same document structure or the same methods.

The final result is a class hierarchy where the concrete classes represent those document classes which are part of the final product to be developed (e.g. *source code*, *documentation*) as well as the document classes produced to ease development and to improve quality and performance (e.g. *test plan*, *project plan*). Figure 3 shows a sample class hierarchy. For the abstract classes *executable* and *editable* the specification of the attribute *document_type_structure* is given. The attribute is inherited from class *documents*, its type is an enumeration and for both classes, a value is selected. In the final shaping, for each class the attribute *document_type_structure* is specified.

The interactive methods *ascii_pager*, *ascii_printer*, and *ascii_editor* are specified for the abstract class *editable*, because they can be invoked on every editable document. The concrete class *c_module*, for example, inherits these methods and is enhanced by the batch method *c_compiler*.

Step 3: Identify the relationship-types which exist between instances of concrete document classes.

An example for a relationship type is *is_implemented_in*. The relationship type relates the *specification* and *implementation* of a module. It is used to perform changes to the

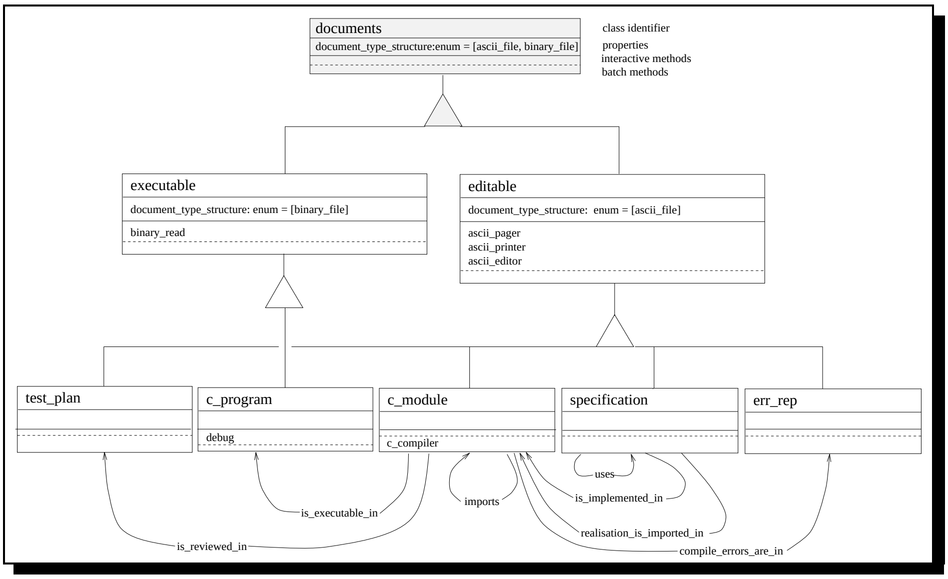

Figure 3: Document classes and their relationship types

implementation if the specification has been changed or vice versa.

Up to this point, the structural perspective of the process was specified. For the specification of the behavioural perspective, for each concrete class a Statechart must be specified.

For the specification of a Statechart, steps 4 to 6 have to be performed. The given examples are part of the Statechart used to specify the behaviour of class *c_module*.

Step 4: Identify the states a document may be in. One of these states has to be selected as default-state. The default-state is assigned to document instances after creation.

For class *c_module*, the states *incomplete*, *not_yet_implemented*, *implemented*, *not_yet_compiled*, *compiled*, *compiled_with_errors*, *not_yet_tested*, and *complete* are specified. As default state, *incomplete* is choosen.

Step 5: For all methods, the preconditions and postconditions must be specified. This is performed through the specification of labels having an empty event and as action a method identifier. The source and target state of the transition correspond to the pre- and post-state for method invokation.

The labels shown in figure 4 are interpreted as follows. The two methods *ascii_printer*

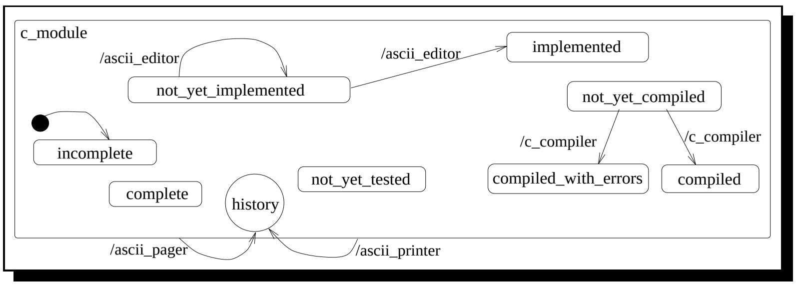

Figure 4: Specification of tool invocation

and *ascii_pager* can be applied to a *c_module* any time, i.e. each state is a pre-state for method invokation. This makes sense, because the user must be able to print or visualize the contents of the document whenever needed. Since there is no effect on the development state of the document itself, the post-state must be same as before. On the specification level this is expressed by a transition having as source a cluster of all states a document may be in and as target the history-state.

