Just-in-Time Initialization of Objects Representing Software Processes

Martin Verlage & Peter Knauber

Fachbereich Informatik
Universität Kaiserslautern

Abstract

Software development processes and their results have to fulfill a number of non-orthogonal criteria. The processes are characterized by a large number of highly interrelated tasks and data. Object oriented modeling seems to be an adequate way making these processes more manageable. Relying on the object oriented meta model MoMo the elements and tasks of such processes can be treated in a more natural way. But using object oriented techniques to form process descriptions brings up problems during initialization of objects. We will show that an object's creation must differ from common semantics, especially its initialization will be performed just-in-time. By this way, one is able to make forecasts about project steps.

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.

Various models have been designed to guide the users through the software development task; the waterfall model [5] is one of the best known. However, most software development processes used in practice are only implicitly defined. To be able to understand, to compare, to support or to improve software development processes, we have to define them explicitly. Therefore, we use the meta model MoMo that enables us to specify different software process models with an unique mechanism. Object orientation is brought into process programming.

In this paper, we give in Chapter 2 a brief overview of the object oriented MoMo Model. The model serves as the basis for the enactment of software development processes.

In the third chapter it will be shown how a software project is enacted with the MoMo system. Special objects describe the proceeding of a project. Moreover, the object oriented approach allows for an adequate planning about the project steps.

But Chapter 4 shows we have to use a special mechanism to initialize the objects representing software processes when we want to get the benefits. The special circumstances of project enactment make it possible to split the instantiation of an object into distinct phases.

The MoMo Model

Osterweil stated that the descriptions of software development processes are some kind of program [4]. One is able to describe software development processes with an algorithmic formalism. But the processes have some characteristics which violate the soundness of "normal" procedural programming. We consider a model relying on the object oriented paradigm to be adequate to represent the complex subject of software process modeling.

So we have chosen the object oriented meta model MoMo. It was developed from W. Schramm to give a unique mechanism for describing different software process models (cf. [6], [7]). All elements of software processes are viewed as objects. A wide range of structures and dependencies between them is represented by the system. The MoMo model has a three level knowledge representation, organized in classes and objects, for software process descriptions:

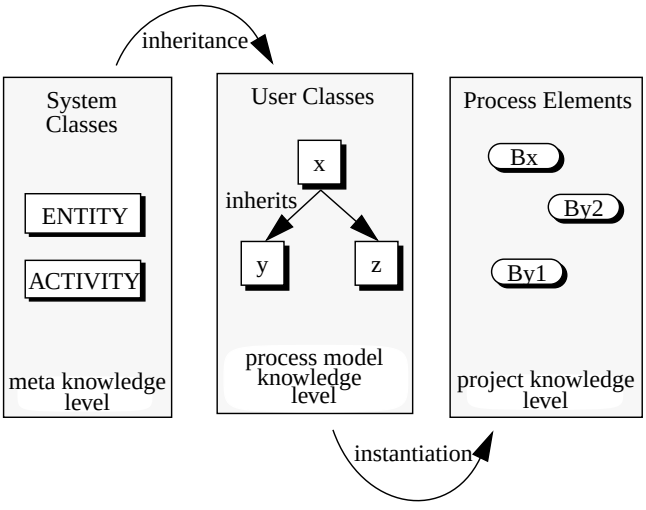

Fig. 1: Three level knowledge representation

The *meta knowledge* level represents a special view of general types needed to describe software process models. To use terms of object oriented programming this level contains the system classes.

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 with the project description language ProLan (cf. [3]).

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

Different types of relations can be defined between the objects and between the classes. 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. In the following chapters we will concentrate on the two main system classes ACTIVITY and ENTITY. The MoMo model contains three more system classes for temporal aspects (i.e. INTERVAL), for state based program flow (i.e. ASSERTION) and for event based program flow (i.e. GUARD). They are in this paper of lower interest and are described in [1].

Objects of the system class ENTITY represent physical or logical aspects of the real world (e.g. programs, documents, persons). They contain 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: Declarative, operational, structural and relational attributes which are shown in Example 1.

Example 1**:**

```
ENTITY Module-Document;
  INHERIT FROM Document;
  ATTRIBUTES:
    Design-Start    :   TIME INIT ACTUAL-DATE();
    Needed-Time     :=  ACTUAL-DATE() - Design-Start;
    Specifications  =   ( Specification-Document );
    Design          ->  Design-Document;
    ...
    Status          :   SYMBOL INIT undefined;
END_ENTITY Module-Document;
```

Activities, that means objects of the system class ACTIVITY, are the most important units of the software development process. 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 consists of the following parts:

(1) Attributes
The same kind of attributes as specified for entities, except the structural attributes, are permissible.

