SOFTWARE PROCESS DESCRIPTION AND ENACTMENT CONSIDERING TEMPORAL CONSTRAINTS

Rainer Bleisinger
German Research Center
for Artificial Intelligence (DFKI)
P. O. Box 20 80, D-6750 Kaiserslautern
e-mail: bleising@dfki.uni-kl.de

Peter Knauber Wolfgang Schramm Martin Verlage
Dept. of Computer Science,
University of Kaiserslautern
P. O. Box 30 49, D-6750 Kaiserslautern
e-mail: schramm@informatik.uni-kl.de

Abstract

The complex task of software development require automatic support. Therefore, we developed the object oriented meta model MoMo to represent the subjects of software development processes including temporal aspects. Based on this meta model several components (compiler, virtual process machine, temporal reasoner, analyzer) are combined to the MoMo system. Our project description language allows for an explicit and formal software process description on the basis of the MoMo model. The correctness of these descriptions are inspected by the compiler and the temporal reasoner. Subsequently, the executable form will be enacted by the MoMo virtual process machine. During the enactment, the software process description can be modified, especially temporal information can be added or made more precisely. In this context the temporal reasoning component guarantees the temporal consistency of the software process. Moreover, the reasoner itself infers temporal data more precisely. Finally, the analyzer evaluates the project actually performed for gathering information to be applied to future projects.

1 Introduction

The high degree of complexity and huge amount of data in the context of software development 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 a 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. The treatment of temporal knowledge about SPs is worthwhile for a closer examination, in particular to support project managers in their management task.

In this paper, first we give in chapter two an overview of the MoMo system architecture and its particular components. According to the system architecture we describe in the following chapters three important parts: modeling of SPs, reasoning about temporal knowledge of SPs, and the enactment of SPs.

In the third chapter we discuss the main properties of MoMo meta model which provides a common layer for describing any SP model. Therefore, we present the basic formal constructs and give examples using the project description language ProLan.

MoMo provides special features to describe various kinds of temporal constraints for processes. The temporal reasoner, called TEMPO, is introduced in chapter four. We identified several types of temporal knowledge associated with SPs. For each type and the relationships, we exemplary show how to establish consistency and propagation conditions, and we shortly explain the ideas for the used algorithms.

We designed MoMo as an environment comprising all jobs concerned with SPs. In chapter five we give a brief overview how to work with our system.

Finally, we make some remarks about related research activities and conclude with a summary of the advantages of the MoMo approach.

2 Architecture of the MoMo System

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

We have identified five major components for the implemented MoMo system (Fig. 1):

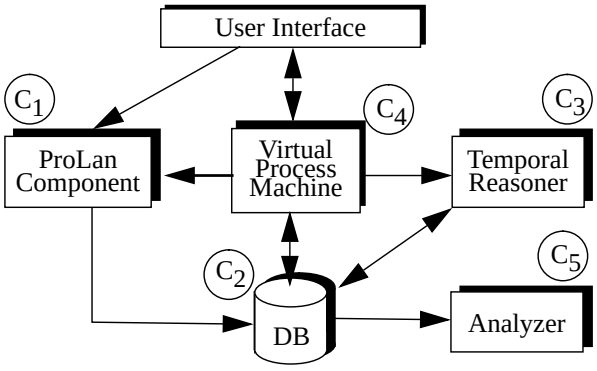

Fig. 1: The architecture of the MoMo system

(C_1) Editor and compiler for the process description language build the *ProLan component*. ProLan must be very expressive to allow for the description of

all kinds of process models. The compiler has to transform this description into an executable program.

(C_2) The so called *Software Engineering DB* holds project specific data while the project is in enaction. The DB serves as a common buffer for the various system components.

(C_3)The *temporal reasoner* checks the organizational dependencies between the activities. Temporal inconsistencies invoke exception handling, which either enforces the correction of ProLan code or the revision of the projects temporal specifications.

(C_4)The *virtual process machine* (VPM), as the central process enacting mechanism coordinates all the SP objects and executes the processes. While interpreting the description the VPM may cause the user to modify or complete the SP description.

(C_5)The process *analyzing component* evaluates the project data to get hints about the suitability of the applied SP model or about the process task complexity. The extracted data must allow for comparison with experiences of former projects.