For the two methods *ascii_editor* and *c_compiler* only the pre-states can be determined. As post-states, two states are possible. Which of the possible states will be the post-state can not be decided in advance (e.g. the result when executing a compiler is for example a successful (post-state *compiled*) or erroneous (post-state *compiled_with_errors*) compilation). This non-determinism within the Statechart exactly reflects the real situation.

Step 6: As already explained in section 2, a change of state for the document of concern might be triggered by a change of state of a related document. Conditions for such changes are specified by a condition as label.

Two kinds of conditions are distinguished: (a) those which ckeck if <u>at least one</u> related document is in a certain state, and (b) those which check if <u>all</u> related documents are in a certain state.

Figure 5 shows the Statechart of figure 4 extended by transitions having a condition as label. The interpretation of these transitions is explained by the following two examples.

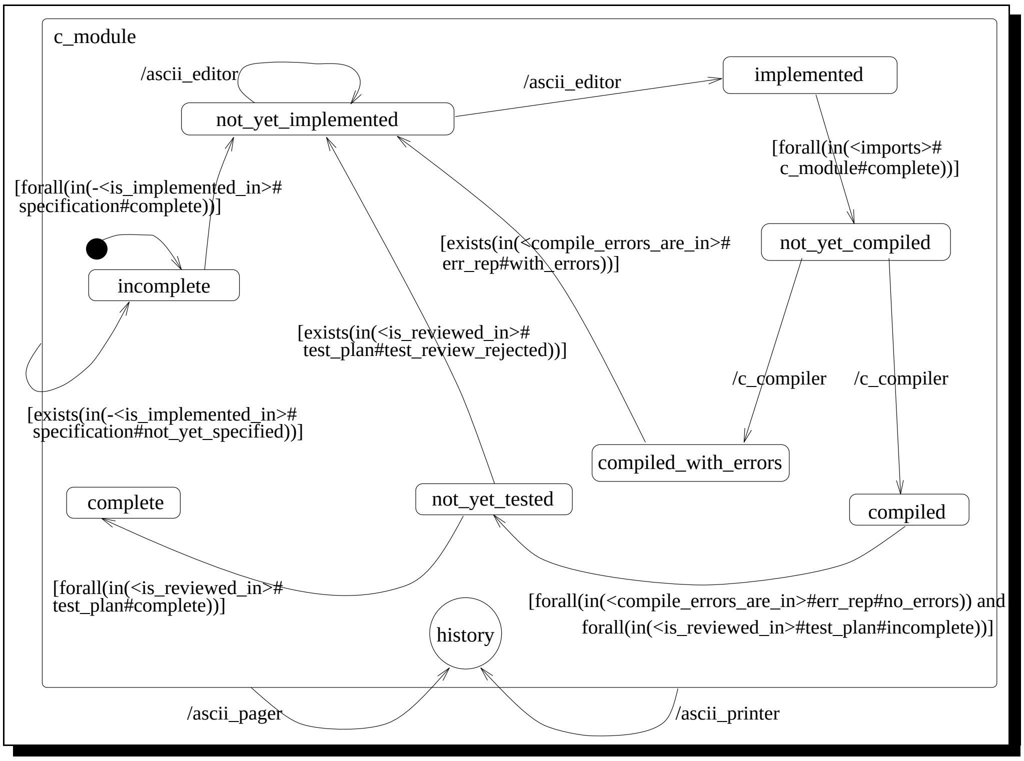

Figure 5: Specification of inter-document dependencies

- The transition between the state *c_module* and the state *incomplete* specifies, that the state of a c-module must be changed to *incomplete*, if the related specification was changed. The label is interpreted as follows: the document of concern can change from every state to *incomplete*, if <u>at least one</u> document of type *specification*, related to the document by an *is_implemented_in* relationship, is in the state *not_yet_specified*.

- The transition between the states *implemented* and *not_yet_compiled* specifies, that a c-module can only be compiled, if all imported modules have been compiled successfully. The label is interpreted as follows: the document of concern can change its state, if all documents of type *c_module*, related to the document by an *imports* relationship, are in the state *complete*.

