Electronic Notes in Theoretical Computer Science 2 (1995)
URL: http://www.elsevier.nl/locate/entcs/volume2.html 10 pages

A Programmed Graph Rewriting System for Software Process Management

Peter Heimann, Gregor Joeris, Carl-Arndt Krapp
Bernhard Westfechtel

Lehrstuhl für Informatik III
RWTH Aachen
D-52074 Aachen
[peter|gregor|krapp|bernhard]@i3.informatik.rwth-aachen.de

Abstract

Managing the software development and maintenance process has been identified as a great challenge for several years. Software processes are highly dynamic and can only rarely be planned completely in advance. We present an approach to software process management which is based on hierarchical nets of processes connected by data and control flow relations. Editing and execution of process nets are highly intertwined. Dynamic process nets are formally defined in PROGRES, a specification language which is based on programmed graph rewriting systems. Graph rewriting systems are a natural choice for several reasons. In particular, process nets are complicated graph structures, and editing as well as execution operations may be specified in a uniform way by graph rewrite rules. The graph rewriting system will form the foundation of a sophisticated process management system.

1 Introduction

Managing the software development and maintenance process has been identified as a great challenge for several years [2]. Only rarely can complex software processes be planned completely in advance. Decisions have to be made during process execution which determine how to proceed. Software process management has to meet the following requirements with respect to process dynamics: (1) Forward development: The process structure depends on the product structure which evolves gradually. For example, the modules of a software system are determined in the design phase. Only then may work assignments for implementation be performed. (2) Feedbacks: As development proceeds, errors are detected in later phases which require enhancements of results produced by earlier phases. The consequences of such feedbacks cannot always be predicted. For example, a bug discovered during module testing may require changes to the module implementation, but it may occasionally even affect the design. (3) Concurrent engineering: In order to shorten development cycles, concurrent engineering [9] proposes methods to increase concurrency in the development process. To this end, cooperation between processes must be enhanced such that reasonable intermediate results may be delivered as soon as possible. As a result, each process operates in a highly dynamic work context.

In this paper, we present dynamic process nets for managing evolving software processes. Dynamic process nets are characterized by the following features: (1) Processes are arranged in a composition hierarchy which represents the work break-down structure. (2) Within a subnet, processes are connected by forward flow relations which determine the order of their execution. (3) In addition, feedback flow relations are used to communicate results of analyses or problem reports from successor processes to their predecessors. (4) Furthermore, processes are connected by data flow relations which refine hierarchical or horizontal relations according to (1)-(3). (5) In order to support forward development and feedbacks, editing and execution of process nets are highly intertwined. (6) To support concurrent engineering, an active process has a dynamic workspace whose input/output ports are used to consume/produce intermediate versions of software documents. (7) The evolution of dynamic process nets is controlled by a schema which defines domain-specific types of processes and relations (by instantiating generic types).

Dynamic process nets take over concepts from net plans (2) [7] and data flow diagrams (4) [12] and adapt them to the needs of software process management. Our approach differs from related work on software process management in various ways. Petri nets have been employed in software process management systems such as FUNSOFT [3] and SPADE [1]. The firing behavior of transitions does not account for concurrent engineering (6). Furthermore, in these systems a process is executed by instantiating a net from a template, populating it with tokens, and firing transitions. Unlike dynamic process nets, FUNSOFT and SPADE do not maintain evolving instance-level nets (5,7). This observation also applies to data flow diagrams. On the other hand, EPOS [8] does maintain instance-level nets which are constructed on demand by an AI planner. However, EPOS supports neither feedback relations (3) nor concurrent engineering (6). Furthermore, EPOS does not distinguish between control and data flow.

Dynamic process nets are formally defined in PROGRES, a specification language which is based on programmed graph rewriting systems ([10], see e.g. [4,6] for related approaches). The formal specification is developed with the help of the PROGRES environment which provides tools for editing, analyzing, interpreting, and compiling PROGRES specifications [11]. Operations on process nets are specified by high-level graph rewrite rules which describe complex replacements of subgraph patterns. In this way, both execution operations, which manipulate runtime data, and edit operations, which perform structural changes, are expressed in a uniform framework. The PROGRES specification defines the structure and behavior of dynamic process nets precisely and unambiguously on a high level of abstraction. Since the specification is executable, rapid prototyping may be applied in order to obtain an experimental software process management system.