3 The MoMo Model

First, we introduce an appropriate meta model - called MoMo - for the description of software development processes ([Schramm 91]). Osterweil stated that the descriptions of SPs are some kind of program, like application programs. Instead of a static meta model description ([Osterweil 87]) we consider a model relying on the object oriented paradigm to be adequate to represent the complex subject of SP modeling ([Meyer 88]). Moreover, MoMo is a behavioral model ([Williams 88]), because the descriptions of the activities focus more on the effects which the activities produce rather 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 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. Every component of a software project corresponds to an object in our model. The number of elements in SPs is unlimited growing and shrinking over the lifetime of the process. Thereby, the quantity of different kinds of elements is restricted. So, we describe the different kinds of the SP elements as classes. These classes again are subclasses of a couple of system classes which determine the fundamental semantics of the user defined classes (Fig. 2).

With respect to the complexity of the SP modeling task, we have developed a three level knowledge representation for SP descriptions:

Level one: System classes represent our *meta knowledge* about SPs 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 ProLan ([Knauber 91]) using the system classes.

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

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.

Different types of relations can be defined between the classes. All SP objects are considered to be objects in the sense of the object oriented paradigm. The different representation of the classes is caused by the different use of their instances. For example, opposed to the longer life time of entities activities are used for determination of the project’s schedule.

Now, we discuss the predefined system classes which are essential to compose the MoMo model.

System Class ENTITY

An entity represents a physical or logical aspect of the real world (programs, documents, persons). It

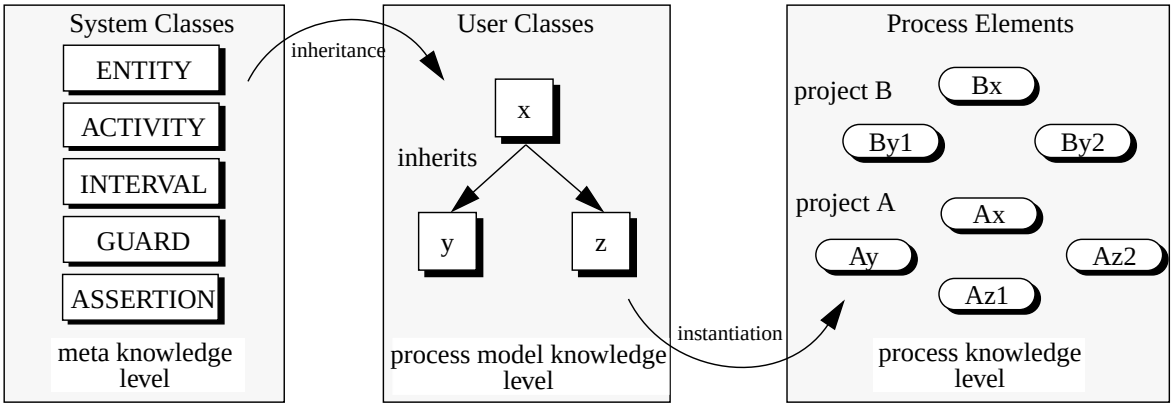

Fig. 2: The three knowledge representation levels

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 entities of lower abstraction levels.

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

Example:

```
ENTITY Program;
  ATTRIBUTES: Name: STRING INIT "NoName";
              Lines-of-code := L_o_C (Name);
              Modules -> (Module);
                  -- Module is another Entity
END_ENTITY Program;
```

System Class ACTIVITY

Activities are the most important elements 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. An activity consist 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
The state that enables the execution of activities, expressing 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 subactivities; 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 both, in a prescriptive (SEQUENCE, SELECTION, CASCADE, REPEAT) and in a descriptive way (SET). The execution order of descriptively specified subactivities is determined at SP runtime.

Constructors often specify temporal dependencies implicitly, e.g. the SEQUENCE constructor specifies a strictly sequential order of activities. To make these implicit information accessible for the temporal reasoner, the ProLan compiler must transfer the information into an explicit representation.

(4)Postcondition
Description of the state after the execution of the activity, assumed to be valid.

Typical for activities are their start and finish times; 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. If there are independent activities, the user may decide the temporal order of enactment of activities. Therefore, our activity concept must provide means for the specification of temporal information.