The following scenario (see figure 6) explains the specification of synchronization in more detail. Consider the following five documents and their states: (1) *Spec1* in state *not_yet_specified*, (2) *Spec2* and *Spec3* in state *complete*, (3) *Mod* in state *incomplete*, and *Arch* in state *complete*. *Spec1*, *Spec2* and *Spec3* are of class *specification*, *Mod* is of class *c_module*, and *Arch* of class *architecture*. *Arch* is related to *Spec1* by a *creates* relationship, *Spec1* is related to *Spec2* and *Spec3* by a *uses* relationship, and *Spec1* is related to *Mod* by an *is_implemented_in* relationship (the upper left part of figure 6 shows the corresponding EER-diagram; the upper right part shows the instances and how they are related).

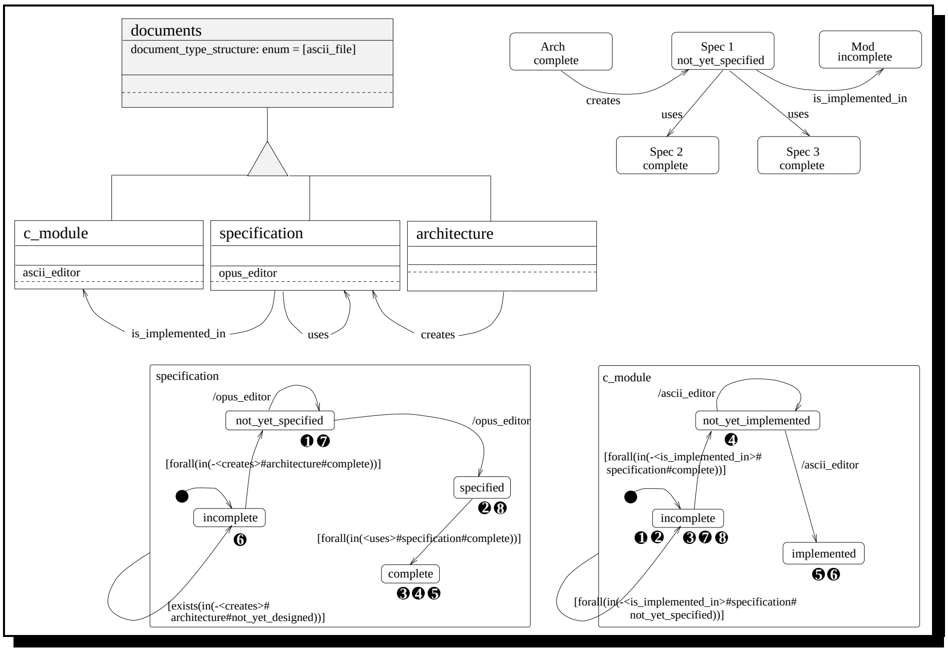

Figure 6: Synchronization

The Statecharts in figure 6 show a sequence of snapshots to explain the dynamic behaviour of the documents *Spec 1* and *Mod*. The state of the documents at a certain moment in time is given in the figure by a pair of black dots with a white number.

In the following, the sequence of state changes is explained (the number refers to the dot in the Statecharts):

1. initial state
2. *Spec1* is modified using the *opus_editor*, the developer selects the state *specified* to fix the development progress
3. the *specifications* used by *Spec1* are in the state *complete*, the state of *Spec1* changes to complete
4. *Mod* is specified and can now be implemented, the state of *Mod* changes to *not_yet_implemented*

5. *Mod* is modified using the *ascii_editor*, the developer selects the state *implemented* to fix the development progress
6. *Arch* is currently be modified, these modifications must be considered for *Spec1*, the state of *Spec1* is set back to *incomplete*
7. *Arch* is now in state *complete* because the modifications are completed, *Spec1* is set to *not_yet_specified*
8. *Spec1* is modified using the *opus_editor*, the developer selects the state *specified* to fix the development progress

After the specification of document classes, relationship types and the behaviour of class instances, the roles are specified.

Step 7: Specify the role-hierarchy through identifying the supported roles (see figure 7). The

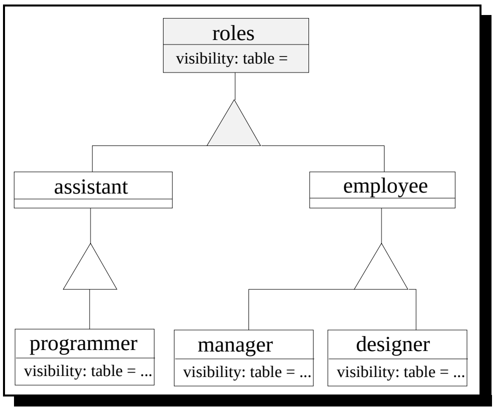

Figure 7: Role hierarchy

attribute *visibility* is used to specify a role dependent view on the total amount of document instances (see step 8).

Step 8: Specify the attribute *visibility* for each role. Attribute *visibility* is of type *table* and defines the document types which are visible under a certain role.

