Software Process Enactment Based on an Object Oriented Description

Wolfgang Schramm[*] Martin Verlage[*] Peter Knauber[*] Rainer Bleisinger[**]

[*] University of Kaiserslautern, Deptartment of Computer Science
P. O. Box 30 49
D-6750 Kaiserslautern
phone: (+49) (631) 205-2642

[**] German Research Center for Artificial Intelligence (DFKI)
P. O. Box 20 80
D-6750 Kaiserslautern
phone: (+49) (631) 205-3216

Abstract

The complexity of software development processes and the cooperative work of many people to accomplish this task require automatic support. Therefore, a variety of components are combined to form a software development environment – the MoMo system. This system is based on the MoMo meta model which relies on the object oriented paradigm to represent the subjects of software development processes including temporal aspects.

The enactment of a software process comprises planning, control and execution of software activities. Based on an object oriented language, developed as a representation mechanism for the MoMo model, software processes are explicitly and formally described. This description is compiled into an executable form which is the basis for project enactment performed and controlled by the MoMo system. Typically, the information about the subjects of the process is incomplete at the start of a project. As the project proceeds more and more information becomes available. This requires for incremental modification of the project description during enactment. After finishing a software development process a quantitative measurement of the enacted project has to be performed.

The MoMo system supports the adaptability and reusability of suitable software process models. Thus, appropriate models may be fitted to any given environment. The need for appropriate flexibility and dynamics are guaranteed by object oriented realization.

Keywords

software development environment, software process, meta model, process model, process enactment, process knowledge, knowledge representation, temporal reasoning

1 Introduction

The high degree of complexity and huge amount of data in the context of software development and the cooperative work of many people require automatic support by constructing software to achieve qualitatively satisfying products. When we are concerned with development, improvement or maintenance of software products, with the objective to accomplish this task in an effective manner, we are primarily interested in the process of constructing these products. We can state that the products are determined by their processes. Thereby, we recognize a scope of three duties: planning, enacting and analyzing the software (development) process.

Various models have been designed to guide the users through the software development task; the waterfall model [Royce 70] is one of the best known. However, most software processes (SPs) used in practice are only implicitly defined. To be able to compare, or to improve SPs, we have to define them explicitly. Therefore, we elaborated a theoretical meta model enabling us to specify different SP models with an unique mechanism. Consequently, we designed an appropriate language as a description formalism. Thus, we can express, enact, and analyze SPs more reliably. For all tasks, the elements of SPs and their relationships have to be identified and several constraints on resources have to be considered, like limitations in budget, staff, equipment and time.

In this paper, first we give in chapter two an overview of the MoMo system architecture and describe its particular components, partially in more detail.

The system reflects the concepts of our meta model and the basic mechanism for modelling SPs. In the third chapter we discuss the main properties of the MoMo meta model. Therefore, we present the basic formal constructs and give examples using the project description language ProLan.

The main focus of this paper is the aspect of using the MoMo system, especially the enactment of SPs. We designed the MoMo system as an environment comprising all jobs concerned with SPs. In chapter four we first give a brief overview how to work with the system. Thereby, we identify three elementary steps to derive benefits from using our MoMo system.

Then we point out how a SP is enacted by the virtual process machine, the central component of the MoMo system. Thereby, according to the description a software process is divided into a hierarchie of subprocesses and the atomic processes are executed to bring the project into another state.

Finally, we make some remarks about related research activities and conclude with a summary of the advantages of the MoMo approach as well as an outlook on some future work.

2 Architecture of the MoMo System

The MoMo system serves as an environment for software process modelling and enactment. The modular layout of the system architecture reflects the several specific tasks of software process management.

We have identified four major components for the system (see Fig. 1): the Virtual Process Machine, the ProLan Component, the Temporal Reasoner, and the Analyzer.

Virtual Process Machine

The *virtual process machine* (VPM), as the central process enacting mechanism coordinates all the SP objects and executes the process activities. While interpreting the process description the VPM may cause the user to modify or complete the SP description. This component will be discussed in more detail in Chapters 4 and 5.

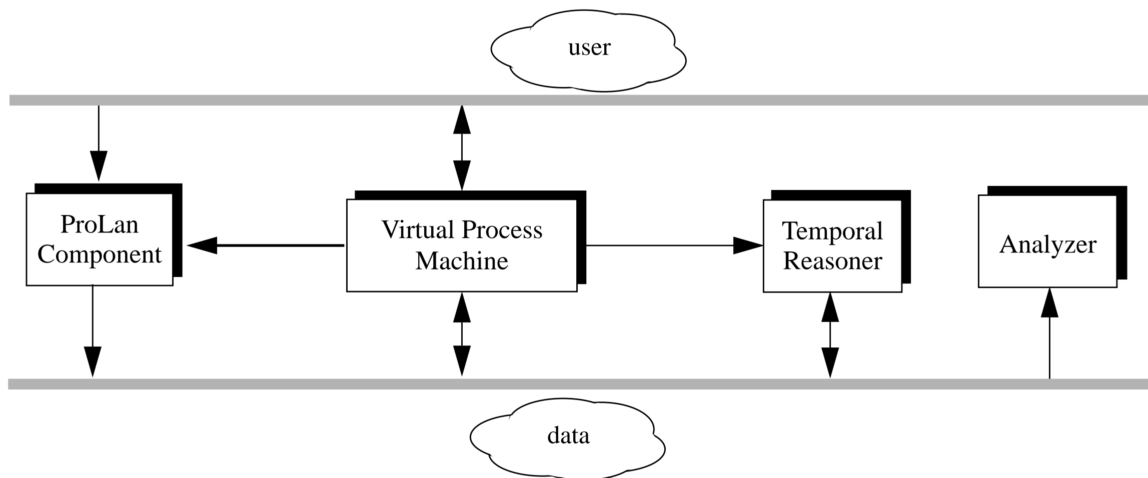

Fig. 1: The architecture of the MoMo system

ProLan Component

For the description of various kinds of process models the very expressive project description language ProLan is designed. An editor and the compiler for this process description language build the *ProLan component* (cf. [Knauber 91]). The compiler has to transform ProLan descriptions into an executable form.