2 Informal Description

First, we describe evolving process nets from a user's point of view. As an example, we use the development of a simple software subsystem consisting of four modules (fig. 1). The top module D imports from modules B and C, which each in turn uses services from A.

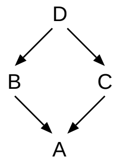


Fig. 1. Subsystem architecture

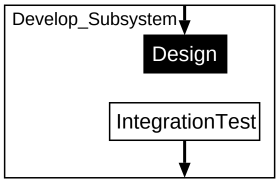


Fig. 2. Initial net (control flow view)

Differing information needs of users are taken care of by views on the process net. While a manager, for example, wants to check on the status of the whole project without getting drowned in details, a technical developer needs the particulars of the tasks that have been assigned to him and their context. The coarse control flow view shows processes and their execution state. A process can be either atomic, or refined by a net of processes connected by control flows. These flows both control the activation of subsequent processes and transport data.

As in our example the process net depends on the subsystem architecture, which is yet unknown when development begins, we start to refine the complex process `Develop_Subsystem` by the initial net of fig. 2. Active processes are shown as black, inactive ones as white boxes. Once the design process has produced a coarse architecture description (fig. 1), the process net can be extended. The process engineer does not have to make this modification fully manually. He has to decide between a bottom up and a top down test strategy. A tool which is aware of the design language can then add the necessary implementation and test processes to the net (fig. 3, bottom-up case).

The design process does not have to deliver all its outputs at once. The detailed module interface definitions can be produced later on. As soon as its needed inputs are complete, a depending module implementation process can start, even if the definitions for other modules are not yet ready. The design process moves into state `done` after it has produced all of its required outputs. Conversely, a process can start before all of its inputs are available. In a test process, for example, the test driver can be written as soon as the module interface definition is available, before the implementation to be tested is ready. As long as a process is active, it can consume inputs and produce outputs. Additionally, preliminary versions can be output first, which are gradually replaced by completed versions. In comparison to a strictly phase-oriented approach, the parallelism thus gained makes better use of personnel resources (a test engineer would otherwise sit idle until after the implementations are completed). Furthermore, the review of preliminary versions by subsequent processes helps to find gross errors as early as possible, when it is still easy to correct them.

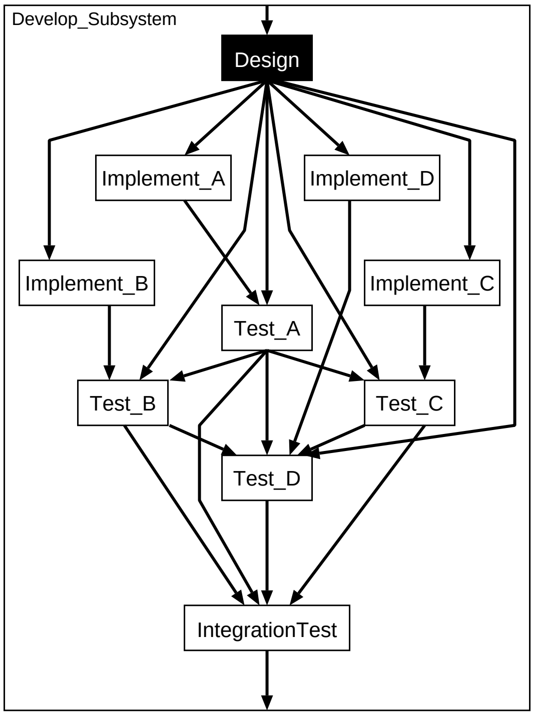


Fig. 3. Process net after extension (control flow view)