Tables are composed out of three columns and an arbitrary number of rows. They contain the document classes the developer can modify under this role (column *working document type*, shaded row) and the related document classes which are visible to perform the work (column *related document type*, white rows) (access rights are preassumed). The column *state* is used to specify the state a document of the given type must be in to be visualized.

Figure 8 shows an example table for the role *programmer*. A *programmer* can modify

working document type	related document type	state
c_module		not_yet_implemented
	specification	complete
	test_plan	review_rejected
c_program		not_yet_executed
	c_module	not_yet_programmed

Figure 8: Specification of the role *programmer*

instances of the document classes *c_module* and *c_program*. The documents become visible in the programmers working context, if they are in the states given in the last column of the table (e.g. for *c_module* this is the state *not_yet_implemented*). To perform his work, the programmer needs further information available in documents of the related classes. These classes are specified in a similar way. For the programmer, these are for example document instances of class *specification* and *test_plan* if in the state *complete* and *review_rejected*, respectively. Dependent on the states a document is in, the methods which can be applied to this document can be derived out of the specification given in the Statecharts.

4 A Dedicated Process Design Language

The example given in section 3.3 has shown, that only a limited part of those concepts offered by EER-models and Statecharts is needed for the design of process models.

In this section, application specific adaptions to the before introduced language are made. Based on these adaptions, context-sensitive conditions are introduced. Context-sensitive conditions guarantee, that only syntactically correct process models can be specified. Since not every syntactically correct process model makes sense, those semantic conditions which can be checked statically are given. Those semantic conditions which have to be checked dynamically can only be identified by a simulation.

In the subsequent sections, adaptions and conditions are only introduced by examples due to the limited space available for this paper. Simulation is not discussed within this paper.

4.1 Application specific adaptions

This subchapter includes examples for application specific adaptions. For EER-models these are:

Since a process model only deals with objects of type document and role, the number of hierarchies which can be specified for one process is restricted to two, namely the document-hierarchy and the role-hierarchy.

Any document and role class can and must be specified by the same attributes. The reason for having the same attributes is, that exactly the same information must be available for all document and role classes to support their unique interpretation. Making use of inheritance, the attributes have to be defined once for the root-class of the document hierarchy. For inherited attributes, redefinition is supported. In the EER-models shown in section 3.3 the predefined part of the hierarchy is represented shaded.

A modification of class instances is performed through methods. Since only documents are modified during process enactment, the specification of methods is limited to document-classes. For every documement class, an unlimited number of methods can be specified.

In accordance to the distinction made in section 2, methods must be further divided into those methods which are applied by the user (e.g. *ascii_editor*) and those methods which can automatically applied by the system (e.g. *compiler*). On the specification level this distinction is necessary, because otherwise the enaction component can not distinguish between those tools which must be invoked automatically and those which must be invoked interactively.

Since the problem definition does not give reasons for the introduction of multiple inheritance, only single inheritance is supported. If two classes are related by an inheritance relationship, all attributes, methods, and relationships are inherited. Redefinition of attributes is supported.

For Statecharts, examples for application specific adaptions are:

The behavioural perspective is specified based on state changes. A change of state can only be triggered by a tool invokation or a change of state for a related document. Therefore, only labels

- having an empty event and a method identifier in the action part, and
- having a condition (see section 3.3, step 6) in the event part and an empty action part

are supported.

The conditions are structured as follows: the first part is a relationship identifier, the second part a class identifier, and the third part a state identifier. For separating the identifiers, a # is used (e.g. *in(<compile_errors_are_in>#err_rep#no_errors)*). If the relationship identifier is given in <>, the target document of the relationship is refered to. To refer the source document, -<> must be used. The class identifier in the second part contains the identifier of the source or target class, respectively. The third part, the state identifier, contains the state the refered document must be in that the condition becomes true.

Two kind of conditions are distinguished:

- those which check if at least one related document is in a certain state (*exists*-quantor), and
- those which check if all related documents are in a certain state (*forall*-quantor).

Conditions starting with an *exists*-quantor are connected by a logical *or*, those starting with a *forall*-quantor by a logical *and*.

4.2 Context-sensitive conditions

This subchapter includes examples for context-sensitive conditions. Some of those which must be kept for Statecharts are:

Method identifiers used in the action part of a label must be specified in the document class the Statechart belongs to.

For a condition as label the following context-sensitive conditions must be kept:

- a relationship with the given relationship identifier must be connected to the document-class the Statechart belongs to,
- if there is a - in front of the relationship identifier, the identifier of the source class, the relationship is connected to, must be the one used in the second part; otherwise, the identifier of the target class must be used, and

- the state identifier must be defined for the Statechart of the class identified by the second part.