The compiler, itself organized in an object oriented way, checks the correctness of the syntax and semantics of ProLan descriptions. In the syntax analysis for each ProLan class there is built a corresponding object in the nametable. Both, the semantic analysis and the temporal reasoner use this structure to perform their tests. The code generator translates each ProLan class (see Chapter 3) into a CLOS class description (cf. [Steele 90]) and/or a number of methods.

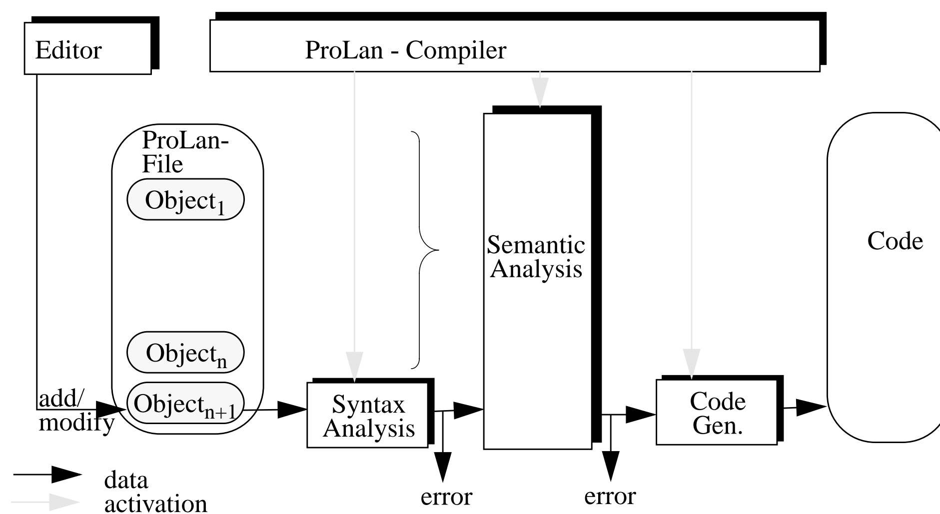

Fig. 2:The ProLan component

As a consequence of the object oriented approach, single ProLan classes can be added or modified during project enactment (see Fig. 2), controlled by the VPM. In the case of correct changes the new class is translated into CLOS code for further use. If there occures a syntactical or semantical error, the modifications are ignored.

Temporal Reasoner

The *temporal reasoner* (TEMPO) checks the temporal dependencies between the activities involved in the software development process (cf. [Bleisinger et al. 91]).

On the one hand, the code generated by the ProLan compiler builds the input for the temporal reasoner. Thereby, the predefined temporal information is verified. On the other

hand, the temporal reasoner is invoked by the VPM while enacting the software process. The reason for that activation of TEMPO is either additional temporal information specified by a user or an update by the VPM itself.

Temporal inconsistencies invoke exception handling, which either enforces the correction of ProLan code or the revision of the projects temporal specifications. Because we have defined the system class INTERVAL (see Chapter 3) for representing temporal knowledge of activities, this task can be done ignoring the other SP elements. Only the interval set which builds a network resulting from the temporal relations must be considered.

The temporal reasoner has to perform the following tasks:

- inspecting for correct specification (values in the defined set),
- consistency checking of the stated values,
- infering additional temporal knowledge.

The different temporal knowledge types (qualitative and quantitative order, qualitative and quantitative duration) enforce several techniques for performing all tasks. Moreover, the relationships of these types must also be taken into account [Faidt et al. 89]. So, three levels of temporal reasoning are identified:

- qualitative temporal reasoning (relational),
- quantitative temporal reasoning (numeric),
- combination of qualitative and quantitative temporal reasoning.

The architecture of the temporal reasoner contains two layers: the declarative representation layer and the execution layer (see Fig. 3).

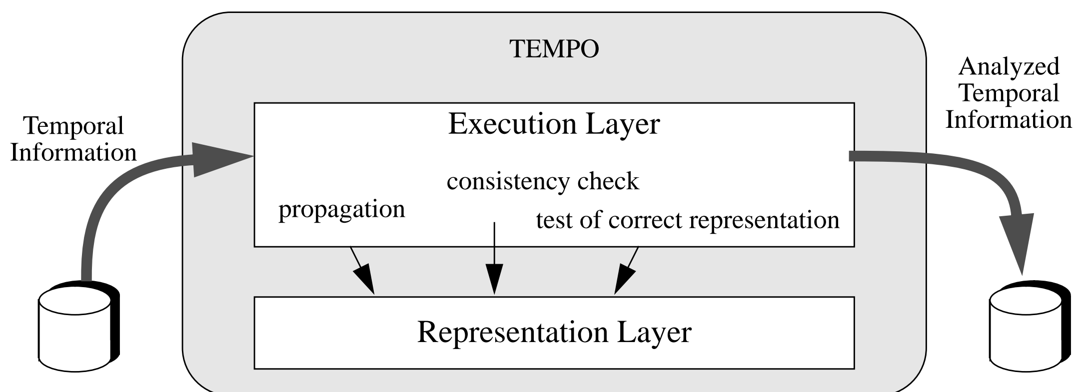

Fig. 3: Two layered temporal reasoner architecture

Within the representation layer, the sets of admissible values for each temporal knowledge type, the consistency conditions, and the propagation conditions can be defined. The algorithms depending on the specifications of the representation layer and their order of application build the execution layer [Bleisinger 91].