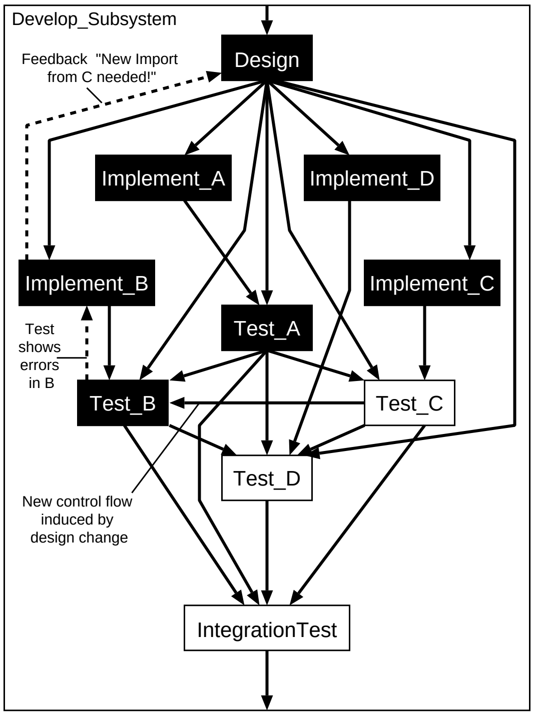


Fig. 4. Concurrently active processes and feedbacks (control flow view)

If a process detects a bug in one of its input documents, a feedback flow is added to the net (dashed arrows in fig. 4) along which the bug report is propagated to the producer of the faulty document. In order not to clutter the net, feedback flows are dynamically inserted only when needed. The responsible process then has to correct the error and produce a new output version. It might itself trigger a further feedback. If in our example the coarse architecture gets modified, the structure of the process net is affected as well. As a simple case, a new import from module B to C results, according to the bottom up test strategy, in a new control flow from process `Test_C` to `Test_B`.

In the data flow view, the control flows and the input and output ports of processes are refined, so that the flow of every information unit or document can be seen separately. Fig. 5 shows a cutout of the process net, where input ports are denoted by black, output ports by white circles. Data flows that refine control flows are shown as thin arrows. While there is only one control flow between `Design` and `Implement_B`, the data flow view shows two distinct information flows: for the interface definition of B and the interface definition of the imported module A.

The evolution of the process net on instance level, as shown in preceding figures, is governed by the specific model in fig. 6. Process types are represented by ellipses. For each type, the model defines the minimum and maximum numbers of instances a correct net must have. The definitions of input and output ports each carry a name, the type of document they can pass, and the minimum and maximum number of ports of this type a process instance may have.