Examples for context-sensitive conditions which must be kept for tables are:

For all identifiers used in the columns *working document type* and *related document type*, a class with the given identifier must exist.

The state (column *state*) specified for the document class must exist for that class.

4.3 Static semantic of process models

The consideration of the above given application specific adaptions as well as the context-sensitive conditions can be supported using a syntax-directed editor. What can not be ensured using an editor is, that the specified process models are semantically correct. To ensure semantic correctness, an analyser has to be used. When analyzing a process model, to kind of faults may occur:

- warnings and
- errors

A *warning* indicates that the process model can be enacted but that some inconsistencies have been found. An *error* indicates that this process model can not be enacted.

Examples for situations which cause warnings are:

A document-class is specified without methods. The consequence is, that the document contents can not be accessed or manipulated. The document becomes useless for the process.

There are similar tables specified for different roles. Since the role specification is the same, the two roles can be combined to one.

A document can not be set to a specified state. The reason might be, that this state is isolated (no transition is leading to this state) or that the condition can not become true.

Examples for situations which cause errors are:

Attributes are not fully specified. The interpretation will cause on error due to missing information.

A transition, having as source and target state the same state and as label a batch-method. This will result in a non determining tool execution.

5 From process-design to process-program

In the previous section a graphical design-language for the specification of software processes has been introduced. In the following it is shown, how the graphical language can automatically be mapped to an enactable process programming language. Advantages of this approach are, that (1) the specified processes can immediately be enacted, (2) the graphical language gets a well defined semantic, and (3) process improvements become part of the enactable process.

As enactable process programming language, the process programming language of Merlin is chosen. The Merlin process programming language is a dedicated rule-based language whose semantics is PROLOG-like [JPSW94]. The reason for chosing this approach is its efficiency in execution (no type checking is performed) and its possibility to realize changes on the fly.

In Merlin, software development processes are described on three different levels, namely the **Project**, the **Process**, and the **Kernel** level respectively (see [Kru93]).

The **project level** description is supported by a set of predefined predicates which define a particular project status. This concerns, for example, the name of a project and the persons who participate in that project, the roles of those persons, and their responsibilities.

Those facts are usually not introduced as "flat" facts by the process engineer but rather generated as output from a project management tool. The predefined predicates serve as an interface definition between such a tool and the Merlin fact base. For prototyping purposes those facts can, of course, also be types in directly.

The next level of description, namely the **process level**, is again supported by providing a predefined set of predicates. Process level means, a definition of the ingredients of any process which are independent of any particular project. Those ingredients include e.g. types of documents, possible document states, possible state transitions.

Finally, the **Kernel** of a process definition in Merlin is defined by a set of PROLOG-like rules using the above described fact base. This level encapsulates the Merlin philosophy and acts as the inference machine for software processes. The Kernel specifies for example how to interpret the basic entity-types (e.g. roles, documents), how to build a customized working-context for any user, how to deal with a client/server architecture and how to realize multi-user support through a sophisticated transaction concept ([PSW92], [JPSW94]).

Process and project description are simplified as the Kernel predefines the predicates to be used for describing the Process level and the Project level. Defining new processes and projects basically means filling in the values of parameters of these predicates.

The above informally given semantics of the various facts is formally and precisely defined by the rules of the Kernel. The Kernel only needs to be changed when the user interface paradigm is changed or new features like configuration management or transaction management are added. It is thus the most stable part of the process description. Changes to the software process require changes to facts describing the Process and Project (which can be held more easily in a consistent state).

Further examinations of the Merlin tripartition have shown, that the *Kernel* is the invariant part of the process-program. The *Project* concerns the management of instances during process enactment and therefore can also be ignored during process-design. The part of a Merlin process-program describing a specific process is the *Process* description.

For a detailed description of the predicates which represent the *Process* level the reader is refered to [JPSW94].

In the following, the feasibility of mapping the process design to a process programm is shown.

Therefore, parts of the example given in section 3.3 are mapped to the *Process* level of the Merlin process programming language. The resulting facts can be interpreted by the process-engine.

The EER-diagram shown in figure 3 (page 8) can be mapped to the facts as follows:

Step 1: For each concrete document class, a fact of type *document_type_structure* is generated. Predicate *document_type_structure* specifies the existing document classes as well as the type of file used by these classes to store the document contents. The first parameter includes the class identifier and the second the type.

The information needed is derived out of the class identifier (first parameter) and the value of attribute *document_type_structure* belonging to the document class.

Examples for generated facts are:

document_type_structure(c_module, ascii_file).

document_type_structure(specification, ascii_file).