Example:

```
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 (Fig. 3).

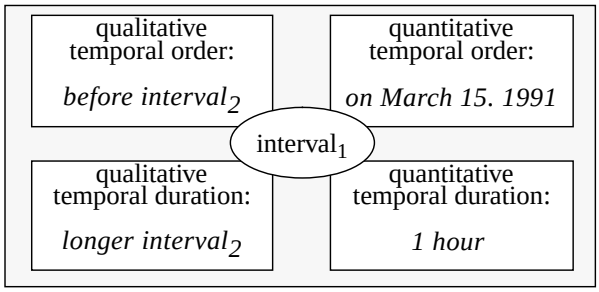

Fig. 3: Temporal knowledge types for intervals

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 (e. g., possible minimal and maximal durations, actual and average durations, earliest and latest possible start and end times, actual start and end times).

Example:

```
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:
```

Minimal_Duration: (2 months);
QUAL_DURATION: (longer)(Edit, Compile, Link);
END_INTERVAL Implement_Module_Interval;

Each activity class is related to exactly one interval class. If for an activity no temporal knowledge is explicitly specified, the associated interval class is supplied with default values of INTERVAL.

System Class GUARD

Guards wait for the occurrence of certain events (e.g. creation of an entity, expiry of a deadline). If an event has occurred the guard invokes the associated activities (causes its instantiation) , 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:

ACTIVITY_GUARD Start-Project;
ON CREATE-EVENT (Project-Entity)
DO Perform-Project (ACTIVATOR);
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:

ASSERTION Control-Programmers;
UNLESS NUMBER-OF-WHERE
(Workstation, NOT busy) < 10
DO EXECUTE:Warning (“programmers lazy!”);
END_ASSERTION Control-Programmers;

4 TEMPO - the MoMo System Temporal Reasoner

In the MoMo system both, the process model knowledge level and the process knowledge level are of interest for checking temporal specifications (Fig. 2). The code generated by the ProLan compiler - the dynamic data - builds the input for TEMPO. On the model level which looks like a skeleton plan, the predefined temporal information is verified. While enacting the SP, TEMPO is invoked by the runtime system either after a user has specified additional temporal information or the runtime system itself has updated the instances.

Because we defined the system class INTERVAL 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.

On both levels the following tasks are necessary:

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

The different temporal knowledge types enforce several techniques for performing all tasks. Moreover, the relationships of these types must also be taken into account [Flohr et al. 88]. So, three levels of temporal reasoning are identified (see Fig. 4):

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

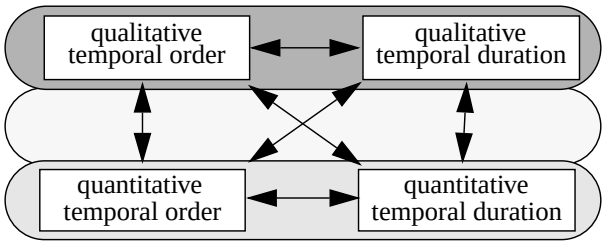

Fig. 4: Relations between temporal knowledge types.

The underlying temporal model contains two layers: the declarative representation layer and the execution layer [Bleisinger 91]. 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 excution layer.

In the following we shortly explain the three temporal reasoning levels focussing on the central issues.

Qualitative Temporal Reasoning

The treatment of qualitative temporal order is based on the allowed value set of 13 elementary temporal order relations (e. g., before, overlaps, started-by) and the consistency and propagation conditions represented by a transitivity table introduced by [Allen 83]. Unexact and vague temporal order is expressed by a set of mutually exclusive elementary relations. Belonging to this, the integrated propagation and consistency check algorithm developed by [Allen 83] and the extended algorithm of [van Beek 90] are taken. Both algorithms guarantee only local network consistency. Thus, we additionally use the algorithm of [Valdez-Peres 87] which checks global network consistency.

In analogy, we have defined a set of three elementary temporal duration relations (longer, equal, shorter) and the appropriate transitivity table. Due to this approach, we can reuse the algorithms specified for qualitative temporal order without any extention of the execution layer [Bleisinger 91].

Temporal order and duration relations are bidirectionally correlated to each other. Based on specified order relations between two intervals, the stated duration relations for these intervals are examined and vice versa. For each direction, consistency and propagation conditions are formulated; for example:

$interval_1$ (contains) $interval_2$
-> $interval_1$ (longer) $interval_2$

$interval_1$ (shorter) $interval_2$
-> $interval_1$ (before, meets, overlaps, starts, finishes, during, overlapped-by, met-by, after) $interval_2$.

An algorithm is implemented which checks the relational consistency and, thereby, infers new relations between two intervals or restrict the previous defined set of relations using the former descriptions.

Quantitative Temporal Reasoning

On this level, all sets of valid data are defined by the specification of a calendary-chronological system (CCS), e.g., the Gregorian Calender extended by todays clock time system, or a working calendar with five day week and three shifts a day. To handle different kinds of CCSs in a declarative way, we elaborated common essential criteria of CCSs. These serve as a uniform and abstract CCS model [Faidt et al. 89]. So, each user can define within the model his/her own appropriate calendar.

Additionally, the particular items for quantitative temporal order and duration are defined (e. g., earliest_start, observed_start, latest_end, minimal_duration, average_duration). According to this definition the consistency conditions (CCs) and propagation conditions (PCs) are specified. [Huber 90] also treats different items for quantitative temporal knowledge.

In the following examples the subscripts I denotes an interval, old and new depict the value currently specified, resp. the new value which may be infered.

(CC) $earliest\_start_I$ (before, meets, equal) $latest\_start_I$

(CC) $minimal\_duration_I$
(shorter, equal) $maximal\_duration_I$

(PC) $_{old}latest\_start_I$
{equal, met-by, after} $_{new}latest\_start_I$

(PC) $_{new}maximal\_duration_I$
(shorter) $_{old}maximal\_duration_I$

To treat the relationships between quantitative temporal order and duration, the focus of attention is of relevance. On the one hand, duration specifications are “added” to or “subtracted” from particular order specifications to obtain other order specifications. This computed values must fit special CCs, e.g.

(CC) $earliest\_end_I$ (after, met-by, equal)
($earliest\_start_I$ “add” $minimal\_duration_I$),

(CC) $earliest\_start_I$ (after, met-by, equal)
($earliest\_end_I$ “subtract” $maximal\_duration_I$).

On the other hand, the order specifications are “subtracted” to get values for different temporal durations which also must fit CCs, e.g.

(CC) $minimal\_duration_I$ (longer, equal)
($earliest\_end_I$ “subtract” $latest\_start_I$).

For consistency checking, each temporal knowledge declaration of the SP is compared with the appropriate CCs. Detected inconsistencies are tried to resolve by propagation according to the suitable PCs. If this is impossible an inconsistency is announced and a user interaction for correction is forced.

Combination of Qualitative and Quantitative Temporal Reasoning

The intermediate level in Fig. 4 integrates the qualitative and quantitative temporal knowledge levels. First ideas for combination of qualitative and quantitative temporal knowledge are proposed by [Vere 83]. In our model four combinations are identified, but for the combination of qualitative duration and quantitative order no meaningful conditions can be specified. Each one of the rest is treated in the same manner like above. Several conditions considering different points of view and unexactness are defined building the basis for consistency checking and propagation. Now, we give some condition examples:

- qualitative order / quantitative order
  I_1 (starts) I_2 -> $latest\_start_{I1}$ (equal) $latest\_start_{I2}$
  $latest\_start_{I1}$ (before) $earliest\_start_{I2}$
  -> I_1 (before) I_2
- qualitative order / quantitative duration
  I_1 (starts) I_2 -> $maximal\_duration_{I1}$ (shorter) $minimal\_duration_{I2}$
  $minimal\_duration_{I1}$ (longer) $maximal\_duration_{I2}$
  -> I_1 (started-by, contains, finished-by) I_2
- qualitative duration / quantitative duration