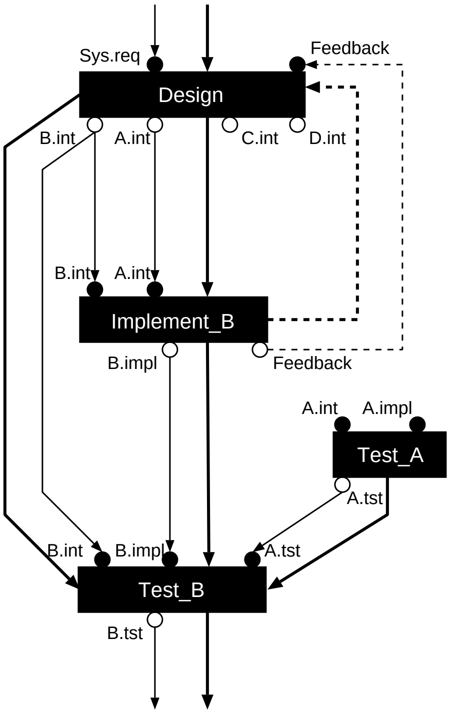


Fig. 5. Cutout of process net, data flow view

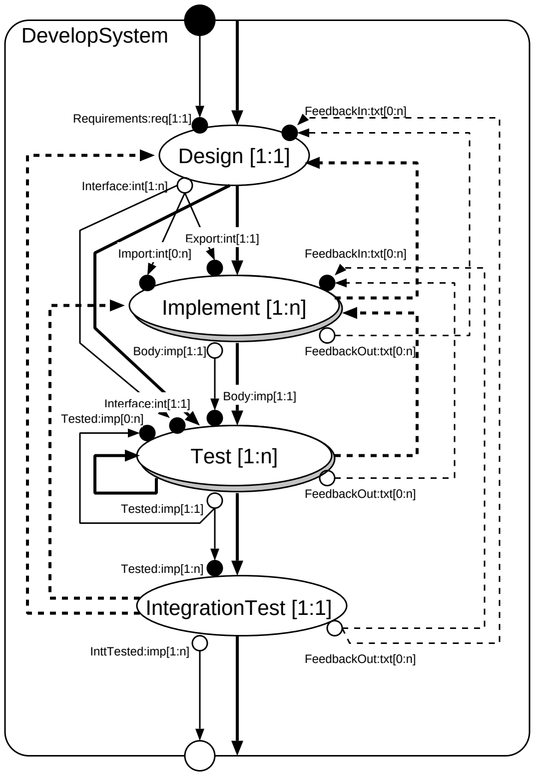


Fig. 6. Specific model for software development nets

3 Formal Specification

In this section, we will be concerned with the developer's view of an environment for dynamic process nets. To define the semantics of such nets, we use the specification language PROGRES which is based on on programmed graph rewriting systems.

We use a generic approach for the design and implementation of an environment for dynamic process nets which is sketched in fig. 7. PROGRES has a stratified type system which distinguishes between node classes, node types and node instances. This permits us to define a meta model for process nets which factors out all common properties for different scenarios, to define a specific model which describes all relevant processes, and finally, to obtain a data structure which describes the process nets to be executed (process graphs).

We define the meta model by the classes and productions of a PROGRES specification (upper left corner of fig. 7). It is independent of an application domain and considers the common properties for process nets in different application areas like software engineering or CIM. The attributes defined in the classes are divided into type-level and instance-level attributes. While the instance-level attributes are attached to the nodes of the process graph, the type-level attributes are attached to the instances of classes (i.e. the node types). Besides the derived attributes (not described in this paper), the latter ones are mainly used as generic parameters in order to adapt the meta model to a specific scenario.

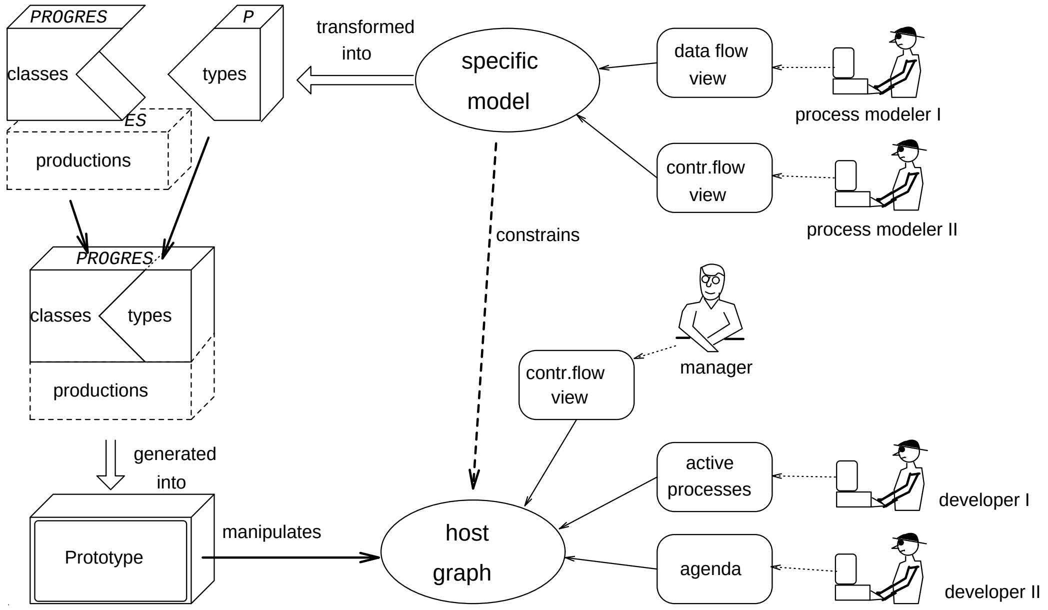


Fig. 7. Generic approach for dynamic process nets

As a first step of the adaption, process modelers specify a model of an application domain in different views (upper right corner of fig. 7). For example, one modeler defines the control flow between processes occurring in the application domain, while another defines the data flows between these processes. This can be done by ER-like diagrams (cf. fig. 6). These views are on top of a data structure which has to be transformed into the type system of PROGRES to obtain a specification of an environment for process nets.