Step 2: For each relationship type, a fact of type *document_relation_type* is generated. Predicate *document_relation_type* specifies the the type of relationships which exist between document classes.

The first parameter includes the relationship identifier, the second the source class, and the third the target class of the relationship. The information needed is derived out of the source and target class of the relationship and the relationship identifier.

Examples for generated facts are:

document_relation_type(imports, c_module, c_module).

document_relation_type(uses, specification, specification).

document_relation_type(is_implemented_in, specification, c_module).

document_relation_type(realisation_is_imported_in, specification, c_module).

The next steps are dedicated to the mapping of the Statechart given in figure 5 (page 9).

Step 3: For each Statechart, one fact of type *document_type_states* is generated. Predicate *document_type_states* specifies the states allowed for a document instance of the specified class.

The first parameter includes the class identifier and the second the list of states allowed.

The following fact is generated for the document class *c_module*:

document_type_states (c_module, [incomplete, not_yet_implemented, implemented, not_yet_compiled, compiled_with_errors, compiled, not_yet_tested, complete]).

Step 4: For each outgoing transition, having an identifier of an interactive method as action part of the label, a fact of type *document_type_tools* is generated. If several outgoing transitions have the same method identifier in the action part, they are combined in one fact.

Predicate *document_type_tools* specifies the interactive tools which can be applied to a class instance. The state the document instance must be in before tool invokation as well as the states the document instance can be set to after tool termination is specified. The access right of the tool to the document contents is given to be able to distinguish between tools which allow modification and those which do not.

The first parameter contains the class identifier of the class this Statechart belongs to, the second the method identifier, the third the access rights to the document, the fourth the source state of the transition, and the fifth all the target states of those transitions, having the before given source state and the method identifier given in the second parameter. *Any_state* is a placeholder for all states this document instance may be in and *[]* indicates, that the document instance will be set to the state it has been in before tool invokation.

Examples for generated facts are:

> *document_type_tools (c_module, ascii_editor, writable, not_yet_implemented, [not_yet_implemented, implemented]).*
>
> *document_type_tools (c_module, ascii_pager, readable, Any_state, []).*

Step 5: Similar as for interactive methods, for batch methods a fact of type *document_type_call* is generated, except that no access right is needed.

An example for a generated fact is:

> *document_type_call (c_module, c_compiler, not_yet_compiled, [compiled_with_errors, compiled]).*

Step 6: For each outgoing transition, having an "exists" condition as label, a fact of type *next_state_or_condition* is generated. Predicate *next_state_or_condition* specifies the change of states for a document instance triggered by a change of state for a related document.

The first parameter contains the class identifier of the class this Statechart belongs to, the second the source state of the transition, the third the target state, and the forth a list of triples which specify conditions which check, if a related document is in a desired state. The first parameter of this triple is *source* or *destination*, indicating, which of the document classes related by the relationship type given in the second parameter must fulfil the condition. The third parameter includes the state this document instance must be in that this part of the condition becomes true. If there is more than one triple, the corresponding conditions are connected by a logical *or*. Any_state and *[]* are interpreted similar as explained before.

Examples for generated facts are:

> *next_state_or_condition (c_module, Any_state, incomplete, [[source, is_implemented_in, not_yet_specified]]).*
>
> *next_state_or_condition (c_module, compiled_with_errors, not_yet_implemented, [[destination, compile_errors_are_in, with_errors]]).*

Step 7: Similar as for transitions having an "exists" condition as label, for those having a "forall" condition a fact of type *next_state_and_condition* is generated.

Predicate *next_state_and_condition* specifies the change of states for a document instance triggered by a change of state for a related document. Automation conditions are used to change the state of a document if the preceding activity is completed correctly (for further details see section 2).

Examples for generated facts are:

> *next_state_and_condition (c_module, incomplete, not_yet_implemented, [[source, is_implemented_in complete]]).*
>
> *next_state_and_condition (c_module, implemented, not_yet_compiled, [[destination, imports, complete]]).*

The last step is dedicated to the mapping of the role hierarchy given in figure 7. The tables shown in figure 8 are part of the role specification and have to be mapped too.

Step 8: For each table, several facts of type *roletype_document_work_on* are generated. Predicate *roletype_document_work_on* specifies the visibility of document instances for a developer under a certain role.

The first parameter contains the role identifier of the class this table belongs to, the second the first document class (column *working document type* in the shaded row), the third the state document instances of this class must be in to be visualized (column *state* in the shaded row). The last parameter contains tuples of document class identifier and state and is derived out of the rows below the shaded row.