Analyzer

The process *analyzing component* evaluates the data of software development processes. The evaluations are necessary for two clients: to give the users feedback about the actual project as well as to give the management information for future projects.

First, during the enactment of a single project the analyzing results inform the user about the progress of the software development process. In this way, hints are given about the suitability of the applied software process model.

Second, experiences of former projects compared with data of current enactments can cause an update of the knowledge base, used for further projects. For example, the fact that a class is unchanged during several projects leads to the assumption that this class is useful in more projects.

3 The MoMo Model

The MoMo system is based on an appropriate meta model - called MoMo - for the description of software development processes (for a detailed description of the MoMo model, cf. [Schramm 91]). Osterweil stated that the descriptions of software processes are some kind of program, like application programs. Instead of a static meta model description (cf. [Osterweil 87]) we consider a model relying on the object oriented paradigm to be adequate to represent the complex subject of software process modelling (cf. [Meyer 88]). Moreover, MoMo is a behavioral model (cf. [Williams 88]), because the descriptions of the activities focus more on the effects which the activities produce than on the specific procedures used to produce those effects.

The MoMo model is more than a combination of these two approaches. As we see below, additional aspects increase the expressiveness of the model, in particular with regard to its use in a “realistic” software project. A wide range of structures and dependencies between the elements of SPs is represented by the model. According to the requirements of project management we are also able to specify temporal knowledge at different levels of accuracy and abstraction.

To understand the complex relationships in process models, they are described by less complex subsystems and relations between them in a hierarchical manner. So, we describe the different kinds of the SP elements as classes which serve as templates for their instances. These classes again are subclasses of system classes which determine the fundamental semantics of the user defined classes (for illustration see Fig. 4).

With respect to the complexity of the software process modelling task, we have developed a three level knowledge representation for software process descriptions:

Level one: System classes represent our *meta knowledge* about software processes in general.

Level two: User classes represent *process model knowledge* which is determined by the environment, e.g. the organization in which the system is running. On this level life-cycle models are specified. The user classes are described at this level with the project description language (ProLan, cf. [Knauber 91]) using the system classes.

Level three: Process elements (i.e. the instances of the user classes) represent the *project knowledge* about current processes (projects in enacment), combining the software process model with values of the real world.

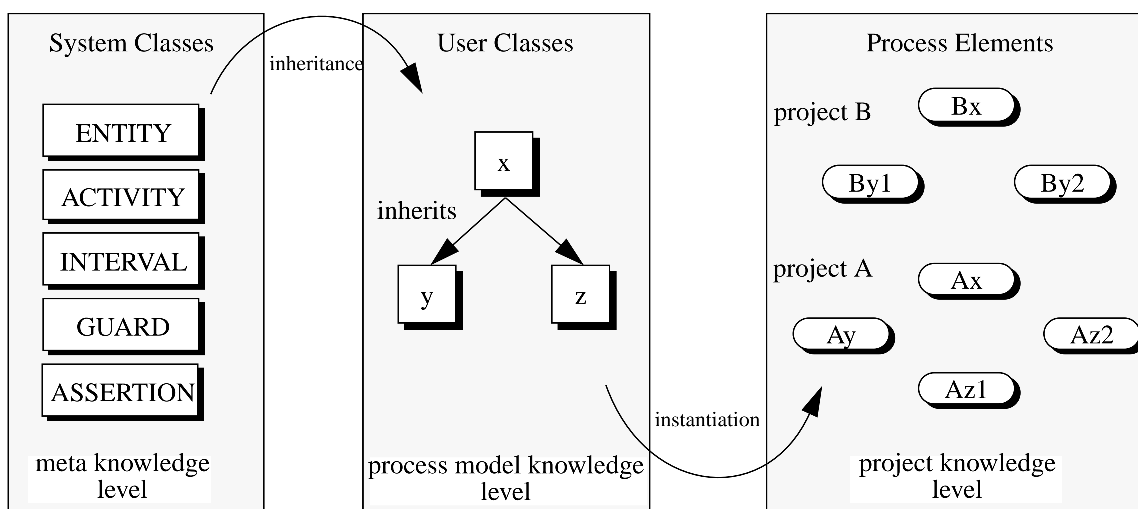

Fig. 4: The three knowledge representation levels

To achieve flexible handling of user classes, i.e. changing the class definitions during runtime of the SP, we treat user classes like ordinary objects. Therefore, we introduce a

metaclass, named MOMO-CLASS. This metaclass is the class of all user classes (not shown in Fig. 4).

Different types of relations can be defined between these classes. All objects of the software process are considered to be objects in the sense of the object oriented paradigm. Every object contains slots to store values and methods to modify the values or to communicate with other objects. The different representation of the classes is caused by the different use of their objects. Opposite to the longer life time of entities, for example activities are used for determination of the project's schedule.