(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.

(4) Postcondition
Description of the state after the execution of the activity, assumed to be valid. In combination with the precondition, the behaviour of the activity can be modelled [10].

This system classes serve as templates for all user classes (cf. Fig. 1). The instances of these classes represent the state of a process, or short the process itself. We are able to proceed a project by interpreting activities. So let us take a closer look at this step of project support.

Enacting a Project

When we want to enact a software development process, we first have to model the involved infrastructure. Therefore we describe every type of resource of the real world as a user class to represent it in the model world. For example, every workstation or programmer has a counterpart in our implemented system or it is expressed which software engineer has which skills.

After this initial phase the system is ready to start the enactment of the project. It begins a new project by reacting on a user event and instantiating one single activity - the activity which represents the whole software development process. All knowledge about the project resides in this object.

We will now show how the project's state is brought into another one by interpreting the action part of an activity, depending on the mechanism of our current implemented system (cf. [9]). Project enactment is a looping execution of action parts:

- Collect those activities whose preconditions are true
- Select one activity (with user's help)
- Interpret this activity
  (Maybe instances of subactivities will be created;
  this will lead us to a special problem.)
- Evaluate the postcondition and display the value to the user
- Continue with the first step

ProLan statements contained in the object's action part are translated into an internal representation. An activity is executed by interpreting this internal code. Statements describe not only the modification of entities or invocation of functions and tools, moreover the activity can be refined into subactivities to delegate tasks. In an initial phase of the interpretation one single object is created to represent the task to be performed for every appearing (sub-) activity name. But not only the tasks are refined, also data is split and assigned to the objects. By this way, data and process model knowledge are bound to express process knowledge. One may illustrate the interpretation of such a ProLan statement as the expansion of a macro. No knowledge is added, only the granularity of representation is refined. We get a tree of objects which represents a real world project step, or process. For instance, regard the activity *Implement_Module*:

Example 2**:**

```
ACTIVITY Implement-Module;
  ATTRIBUTES:
    Module          ->  Module-Document;
  SEQUENCE:
    Module := CREATE ( Module-Document );
    Design ( Source = Module );
    Code ( Specification = Module.Design );
    Validate ( Source = Module );
END_ACTIVITY Implement-Module;
```

Interpretation of an instance of that activity would lead to the following tree:

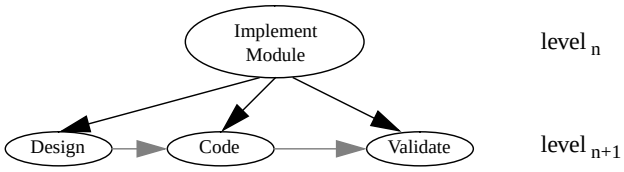

Fig. 2: Instantiation Tree of Activities

The grey arrows between the subactivities show the logical dependencies regarding the order of execution. For instance, the earliest possible start of activity *Code* is after ending activity *Design*.

The action parts of the created activities may be refined, too. We repeat dividing software development processes if the preconditions are true. This results in trees representing the whole project on several levels of abstraction and accuracy.

An activity's interpretation ends with the evaluation of its postcondition when the last statement of the action part has been executed. The intention is to check the consistency of the enacted project step.

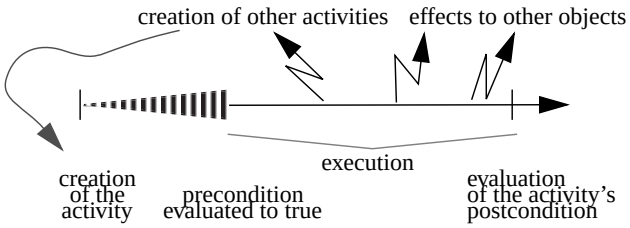

Fig. 3: Lifetime of an Activity

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

First, the moments of an activity's creation ("macro call") and the execution of its action part do not fall together. The invocation of a subactivity doesn't work like a usual function call, but the declaration of an intention to execute this activity. We get knowledge about a project step a long time before it has to be enacted (for illustration see Fig. 3). This gap can be used regarding several aspects, e.g. to assign resources or to make assumptions about the costs of project steps.

Second, every activity gets its own substance and identity, the instance itself. The object is a counterpart for the real world process which gets public in the system for further examination or inclusion in expressions. The object representing this step can be scheduled and integrated in different plans.