Examples for generated facts are:

roletype_document_work_on (programmer, c_module, not_yet_implemented, [[specification, complete], [test_plan, review_rejected]]).

roletype_document_work_on (programmer, c_program, not_yet_executed, [[c_module, not_yet_programmed]]).

roletype_document_work_on (designer, specification, not_yet_specified, [[specification, complete], [specification, not_yet_specified], [specification, specified]]).

6 Related Work

A number of related projects have also combined the ideas of an integrated environment and the explicit representation of a software process. This paper is focused on the design of process-programs and therefore existing PSDEs are examined under its support for a user-friendly representation of the process.

Marvel [BK90] uses a rule-based language for process modeling. The recently developed language ASL [KPBS93] is an extension of MSL [EFP88] and combines the specification of local constraints on individual tools and data with the specification of global control flow and synchronization. A main disadvantage of MSL, using post-conditions to specify how an activity on one object might influence all related objects, has not been considered for the new implementation. These post-conditions become very complex because also the continuous changes have to be considered and are therefore difficult to change and keep consistent. Similar to MSL, ASL is still on the level of "traditional" programming languages and does not support structuring mechanisms or an intuitive representation of the process.

The petri-net approach followed by **MELMAC** [GJ92] is a good candidate for a graphical representation of the behavioural perspective of a process. A disadvantage is, that the modeling of the behaviour of one object is completely interwined with the modeling of the behaviour for a set of objects. This results in very complex nets which tend to become so large that they are neither comprehensible nor manageable, even through providing tool support.

SOCCA [EG94] uses an EER-model for modeling the date perspective, state transition diagrams and PARADIGM for modeling the behaviour and synchronization perspective, and object flow diagrams for modeling the process perspective. SOCCA and ESCAPE are based on similar formalisms, but there is one important difference which set off ESCAPE. In SOCCA, the external behaviour of an object class is specified by a state transition diagram. At this level, only those aspects are considered which result out of a modification of the object of concern. For specifying the synchronization aspect, PARADIGM is used. Synchronization is specified on the instance level and results in very complex state transition diagrams for relatively small problems. What still remains unclear is, how to specify changes triggered by a change of state for a related document. The Statecharts used by ESCAPE support the specification of external behaviour and synchronization in a concise way.

The Software Engineering Institute (SEI) [3] uses the commercial available tool STATEMATE for software process modeling [CKO92] [HLN+90]. Main goal is a better understanding of processes, which is reached by a simulation of the specified process models. Furthermore, through simulation, also a validation of processes is supported. Similar to ESCAPE, STATEMATE offers an intuitive graphical representation and uses Statecharts for the specification of the behavioural perspective of processes. A disadvantages is, that no object-oriented approach is supported to specify the structural perspective.

All mentioned languages, except Marvel, support a graphical language. They are on a higher level of abstraction than "traditional programming" languages. MELMAC lacks in the point that nets become very complex because the specification of the behaviour of one object is interwined with the specification of the behaviour of a set of objects. In STATEMATE, the specification of the static perspective is build around functionality rather than around object classes, which decreases the support for changeability. SOCCA is the approach which supports most of the aspects to be considered for process modeling, but its support for synchronization must be improved.

7 Conclusion

To summarize, none of the prevailing PSDEs today support the design and enactment of process-models. The introduced process design language ESCAPE is the first approach which offers a convenient representation formalism which supports a visual representation, enactment, and is based on concepts which are well accepted. The process-information is structured through object-oriented clustering and the process design becomes minimal because features offered by Merlin (e.g. a sophisticated transaction concept, a customized users working context) can be presumed and must not be considered for process design. ESCAPE supports an explicit modeling of the structural and behavioural perspective of processes and through mapping it to the Merlin rule-language, the language becomes formally well defined. A further advantage is, that the semantic of the *Process* facts is fixed which frees the process-engineer from technical details.

The next steps in extending the prototype include:

- the development of a syntax-directed editor for the process design-language introduced,
- the development of a tool to transform a given design into the enactable Merlin *Process* facts,
- an examination of how changes on the fly can be supported by the given approach and
- the development of a semantic analyzer which supports the analysis of a variety of aspects of completness, correctness, and consistency.

[3] Carnegie-Mellon University, Pitsburgh

Acknowledgements

I want to thank the entire Merlin team for intensive discussions and especially S. Sachweh, D. Jacobs, O. Neumann, K. Ross and W. Schäfer for giving helpful comments on earlier versions of this paper.

References

[AM92] V. Ambriola and C. Montagero. *Oikos at the Age of Three.* In *Proceedings of the* 2^{nd} *European Workshop on Software Process Technology*, Trondheim, Norway, sept. 1992. LNCS 635, Springer, Berlin.

[BaAM91] Sergio Bandinelli and Carlo Ghezzi amd Angelo Morzenti. *A Multi-Paradigm Petri Net Based Approach to Process Description.* In *Proceedings of the* 7^{th} *International Software Process Workshop*, Yountville, California, October 1991.