In the following, we consider the predefined system classes which are essential to compose the MoMo model. The given examples are written in ProLan.

System Class ENTITY

An entity represents a physical or logical aspect of the real world (programs, documents, persons). It contains data and/or aggregates of other entities. We use aggregation to model abstraction. Entities with similar properties are related by inheritance. There are four kinds of attributes which can appear in an entity:

(1) Declarative Attributes
Attributes in the common sense.

(2) Operational Attributes
Methods which are bound to the subclasses of the system class ENTITY; operational attributes can be bound to so called *active values*.

(3) Structural Attributes
Aggregates of one or more entities of lower abstraction levels.

(4) Relational Attributes
Logical link to an entity or an activity.

Example 1:
```
ENTITY Program;
 ATTRIBUTES:
  Name       :   STRING INIT "NoName";
  Modules    ->  (Module);              -- Module is an Entity
  Programmer ->  Programmer-Person;     -- P-P is an Entity
 METHOD Lines-of-code( Name : STRING ) : INTEGER;
  RETURN L_o_C( Name );                 -- L_o_C is a tool
 END_METHOD Lines-of-code;
END_ENTITY Program;
```

System Class ACTIVITY

Activities are the most important units of the SP. They transform the project from one state into another one and thus, elaborate the project. Activities can be structured by decomposing into subactivities. This approach is similar to the technique to solve complex problems by dividing them into less complex subproblems. Each of these small problems can be treated isolated and by combining the partial solutions the main problem is solved.

An activity consists of the following parts:

(1) Attributes
The same kind of attributes as specified for entities, except the structural attributes, are permissible. Additionally, temporal attributes are associated with activities. For the sake of simplicity we treat temporal attributes like ordinary attributes in this context.

(2) Precondition
Description of the state that enables the execution of the activity. It is used to express logical dependencies between activities.

(3) Action Part

Description of the work performed by an activity. There are two types of actions: Elementary activities modify objects or invoke tools; these are the modifications of data the user will see.

Structured activities initiate other (sub)activities[1]; compare this to the structural attributes of the system class ENTITY. With the help of structured activities, we are able to describe different SP models. MoMo provides two kinds of constructors for organizing the enactment of subactivities. It is possible to specify the order of subactivity enactment in both ways, a prescriptive one (constructors: SEQUENCE, SELECTION, CASCADE, REPEAT ... UNTIL) and a descriptive one (constructor: SET). The execution order of descriptively specified subactivities is determined at SP runtime.

In Chapter 5 we will discuss the meaning of the action part in more detail.

(4) Postcondition

Description of the state after the execution of the activity, assumed to be valid.

A typical property of an activity is the fact that it consumes time. This means that an activity starts and finishes at particular time; for that reason we associate a time interval (see below) with each activity. There are two kinds of non orthogonal dependencies between activities: logical and temporal ones. The logical dependency always implies a temporal dependency. When an activity a_1 produces results needed by another activity a_2, then a_2 depends on a_1. This implies that a_1 must be enacted before a_2. When there are two independent activities, the project manager may decide the temporal order of enactment of the particular activities. Therefore, our activity concept must provide means for the specification of temporal information.

Example 2:
```
ACTIVITY Implement-Module;
  SUPER_ACTIVITIES: Perform-Project;
  ATTRIBUTES: Source  -> Module;
              time    -> INTERVAL;
  PRECOND:  Source.status = designed;
  ACTION:   REPEAT
               Edit(Source.Name);
               Compile(Source.Name);
            UNTIL Source.status = compiled;
            Link(Source.Name);
  POSTCOND: Source.status = linked;
END_ACTIVITY Implement-Module;
```

System Class INTERVAL

Temporal information about activities will be specified by the association of an interval. Relevant knowledge for intervals is partitioned in a quantitative (numeric) and a qualitative (relational) part. Both parts contain declarations for the temporal duration and the temporal order of intervals.

The object oriented representation of this temporal knowledge within the system class requires at least different slots for the identified four temporal knowledge types. For handling roughly numeric data, the quantitative information is additionally structured. For example, possible minimal and maximal durations are used, supplemented by actual and average durations for later statistical evaluation. Analogous, for the temporal order earliest and latest possible start and end times are stored. Additionally, the actual start and end times are recorded.

1. Note, the term *subactivity* is not used in the sense of object oriented programing. The term does not express a “is-a” relation, but a “part-of” relation.

```
Example 3: INTERVAL Implement_Module_Interval;
            QUANT_ORDER:  EARLIEST_START: 01.01.1991;
                          LATEST_END: 24.06.1991;
            QUAL_ORDER:   (started_by)(Edit);
                          (contains)(Compile);
                          (finished_by)(Link);
            QUANT_DURATION:MIN_DURATION: 2 m;
                          MAX_DURATION: 4 m;
            QUAL_DURATION:(longer)(Edit, Compile, Link);
           END_INTERVAL Implement_Module_Interval;
```

Each activity class is related to exactly one interval class. If a user does not specify any temporal knowledge explicitly for an activity, the associated interval class is supplied with default values of system class INTERVAL.