  I_1 (shorter, equal) I_2 -> $minimal\_duration_{I1}$ (shorter, equal) $maximal\_duration_{I2}$
  $minimal\_duration_{I1}$ (equal) $maximal\_duration_{I2}$
  -> I_1 (longer, equal) I_2

It is important to notice that in the actual implementation the order of application of the different algorithms is static. A dynamic strategie for the application order will be elaborated. Thereby, the actual temporal knowledge is considered to choose the most suitable algorithm first. This leads to an opportunistic strategie with an appropriat, efficient execution order.

5 Working with the MoMo system

Applying the realized prototype of the MoMo system, an “intelligent” support of process management could happen. For that purpose, the following three steps are performed (Fig. 5).

Step 1:Planning a project - describing a SP model

A ProLan project description is primarily a skeleton plan of the SP, the SP elements are described in terms of classes. ProLan also provides several constructs to express causal and temporal relationships between project activities.

The ProLan compiler tests the correctness 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 generated code is stored in the dynamic database. These data is denoted as dynamic data, because it can be changed dynamically during process runtime.

Step 2:Enacting a project

While enacting a project ([Verlage 91]), instances of user classes (dynamic data) are created, modified, or deleted. These instances - called flexible data - are the real SP elements. They represent actual entities or

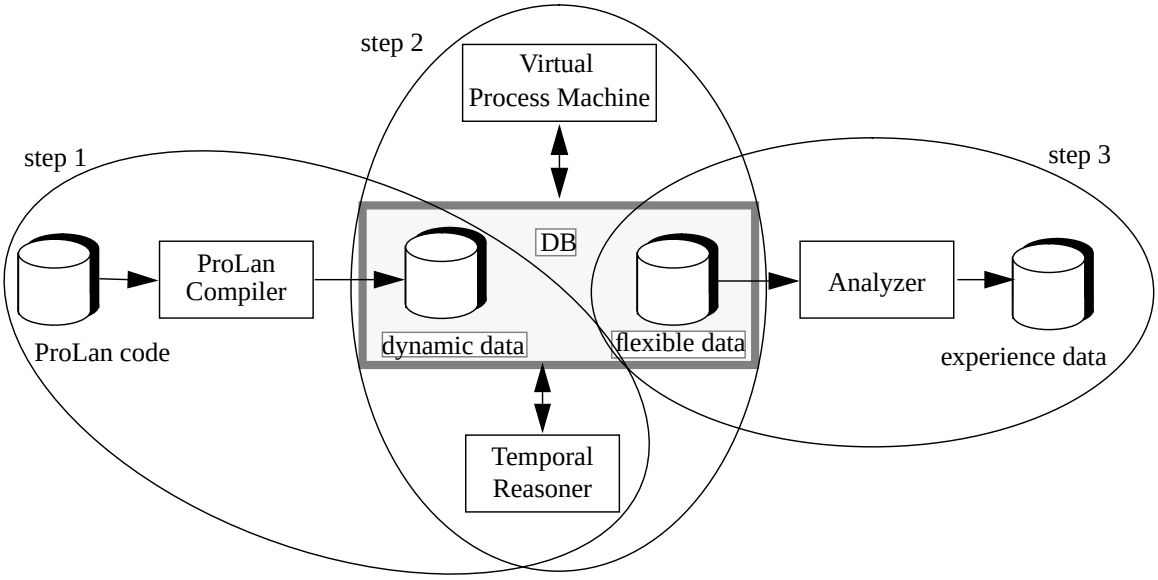

Fig. 5: Three steps of the software development process with the MoMo system

activities to be performed. The virtual process machine (VPM) evaluates the flexible data (the state of a project) to choose an activity for interpretation. Thereby, other activities may be instantiated, entities are modified, or tools are invoked. An activity is executed only when its precondition is true. Specified temporal information is used for scheduling activities (e.g., if the actual time is equal the specified latest start time, an activity's priority is increased). Additionally, the VPM complements quantative temporal information (e.g., by assigning the real start date).

During SP enactment assertions watch the state of the project. Like guards, assertions cause special activities to bring the SP back to the right way.

TEMPO checks temporal consistency between activities and solves conflicts when possible. Unresolvable conflicts require user interaction.

Sometimes, errors or other events enforce the correction of specified SP. In such a case, the ProLan description must be altered and translated again (see step 1) before continuing process enactment.

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 flexible 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 SP model. As a result of this task reliable models (stored in the dynamic DB) may be directly reused in other projects.

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 modeling languages, models for the enaction of SPs 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 ([Curtis et al. 87]). The imperative specification of process programs is not expressive enough. Other approaches allow for a more flexible description for the enaction of SPs. With these models, users can formally describe the behavior of SPs 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 SP are missed. The conceptual modeling language (CML, [Ramanathan, Sarkar 88]) and the model for SPs (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 SPs. 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 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

specify temporal relationships explicitly (e.g. perform design before implementation, 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 Conclusions

With our MoMo meta model we provide 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, activities and their relationships. The underlying object oriented approach yields an adequate modeling of the dynamic and flexible properties of SPs. Classes of entities, activities to be performed during a project, and intervals containing temporal data and relations are specified.