By merging the specific types with the classes and productions, we obtain a complete graph grammar specification for dynamic process nets. The generic parameters of the meta model are bound to the actual parameters of the specific model. With the help of a generator we get a prototype with edit and execution operations for such nets. The prototype manipulates a data structure (the process graph) which describes the internal representation of such an environment. Different views can be installed on top of this representation for the different persons working in a development process.

We are now considering the meta model in more detail (cf. fig. 8). As mentioned above, this part is specified by the class level of the PROGRES graph schema. All classes and types of nodes and edges occurring in a process graph are defined in a PROGRES graph schema. The presented schema is incomplete inasmuch as it does not define the node attributes. Boxes, dashed and solid lines represent node classes, inheritance relations, and edge types, respectively. `ITEM` acts as root of the class hierarchy, not only for the process model, but also for the resource model and the product model which are not discussed in this paper. `PROCESS_ITEM` is a superclass which covers all entities occurring in our process model. A `PROCESS` can be an `ATOMIC` or a `COMPLEX` one. While atomic processes are not refined, a complex process is composed of some processes which in turn can be atomic or complex ones. A `Has` edge between `PROCESS` and `PARAMETER` nodes models the fact that each

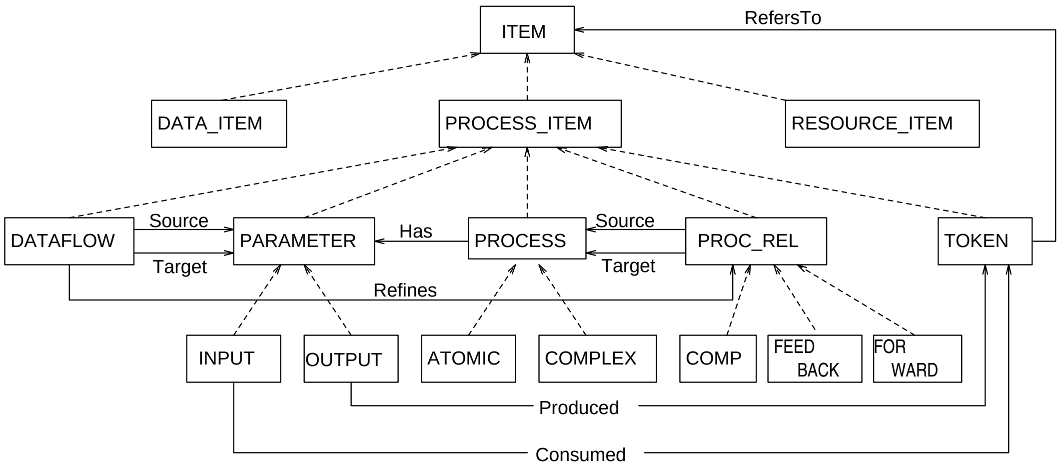


Fig. 8. A PROGRES graph schema for the meta model

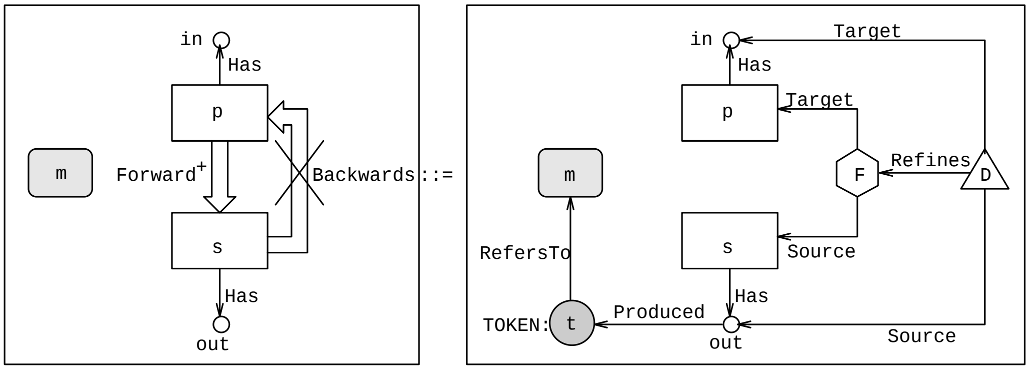
process needs some input data (`INPUT`) to produce some output (`OUTPUT`). Via a `DATAFLOW` node, we connect output parameters to input parameters of succeeding processes. Between processes we have three different relations. The control flow dependencies between succeeding processes are covered by `FORWARD` nodes. Between a complex process and all its child processes we have a composition relation (`COMP`). In order to handle feedbacks, we relate processes by `FEEDBACK` nodes. If in addition to a process-relation a data flow exists between two processes, the data flow refines the corresponding relation.