System Class GUARD

Guards wait for the occurrence of certain events (e.g. creation of an entity, modification of a slot value, expiry of a deadline). Each guard associates events with activities. If an event has occurred the guard invokes the associated activities (causes its instantiation) and so, an appropriate reaction to the event is initiated. The new activities need not to have structured relations to existing activities. In particular, the activity that caused the event needs not to have regard for it.

```
Example 4: ACTIVITY_GUARD Start-Project;
            ON USER-EVENT (Begin-Project)
            DO Perform-Project;
           END_GUARD Start-Project;
```

System Class ASSERTION

Assertions watch the current state of a project. They define constraints which limit the set of valid project states. After execution of an activity, all assertions are checked. If a constraint is violated, the exception handling of the corresponding assertion is executed. Imagine constraints as user defined runtime errors. An exception handling is similar to the action part of activities. The execution of an assertion runs with high priority to obtain a valid project state.

```
Example 5: ASSERTION Control-Programmers;
            UNLESS NUMBER-OF-WHERE (Workstation, NOT busy) < 10
            DO EXECUTE:Warning ("Too less programmers working!");
                       MONITOR (Programmer-Person);
           END_ASSERTION Control-Programmers;
```

4 Working with the MoMo system

Applying the MoMo system, an “intelligent” support of process management could happen. For that purpose, the following three steps are performed: describing a software process model, enacting a project, and finally analyzing a project (see Fig. 5).

Step 1: Preplanning of a project - describing a software process model

To provide knowledge for development and maintenance of software it is necessary to describe the development process formally. The project description language ProLan supports an object oriented view of activities, entities, and other SP elements. A ProLan description is primarily a skeleton plan of the SP. SP elements are described in terms of classes. The classes serve as templates, instantiated and filled during project enactment. ProLan also provides several constructs to express causal and temporal relationships between project activities.

The ProLan compiler tests the correctness and consistency of the SP description. Especially, the temporal information is checked for contextsensitive consistency in interaction

with the temporal reasoner. The results of temporal inference complete the explicitly and implicitly specified temporal information of the SP model.

Accepting the description of the project, the ProLan compiler stores the generated code in the process database. This data can be changed dynamically during process runtime.

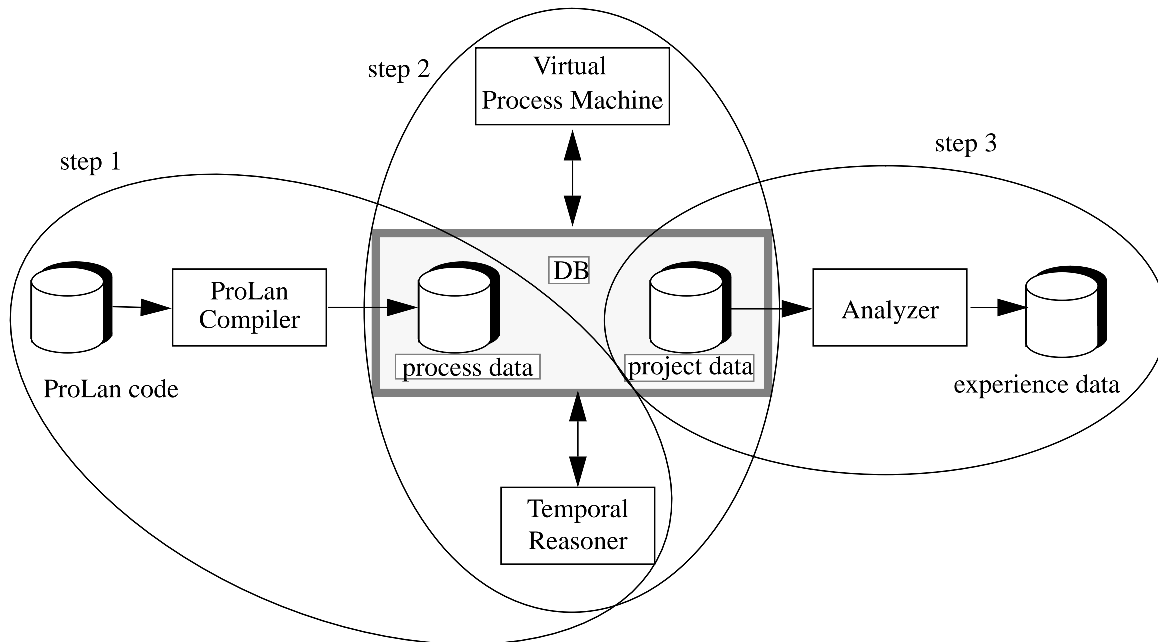

Fig. 5: Three steps of the software development process with the MoMo system

Step 2: Enacting a project

While enacting a project, instances of user classes are created, modified, or deleted. These instances - called project data - are the real SP elements. They represent actual entities or activities to be performed. The VPM evaluates the project data (the state of a project) to choose an object of a subclass of the system class ACTIVITY, to be interpreted. Thereby, other activities may be instantiated, entities are modified, or tools are invoked. The VPM may use the specified temporal information for scheduling activities (e.g., if the actual time is equal the specified latest start time of an activity, the priority of this activity is increased). On the other hand the VPM complements quantative temporal information (e.g., by assigning the value of the real start date of an activity).

The temporal reasoner checks the consistency of temporal dependencies between activities and solves conflicts when possible. Unresolvable conflicts require user interaction which can be supported effectively by identifying the reasons ([Weigel, Bleisinger 91]).

Errors or other events that enforce correction of previously specified SPs may occur from time to time. In such a case, the ProLan description must be altered and translated again (see step 1) before continuing process enactment.

During project enactment values of specified attributes and states of all instantiated activities are displayed to guide the project manager. The manager can use this information to control the running project, by selecting one alternative out of a set, presented by the MoMo system.