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".

Within realistic SPs, time constraints must be expressible. Because the activities and the resources must be planned, we provide a suitable representation of time and a calendary chronological system in MoMo. On this foundation, TEMPO supports project planning at the various knowledge representation levels of the MoMo model. At the process model level, strategic model dependend temporal knowledge is elaborated. At the process level, individual temporal interval networks are processed.

We identified different types of temporal knowlegde, pointed out their corelations, and established three levels of temporal reasoning. Based on temporal constraint conditions TEMPO guarantees the consistency of the specified temporal information. Moreover, TEMPO infers temporal specifications according to propagation conditions to achieve more detailed information.

In MoMo, there are two kinds of controlling execution order of activities. First, the user is able to implement well known structures of SPs with the help of simple - in relation to the complexity of the SP management task itself - constructs. Second, a nondeterministic control of activities is expressed by pre- and postconditions.

An introduced method of describing systems is the refinement by less complex subsystems. With support by the MoMo system the user states, modifies, or imports subsystems at runtime. She/he need not to specify the whole SP at the early beginning of a project. This feature allows for reacting on requirements become known while enacting a project. A dynamic, incremetal generation of code is provided.

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.

8 REFERENCES:

[Allen 83] J. F. Allen: *Maintaining Knowledge about Temporal Intervals*, Comm. of the ACM, 26, No. 11, Nov. 1983, pp 832-843.

[Bleisinger 91] R.Bleisinger: *TEMPO - ein integrierter Ansatz zur Modellierung qualitativer und quantitativer zeitlicher Informationen*, accepted on GWAI-91.

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

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

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

[Flohr et al. 88] S. Flohr, R. Bleisinger, K. Faidt: *Verarbeitung von zeitlichem Wissen*, WISDOM-Research Report FB-AGB-88-26, December 1988.

[Huber 90] A. Huber: *Wissensbasierte Überwachung und Planung in der Fertigung*, Schmidt Verlag, 1990.

[Humphrey, Kellner 89] W. S. Humphrey, M. I. Kellner: *Software Process Modeling: Principles of Entity Process Model*, Proc. 11th IC SE, 1989

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

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

[Osterweil 87] L. Osterweil: *Software Processes are Software too*, Proc. 9th ICSE, Monterey, April 1987

[Ramanathan, Sarkar 87] J. Ramanathan, S. Sarkar: *Providing Customized Assistance for Software Lifecycle Approaches*, IEEE Trans. on SE, (14,6), 1988

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

[Schramm 91] W. Schramm: *Entwurf und Implementierung eines objekt-orientierten Metamodells zur Beschreibung von Software-Entwicklungsprozessen*, Ph. D. Thesis, Univ. Kaiserslautern, 1991 (to appear)

[Taylor et al. 89] R.W. Taylor et al.: *Foundations for the Arcadia Environment Architecture*, ACM SigPlan Notices, (21,2), February 1989, pp. 1-15

[Valdez-Peres 87] R. E. Valez-Peres: *The Satisfiability of Temporal Constraint Networks*, Proc. of 6th AAAI, 1987, pp 256-260.

[van Beek 90] P. van Beek: *Reasoning about Qualitative Temporal Information*, Proc. of 8th AAAI, 1990, pp 728-734.

[Vere 83] S. A. Vere: *Planning in Time: Windows and Durations for Activities and Goals*, IEEE Trans. on PAMI, (5,3), 1983, pp 246-267.

[Verlage 91] M. Verlage: *Entwurf und Implementierung eines Projektinterpreters*, Diploma Thesis, Univ. Kaiserslautern, 1991

[Williams 88] L. G. Williams: *Software Process Modeling: A Behavioral Approach*, Proc. 10th IC SE, Singapore, April 1988