Third, the tasks and data are encapsulated. Processes can easily be replaced or they may be changed without affecting other activities. This is also an important aspect for migration or concurrency of processes.

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

The specific representation of software process models in an object oriented way enables a more natural view on software development processes. Complex relationships are easier to understand because they can be defined using interfaces and protocols between the involved classes. The large amount of non-homogeneous data, which is needed to describe even one single project, can be structured using classification and abstraction by class hierarchies. Distinct project steps that are not well known at the start of the project can be refined incrementally by specialization of classes at runtime.

But the most important fact is that we are now able to make forecasts about processes with a very simple mechanism: We have just to collect the objects of type ACTIVITY and analyze them.

Object oriented process programming leads to the above strong benefits. We have obvious a powerful mechanism to understand and manage software projects. But do not count the chickens before they are hatched. There is still one problem which has to be solved.

Just-in-Time Initialization

Forecasts about future project steps can be made only using existing objects. They serve as the basis for planning and scheduling. Because we are interested in powerful, prognostic statements about the supported projects, it is clear that the process elements have to be created as soon as possible. But this goal conflicts with the validity of the actual parameters.

In object oriented programming usual the template is filled with values when an object is created. This is called the instantiation of an object, creation and initialization do fall together [2]. All data must be available at the moment the object's lifetime begins. If we would use this techniques all benefits mentioned above would not exist.

For example, the interpretation of activity *Implement_Module* (cf. Example 2) creates three more (subactivity) objects before the first statement, the creation of the *Module-Document*, is executed: *Design*, *Code* and *Validate*. But the activity *Code* needs as input the output of activity *Design*, the *Design-Document*. This dependency between these activities is also expressed by use of the sequential language construct **`SEQUENCE`**...**`END`**. The *Design-Document* entity referenced by *Module.Design* is created during the design phase. The activity *Design* provides the necessary values which serve as basis for the implementation (activity *Code*) of the module.

Using the usual object oriented mechanism for instantiating we will produce the bindings of attributes after the interpretation of the activity *Design* shown in Fig. 4.

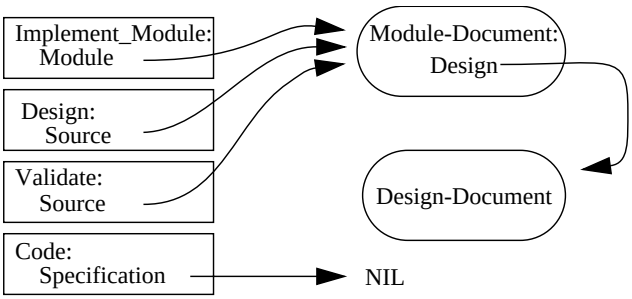

Fig. 4: Bindings after a Usual Instantiation

The attribute *Specification* of the activity *Code* is bound to NIL (if not even a runtime error occures) though we expect it to reference the new *Design-Document*. That happens because the attribute *Specification* gets its parameters (initialization values) at the start of activity *Implement-Module*. At that moment not even the *Module-Document* exists and it may take a long time until the enactment of the activity *Design* will produce the *Design-Document*.

But as we stated before, the instances of activities should be created as soon as possible to enable planning. (Remember that only activities are needed for that purpose. Entities can be instantiated the usual way.) Because of that we have to split the instantiation phase (of activities) into three distinct steps.

In the first step the pure object is created. (To speak in technical terms, only enough memory to store the object is allocated.) Its attributes remain unbound. But the mere existance of the object suffices for scheduling the activity. It cannot be referenced by other activities and therefore there is no need to provide any values.

In the second step the actual parameters of the subactivity call are evaluated and the corresponding attributes are bound to them. This happens before the first possible evaluation of the precondition, that means immediately after the last preceeding action has finished. The bound attributes (and only these!) can be used in the precondition of the activity. Regard

Example 3:

```
ACTIVITY Code;
  ATTRIBUTES:
    Specification   ->  Design-Document;
    start-date      :   TIME INIT ACT-DATE();
  PRECOND:    Specification.Status = finished;
  ...
END_ACTIVITY Code;
```

Because the value of the attribute *Specification* is provided as an actual parameter (cf. Example 2) the precondition of *Code* is well defined. (The ProLan compiler can assure that no unbound values are used in the precondition.)

In the third step the unbound values are initialized with the **INIT** values if provided. This takes place after the precondition of the activity has evaluated to true and the activity becomes enacted. For some attributes it makes sense to choose this moment for initialization; for example look at *start-date* in Example 3. That attribute is bound at the moment when the interpretation of the activity *Code* starts; this is exactly what we expect to happen.