Step 3: Analyzing a project

After project termination the collected dynamic and flexible data are analyzed. The experiences gathered may be compared to other projects experiences.

The components of the MoMo system which form the basis for these tasks are the project data. These data can be used to organize an experience base. Especially, analyzing the actual temporal data of all activities performed during the project will lead to conclusions

about the suitability of the choosen software process model. As a result of this task reliable models (stored in the process database) may be directly reused in other projects.

5 Project Enactment with the MoMo system

In the previous chapter we described how to work with the MoMo system from a higher level of abstraction. Now we focus our attention to the things happening during step 2. We explain the meaning of “project enactment” and “interpreting an activity” in the context of our current implemented system (cf. [Verlage 91]).

Before a project is executed the infrastructure must be modelled. Using the compiled subclasses of the system class ENTITY the project’s staff must have a representation in the computer. Instances of entities must be created and related in an initial phase. For example, every workstation or programmer has a counterpart in the MoMo system. The initial state additionally contains all objects of the classes ASSERTION and GUARD.

Now the system is ready to execute the project. The project’s state is brought into another one by interpreting the action part of an activity. In general the system enacts a project executing the following steps:

- Evaluate the constraints of assertions; if an error occurs -> exception handling
- Match the guards with the events released during the last execution of an activity. If the guard reacts on an event execute the guard’s action part.
- Collect those activities, which preconditions are true
- Select one activity (with user’s help)
- Interpret this activity and continue with the first step

The system starts a new project by reacting on a user event (see Example 2) and instantiating one single activity – the activity which represents the software project. The whole knowledge of the project resides in this object. The loop of the above steps begins. Depending on the type of an activity (structured or elementary) the semantics of an interpretation differs. So we will look closer at this important step.

The refinement of a structured activity is done by the interpretation of its action part. The action part contains both, the description of the activity objects that have to be instantiated and the data bound to the formal parameters passed to these activities. By this way data and process model knowledge are bound to express project knowledge. One may illustrate the interpretation of a structured activity as the expansion of a macro. No knowledge is added, only the granularity of representation is refined. When the VPM interprets a compiled ProLan statement containing a name of an activity, an instance of this class is immediately created.

The execution of structured activities terminates not with the interpretation of their action part. Such an activity is active until the last of its subactivities is being executed. But this subactivities can be structured too. We repeat dividing (structured) software processes until we reach a level of elementary activities. So we get a tree expressing a hierarchy of abstraction of representations. The following figure shows a tree, which could express the results of the repeated interpretation of Example 2.

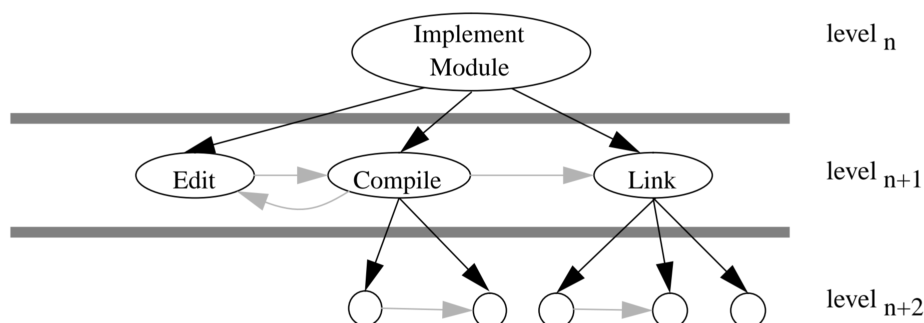

Fig. 6: Instantiation tree with several levels of activity abstraction

The leaves of the tree (e.g., activity “Edit” in Fig. 6) are elementary activities, all other objects are structured activities. The tree and subtrees are processes.

In the case of an elementary activity the following actions can take place:

- invocation of a tool
- invocation of a function
- creation of entities
- deletion of entities
- assignment of values to slots of entities or activities

After an elementary activity is interpreted and its postcondition is evaluated the VPM looks up, whether all subactivities of an activity are executed. In this case the execution of the activity on the next higher level is finished and now its postcondition has to be evaluated. A value FALSE expires an alarm to inform the user, but there is no automatic reaction by the system to repair the reason for the misbehaviour. When the last elementary activity is executed the project is finished.

This approach of enacting activities representing a software process leads to the following advantages:

First, every (sub) activity gets its own “substance” and identity, the instance itself. The “macro call” with local bindings is enclosed. The activity gets public for further examinations and the instance can be included in expressions. In our current implementation activities and the other process elements are general objects in the sense of Common LISP. This guarantees the simple expansion of the MoMo system. Components can be added and observe the objects without knowing the implementational details of other components, and all operate on a common layer.

Second, the moments of an activity’s creation (“macro call”) and its execution do not fall together. It is not a usual “call” in common sense, but more the declaration of an intention to execute this activity. This space of time (the dashed triangle in Fig. 7) can be used, e.g., to allocate resources for the instance or to check temporal relations. The instances can be integrated in several plans.