The data flow is modeled by a token game. The properties of a token are described in the `TOKEN` class. Each token refers to some item (i.e. arbitrary items are permitted by the meta model as inputs of processes, including products, processes and resources). The tokens are flowing via the `DATAFLOW` nodes and are consumed and produced via the input and output ports by the processes. Before these complex operations are described by graph rewrite rules, we give an example how the adaption of the meta model is performed.

```
node type Implement :  ATOMIC
  redef meta
    In := {Export, Import, FeedbackIn} ;
    Out:= {Body, FeedbackOut};
end
node type Import :  INPUT
  redef meta
    FormalType := Interface;
    FormalOptional := true;
    FormalMany := true;
end;
node type Export :  INPUT
  redef meta
    FormalType := Interface;
    FormalOptional := false;
    FormalMany := false;
end;
```

Fig. 9. Cutout of concrete model specified by PROGRES types

```
production Feedback ( s, p : PROCESS; m : MESSAGE;
                      out : OUTPUT;  in: INPUT;
                      D:  type in DATAFLOW; F: type in  FEEDBACK)
```




```
  condition
    ( (s.state = active)  and
    (out.FormalTypes <=> in.FormalTypes) and  (m.type  in out.FormalTypes)  and
    (in.type  in  D.TargetParameter) and (out.type  in D.SourceParameter)  and
    (p.type  in  F.TargetProcess) and (s.type  in  F.SourceProcess))
  transfer
    s.State := suspended;
end;
```

Fig. 10. Graph rewrite rule for inserting a feedback in the process graph

In order to adapt the generic model, specific types of processes, parameters, documents etc. have to be defined. Fig. 9 shows a cutout of a specific model for the implementation process specified by PROGRES types. Note that the process modelers have a more user friendly view on this specific model (cf. fig. 6). The PROGRES types can be easily obtained by a transformation step. To adapt the meta model, the type-level attributes (called meta attributes in PROGRES) are initialized with type-specific values. For example, the type-level attribute `In` of the type `Implement` is initialized with the types `Export`, `Import` and `FeedbackIn`. This means that an implementation process needs its own interface, perhaps some other interfaces and error reports resulting from a feedback. The meta attributes cannot be changed on the process graph level and are used to check the applicability of the graph rewrite rules specified in the productions of the meta model (see below). The meta attributes `FormalType`, `FormalMany` and `FormalOptional` of the node type `Export` are used to check whether the right tokens are consumed and specify the cardinalities of the parameters.

Complex operations on the process graph are specified by graph rewrite rules. The operations are on the level of the meta model and are independent of a specific application area. Fig. 10 presents an example of a rewrite rule for inserting a feedback. To this end, a message `m` (which must have been created before applying the rule) is attached to a token `t`, which in turn is attached to an output port `out` of some successor process `s`. A feedback relation of type `F`, which does not exist yet, is inserted between the two processes. Furthermore, a data flow of type `D` is created, which ends at an input port `in` of some predecessor process `p`. Finally, `s` is suspended to wait for an improved input version. Note that the rule given in fig. 10 describes both a structural modification and a state change; i.e. it combines editing and execution in a single rule.