So we split the initialization of an object into three phases and by this way the object is slowly filled with data. But why does this three step initialization work correctly? In object oriented programming usually there is no sequential control flow, so there is no possibility to decide when an object is referenced. Therefore its attributes have to be valid from the beginning of its existance on, that is the moment of its creation.

In the MoMo model an activity's attributes are divided in two classes, depending on their use in precondition or action part. Remember Example 2 again: The precondition of activity *Code* can be evaluated the first time after activity *Design* has finished. The ProLan compiler can assure that no unbound attributes are used in the precondition and the remaining attributes may be used only in the action part, that means after their initialization. Because the attributes are bound in time before their first use but as late as possible, we call this mechanism **Just-in-Time Initialization**.

In our current (prototypical) implementation of the MoMo system (cf. [3], [9]) we use the Common Lisp Object System (CLOS, cf. [8]), as implementation language. The ProLan compiler as well as the Virtual Process Machine are written in CLOS and the user classes defined in ProLan are translated into CLOS code. A software development process will take at least some weeks until its termination and deals with units of time like hours and days (instead of milliseconds and seconds). So we are not handicapped by the use of an interpreter instead of a compiled language. Moreover we take advantage of the possiblity to define ProLan user classes and to translate them into adequate CLOS constructs at runtime without stopping and restarting the whole system. Last but not least the interpretation of actions gives us more chances to recover from runtime errors than executing the compiled code would do and the user can change the incorrect ProLan code at runtime if neccessary.

Conclusion

The object oriented approach yields an adequate modelling of the dynamic and flexible properties of software development processes. Especially the handling of non-heterogenous data, the relations between them and the incremental refinement of entities and activities at runtime are perfectly supported by the object oriented paradigm.

Moreover, object oriented techniques allow for a special representation of the tasks to be performed during a software project. The processes are enclosed and get an identity. Using the MoMo model a process and its environment is represented by objects of the class ACTIVITY. MoMo's Virtual Process Machine interprets the action part of these objects to enact a project. During interpretation more and more objects of class ACTIVITY will be created to represent the more detailed knowledge which becomes available while the project proceeds.

The (subactivity) objects should be created as soon as possible to enable planning, because the mere existence of objects leads to informations about future projects steps. But this arises a conflict with the validity of parameters passed to the new created objects. They should be initialized as late as possible, because the actual parameters must be computed from actual values. Unfortunately, there are two non-orthogonal goals: planning vs. correctness.

The usual mechanism for instantiating objects can not solve this conflict. There exists no time gap between an object's creation and its initialization which could be used for planning aspects. So we choose a slight different approach for supplying the objects with data. The instantiation of an object is splitted into distinct steps, according to the different use of the object. We are able to do so, because we know exactly when the data is needed.

The attributes of activities must be initialized just-in-time before their use. The moment of initialization can be determined from the part of an activity where they are used first (i.e. precondition,

action part or postcondition). Depending on the use of the data we can demand the assignment of values to formal parameters, until they are referenced the first time.

Using this special mechanism we are able to create an object a long time before we can create it using usual semantics, and so planning about future project steps can take place.

References:

[1] Rainer Bleisinger, Peter Knauber, Wolfgang Schramm, Martin Verlage: *Software Process Description and Enactment Considering Temporal Constraints*, IV. International Symposium on AI, Cancun, Mexico, Nov. 1991.

[2] Grady Booch: *Object Oriented Design With Applications*, Benjamin/Cummings Publishing Company Inc., 1991

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

[4] Leon Osterweil: *Software Processes are Software too*, Proceedings of the 9th International Conference on SE, Monterey California, April 1987

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

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

[7] Wolfgang Schramm, Martin Verlage, Peter Knauber, Rainer Bleisinger: *Software Process Enactment Based on an Object Oriented Description*, Proceedings of the 4th International Workshop Software Engineering & its Applications, Toulouse, December 1991

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

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

[10] Lloyd G. Williams: *Software Process Modeling: A Behavioral Approach*, Proceedings of the 10th International Conference on SE, Singapore, April 1988

Authors:

Martin Verlage
Dept. of Computer Science, University of Kaiserslautern
P.O. Box 3049,
D-6750 Kaiserslautern, Germany
e-mail: verlage@informatik.uni-kl.de

Peter Knauber
Dept. of Computer Science, University of Kaiserslautern
P.O. Box 3049,
D-6750 Kaiserslautern, Germany
e-mail: knauber@informatik.uni-kl.de