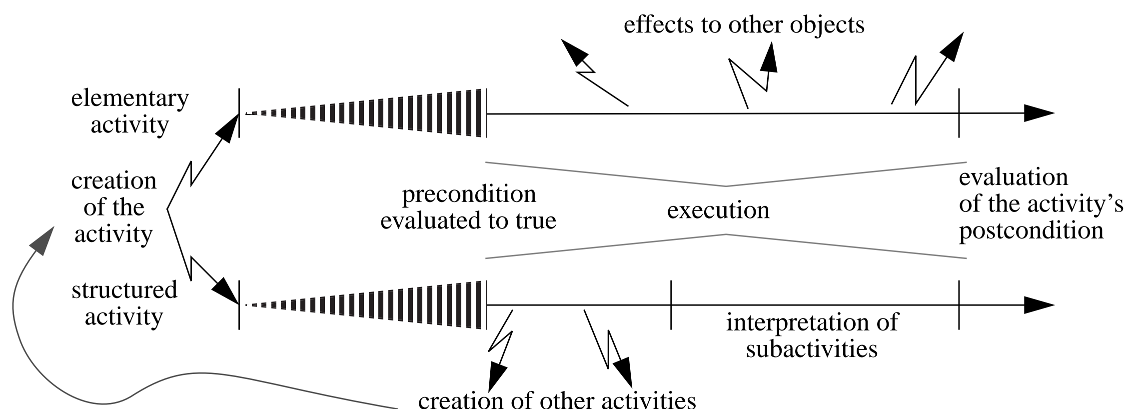

Fig. 7: Lifetime of an activity

Third, an object itself does not know how it is related to other activities (e.g., the activity “Compile” has no information about the restriction of the process model that the activity “Edit” has to be executed first). Remember, dependencies between activities belonging the order of execution (grey arrows in the figure, constructs like SEQUENCE or REPEAT ... UNTIL in ProLan) at a specific level ($level_{n+1}$) are expressed explicitly by the aggregate activity at the next higher level of abstraction ($level_n$); additional temporal information is inferenced by the temporal reasoner. Changing an activity’s class will only touch the object itself and its subtree.

Fourth, it is impossible that two activities in distinct trees, which represent two distinct projects, are related together according to the order of execution. Also activities have no

relation, when their ProLan representations are enclosed by a SET constructor (rightmost leaves in the tree of Fig. 6). This enables the system to swap subtrees easily.

Finally, resulting out of the points above, one can discuss a project on different levels of abstraction. For example, the project manager ist interested in monitoring the higher levels, on the other hand a programmer watches only the lower levels in a specific subtree.

6 Related Work

Since [Osterweil 87] stated the importance of formalizing the software development process other research projects (e.g. Arcadia [Taylor et al. 89]) are concerned with the inclusion of SP models into software development environments. The process programming approach was one of the first to describe SPs explicitly. The crucial points of the research efforts comprise the development of suitable process modelling languages, models for the enaction of software processes and architectures for process environments.

With the process programming approach [Osterweil 87] software engineers will describe only well-known, idealized procedures, that may not map accurately to actual development behavior (cf. [Curtis et al. 87]). The imperative specification of process programs is not expressive enough. Other approaches ([Ramanathan, Sarkar 88], [Williams 88], [Deiters et al. 89]) allow for a more flexible description for the enaction of software processes. With these models, the users can formally describe the behavior of SPs, which is achieved by stating pre- and postconditions for the project activities. [Williams 88] provides a simple but powerful mechanism for specifying project activities. As this model focusses on activities, appropriate representations for other elements (e.g. entities) of a realistic software process are missed. The conceptual modelling language (CML, [Ramanathan, Sarkar 88]) and the model for software processes (MSP, [Deiters et al. 89]) are examples for object oriented approaches for the description of SPs. This reflects the high degree of flexibility and dynamics typical during the enaction of software processes. CML provides rules bound to passive objects which determine how to react on operations applied to them. But there is no language feature to express constraints concerning more than one object. MSP supports the users in a modular and incremental development of SPs. Because of the consistency guarantees for the extended Petri Net representation of activities, the user is forced to specify runtime modifications of the model in advance.

None of the mentioned work provides any means for integrating temporal aspects into the process model. Only the description of concurrency (overlapping of time intervals) is usually supported implicitly. Like the SP itself, i.e. its elements, we also want to be able to specify temporal relationships explicitly (e.g. perform design before implementation and start implementation at least at the 1.5.91). Moreover, we want our temporal appointments guaranteed to be consistent, or even atomatically completed and infered more precisely.

7 Conclusion and Future Work

To demonstrate the soundness of the MoMo model several medium sized SPs are described and enacted within the MoMo system (cf. [Ridder 91]). For instance, a MVP-L process description (cf. [Lott, Rombach 90]) is translated to ProLan and subsequently enacted within the MoMo system. So, we can summarize that our MoMo meta model provides a common layer to compare processes, to reuse suitable process concepts, or to combine different SP models. The meta model allows for specifying individual SP models by formally describing all relevant elements and activities and their relationships.The underlying object oriented approach yields an adequate modelling of the dynamic and flexible properties of SPs. Here, classes of entities, activities to be performed during a project, and intervals containing temporal data and relations are specified. Individual

entities, activities, and associated intervals are instantiated while enacting the selected SP model. In particular, we are able to modify the actual SP during runtime.

The formalisms of our meta model also allows for specification of the SP model, both descriptive and prescriptive. In the former case, the user specifies the behavior of the model by giving the pre- and postconditions of the activities; this results in a nondeterministic control of activities. In the latter case, the user specifies the execution of the SP by using special control constructs. [Ridder 91] gives some examples expressing the same process model with different kinds of ProLan's description constructs.

Aggregation mechanisms allow for the specification of activities and entities at different levels of abstraction. These serve as a communication interface for the heterogenous group of "project people".