[BK90] Naser S. Barghouti and Gail E. Kaiser. *Multi-Agent Rule-Based Software Development Environments.* In *5th Annual Knowledge-Based Software Assistant Conference*, pages 375–387, New York, September 1990.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design.* Benjamin/Cummings, second edition, 1994.

[CKO92] Bill Curtis, Marc I. Kellner, and Jim Over. *Process Modeling. Communications of the ACM*, pages 75–90, September 1992.

[DGS89] W. Deiters, V. Gruhn, and W. Schäfer. *Process Programming: A structured Multi-Paradigm Approach Could be Achieved.* In *Proceedings of the* 5^{th} *International Software Process Workshop*, Kennebunkport, Maine, USA, September 1989.

[EFP88] Gail E.Kaiser, Peter H. Feiler, and Steven S. Popovich. *Intelligent Assistance for Software Development and Maintenance. IEEE Software*, pages 40–49, May 1988.

[EG94] Gregor Engels and Luuk Groenewegen. *Specification of Coordinated Behaviour by SOCCA.* In Brian C. Warboys, editor, *Proceedings of the* 3^{rd} *European Workshop on Software Process Technology*, pages 128–151, Villard de Lans, France, February 1994. LNCS 772, Springer, Berlin.

[Fer93] Christer Fernström. *PROCESS WEAVER: Adding Process Support to UNIX.* In *Proceedings of the* 2^{nd} *International Conference on the Software Process*, pages 12–26, Berlin, Germany, February 1993.

[FH93] Peter H. Feiler and Watts S. Humphrey. *Software Process Development and Enactment: Concepts and Definitions.* In *Proceedings of the* 2^{nd} *International Conference on the Software Process*, pages 28–40, Berlin, Germany, February 1993.

[GJ92] V. Gruhn and R. Jegelka. *An Evaluation of FUNSOFT Nets.* In J.-C. Derniame, editor, *Software Process Technology - Proceedings of the* 2^{nd} *European Workshop*, pages 194–214, Trondheim, Norway, September 1992. Springer. Appeared as Lecture Notes in Computer Science 635.

[Gru91] Volker Gruhn. *The Software Process Management Environment MELMAC.* In *Proceedings of the 1^{st} European Workshop on Software Process Modeling*, pages 191–201, Milan, Italy, May 1991. A.I.C.A. Press.

[Har87] David Harel. *Statecharts: A visual formalism for complex systems. Science of Computer Programming*, 8:231–274, 1987.

[Har88] David Harel. *On Visual Formalisms. Communications of the ACM*, 31(5):514–530, May 1988.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. *STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering*, 16(4):404–414, 1990.

[HPSS87] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. *On the Formal Semantics of Statecharts.* In *Proceedings of the 2nd IEEE Symp. Logic in Computer Science*, pages 54–64, Ithaca, New York, 1987.

[HSO90] D. Heimbigner, St.M. Sutton, and L. Osterweil. *Managing Change in Process-Centered Environments.* In *Proceedings of the 4^{th} ACM SIGSOFT Symposium on Software Development Environments*, Irvine, California, USA, December 1990.

[JPSW94] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. *Merlin: Supporting Co-operation in Software Development through a Knowledge-based Environment.* In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, Research Studies Press (UK). distributed by: John Wiley & Sons, 1994.

[KPBS93] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. *A Bi-Level Language for Software Process Modeling.* In *Proceedings of the 15^{th} International Conference on Software Engineering*, pages 132–143, Baltimore, Maryland, May 1993. IEEE Press.

[Kru93] Rainer Kruschinski. *Konzeption einer Wissensbasis für die Software-Prozess-Modellierung (in German).* master thesis, University of Dortmund, 1993.

[Lon93] Jacques Lonchamp. *A Structured Conceptual and Terminological Framework for Software Process Engineering.* In *Proceedings of the 2^{nd} International Conference on the Software Process*, pages 41–53, Berlin, Germany, February 1993.

[Ost87] L. Osterweil. *Software Processes are Software Too.* In *Proceedings of the 9^{th} International Conference on Software Engineering*, Monterey, California, April 1987.

[PSW92] B. Peuschel, W. Schäfer, and S. Wolf. *A Knowledge-based Software Development Environment Supporting Cooperative Work. International Journal on Software Engineering and Knowledge Engineering*, 2(1):79–106, 1992.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[War89] Brian Warboys. *The IPSE 2.5 Project: Process Modelling as the basis for a Support Environment.* In *International Conf. on System Development Environments & Factories*, pages 59–74, Berlin, 1989. Pitman.