The rule can only be applied if all conditions are fulfilled. The first condition ensures that the process `s` must be active to evoke a feedback. The other statements in the condition part are used to check the applicability of this operation against the specific model. To this end, the type parameters `D` and `F` are used. For example, a feedback between `s` and `p` can only be inserted in the process graph, if we define in the specific model a feedback type `F` whose `TargetProcess` and `SourceProcess` attributes contain the type of `p` and the type of `s` respectively. A similar check is made for the data flow type `D`. Furthermore, it is checked whether the `out` parameter is compatible with the `in` parameter and the output `m` is of the right type (`out.FormalType`). If all condition statements evaluate to true, the elements of the left-hand side in the process graph are replaced by the elements of the right-hand side. Finally, the value `suspended` is assigned to the `State` attribute of process `s`.

4 Conclusion

We have presented dynamic process nets for managing evolving software processes. A formal specification is under way which currently covers about 50 pages. Due to the lack of space, some important properties of dynamic process nets could not be described here (e.g. distinction between interfaces and bodies of processes, and handling feedbacks through process versions). So far, the specification covers the meta model only. Examples of specific models have been described informally, but still have to be mapped into a formal specification (however, the principles of such a mapping have already been worked out). The specification is large and complex. However, we believe that a practically usable software process management system must support the features which were incorporated into the specification.

When applying PROGRES to software process management, we have identified the following strengths of the specification language: (1) The graph schema allows for describing complex consistency constraints. Furthermore, meta attributes and derived attributes (not described here) can be used to adapt graph rewrite rules without writing ‘code’. (2) By means of graph rewrite rules, complex graph transformations may be specified in a declarative way on a high level of abstraction. (3) Programming adds considerable expressive power to the specification language.

On the other hand, there are also some limitations which are partially addressed in [5]: (1) The current PROGRES environment does not yet support a module concept to structure large specifications. (2) PROGRES provides dynamic binding for attribute evaluation rules only. Object-oriented concepts need to be supported more comprehensively (graphs as objects, redefinition of graph rewrite rules). (3) Genericity has been simulated successfully, but it is not yet supported explicitly. (4) The data model underlying PROGRES does not allow for a natural representation of process hierarchies (flat instead of hierarchical graphs). (5) Schema modifications are not supported. In particular, the approach described in section 3 fails in case of schema modifications. To cope with this problem, a specific model has to be represented on the instance level rather than by a graph schema - with drastic implications for the specification (no instantiation of a generic model with the help of the PROGRES language itself!).

References

[1] S. Bandinelli, A. Fugetta. *Computational Reflection in Software Process Modelling: The SLANG Approach.* 15th International Conference on Software Engineering, 144—154 (1993)

[2] B. Curtis, M. Kellner, J. Over. *Process Modeling.* Communications of the ACM, vol. 35–9, 75—90 (1992)

[3] W. Deiters, V. Gruhn. *Managing Software Processes in MELMAC.* 4th Symposium on Software Development Environments, ACM Software Engineering Notes, vol. 15–6, 193—205 (1990)

[4] A. De Lucia, A. Imperatore, M. Napoli, G. Tortora, M. Tucci. *The Tool Development Language TDL for the Software Development Environment WSDW.* 5th International Conference on Software Engineering and Knowledge Engineering, 213—221 (1993)

[5] G. Engels, A. Schürr. *Encapsulated Hierarchical Graphs, Graph Types, and Meta Types.* SEGRAGRA '95, ENTCS, to appear (1995)

[6] H. Göttler. *Graph Grammars Used in Software Engineering* (in German). IFB 178, Springer-Verlag (1988)

[7] D. Ince, H. Sharp, M. Woodman. *Introduction to Software Project Management and Quality Assurance.* McGraw-Hill (1993)

[8] M. Jaccheri, R. Conradi. *Techniques for Process Model Evolution in EPOS.* IEEE Transactions on Software Engineering, vol. 19–12, 1145—1156 (1993)

[9] R. Reddy, K. Srinivas, V. Jagannathan. *Computer Support for Concurrent Engineering.* IEEE Computer, vol. 26–1, 12—16 (1993)

[10] A. Schürr. *Rapid Programming with Graph Rewrite Rules.* USENIX Symposium on Very High Level Languages, USENIX Association, 83—100 (1994)

[11] A. Schürr, A. Winter, A. Zündorf. *Graph Grammar Engineering with PROGRES.* 5th European Software Engineering Conference, to appear (1995)

[12] E. Yourdon. *Modern Structured Analysis.* Yourdon Press, New York (1989)