An introduced method of describing systems is the refinement by less complex subsystems. Thus, a project manager may describe the process first at a very high level of abstraction for testing the choosen process model in general, ignoring unimportant details. Because of the ability to modify the process description dynamically he may decide to specify the process more precisely when the enactment of the process proceeds. So the MoMo model combines both, so-called process programming in the large and process programming in the small. This feature allows for reacting on requirements become known while enacting a project. A dynamic, incremental generation of code is provided as shown in Chapter Architecture of the MoMo System on page 2.

Common aspects of different processes may be expressed with the help of classes. The same descriptions are used in different environments. In this way MoMo transfers knowledge between projects and compares them on several levels of abstraction.

We do not proclaim MoMo as a system to support the enforcement of only separated projects. Rather, with the help of the features discussed a recognition of the (implicit) used process models and the development of their description by the user can be supported.

Yet the main components of the MoMo system are implemented and integrated to a first prototype of the MoMo system as sketched in Chapter Architecture of the MoMo System on page 2. The temporal reasoner is developed separately. We are now undergoing to integrate it in the system. We will profitate of our object oriented implementation language CLOS which allows for integration of the temporal reasoner more by extending the existing system than by modifying a lot of actually implementation – an additional experience about the suitability of the object oriented approach in general.

There are some points of further research :

The MoMo system is implemented as a single user system. Currently, we extend it to a distributed multi user system based on simple assumptions about the kind of distribution. These efforts must be completed by investigating special (ProLan-) language constructs for the explicit description of distribution.

Until now we have some preliminary experience with the simulation of small or medium sized software projects. To get more reliable information about the suitability of the MoMo model we need a field study comprising several large software projects.

Other work may focus on planning software processes considering causal reasoning about the dependencies of the process' activities.

References:

[Bleisinger 91] Rainer Bleisinger: *TEMPO - ein integrierter Ansatz zur Modellierung qualitativer und quantitativer zeitlicher Informationen*, Proc. of 15th GWAI ,1991, pp 167-176.

[Bleisinger et al. 91] Rainer Bleisinger, Peter Knauber, Wolfgang Schramm, Martin Verlage: *Software Process Description and Enactment Considering Temporal Constraints*, Proc. of the IV. Int. Symposium on Artificial Intelligence, Cancun, Mexico, Nov. 1991.

[Curtis et al. 87] B. Curtis, H. Krasner, V. Shen, N. Iscoe*: On Building Software Process Models under the Lamppost*, Proc. of the 9th Int. Conf. on SE, Monterey CA, April 1987.

[Deiters et al. 89] Wolfgang Deiters, V. Gruhn, W. Schäfer: *Systematic Development of Formal Software Process Models*, 2nd Europ. SE Conf. 89, Springer Verlag Heidelberg.

[Faidt et al. 89] Klaus Faidt, Stephan Flohr, Rainer Bleisinger: *Representation und Verarbeitung von zeitlichem Wissen*, Proc. of 5th OEGAI, 1989, pp 303-312.

[Humphrey, Kellner 89]Watts S. Humphrey, Marc I. Kellner: *Software Process Modeling: Principles of Entity Process Model,* Proc. 11th ICSE, April 1989.

[Knauber 91] Peter Knauber: *Entwurf und Implementierung einer Beschreibungssprache für den Software-Entwicklungsprozeß*, Master Thesis, Univ. of Kaiserslautern, 1991.

[Lott, Rombach 90] Christopher M. Lott, H. Dieter Rombach : *MVP-L1 Solution for the Software-Pocess Modeling Problem*, 6th Int. Workshop on Software Process Modeling, Hakodate, Japan, October 1990.

[Meyer 88] Bertrand Meyer: *Object-oriented Software Construction*, Prentice Hall, 1988.

[Osterweil 87] Leon Osterweil: *Software Processes are Software too*, Proc. of the 9th Int. Conference on SE, Monterey California, April 1987.

[Ramanathan, Sarkar 87] J. Ramanathan, S. Sarkar: *Providing Customized Assistance for Software Lifecycle Approaches*, IEEE Transactions on SE, Vol 14 No. 6, June 1988.

[Ridder 91] Michael Ridder: *Beschreibung ausgewählter Software-Prozesse mit ProLan*, Bachelor Thesis, University of Kaiserslautern, 1991.

[Royce 70] W.W. Royce: *Managing the Development of large Software Systems, Concepts and Techniques*, Proceedings WESCON, August 1970.

[Schramm 91] Wolfgang Schramm: *Ein objektorientiertes Metamodell für die Software-Entwicklung*, Ph. D. Thesis, University of Kaiserslautern, 1991.

[Steele 90] Guy L. Steele jr.: *Common Lisp The Language*, 2nd Edition, Digital Press, 1990

[Taylor et al. 89] R.W. Taylor et al.: *Foundations for the Arcadia Environment Architecture*, ACM SigPlan Notices, Vol 21 No. 2, February 1989, pp 1-15.

[Verlage 91] Martin Verlage: *Entwurf und Implementierung eines Projektinterpreters*, Master Thesis, University of Kaiserslautern, 1991.

[Weigel, Bleisinger 91]Achim Weigel, Rainer Bleisinger: *Detection of Elementary Reasons for Contradictions in Time Interval Networks,* submitted to Int. Conf. on Fifth Generation Computer Systems, Tokyo, 1992.

[Williams 88] Lloyd G. Williams: *Software Process Modeling: A Behavioral Approach*, Proc. of the 10th ICSE, Singapore, April 1988.