Feedback Handling in Dynamic Task Nets

Carl-Arndt Krapp and Bernhard Westfechtel
Lehrstuhl für Informatik III, RWTH Aachen, D-52056 Aachen
[krapp|bernhard]@i3.informatik.rwth-aachen.de

Abstract

While a software process is being executed, many errors and problems occur which require to reconsider previously executed process steps. In order to handle feedback in a process-centered software engineering environment, several requirements need to be addressed (adaptability, human intervention, impact analysis, change propagation, restoration of the work context, and traceability). Feedback management in DYNAMITE meets these requirements. DYNAMITE is based on dynamic task nets and specifically supports feedback through feedback relations, task versions, and customized semantics of data flows. A methodology for feedback handling is also represented. A programmed graph rewriting system serves to specify analysis, planning, and enactment — which may be interleaved seamlessly — in a uniform formal framework.
Keywords: Software process management, feedback handling, graph rewriting

1 Introduction

It has been recognized by many researchers that software processes are highly dynamic [12]. However, current models and tools address process dynamics only to a limited extent. In particular, this applies to management of *feedback*. While a software process is being executed, many errors and problems occur which require to reconsider previously executed process steps. For example, a module test may reveal errors in the implementation, or a designer may detect inconsistencies in the requirements definition.

While feedback is mentioned in virtually every textbook on software engineering, it is by no means obvious or trivial how to support them in a process-centered software engineering environment. To this end, an underlying formal model is needed which meets the following requirements:

R1: Adaptability. In routine processes, feedback may be planned, e.g., by introducing loops into the process model. On the other hand, in less well-structured processes feedback may occur unexpectedly and randomly, calling for flexible and reactive (instead of proactive) support.

R2: Human intervention. In many cases, handling of feedback requires human judgment and can at best be automated partially. For example, in case of severe errors large parts of a process may have to be suspended, while minor errors only affect a small set of process steps. Usually, the "machine" (process engine interpreting a process model) cannot tell these cases apart.

R3: Analyses. Sophisticated analyses have to be offered to assess the impacts of feedback, the current status of processing feedback, etc. These analyses concern the past (terminated steps to be iterated), the present (active steps to be suspended), and the future (planned steps to be postponed).

R4: Change propagation. Flexible commands must be provided to propagate changes induced by feedback timely (to avoid wasted work on obsolete inputs), precisely (to exactly determine actually affected steps), and incrementally (to process changes rather than to restart from scratch).

R5: Restoration of the work context. In many cases, iteration of a process step is performed by refining its previous results. To support incremental improvements, the old work context (inputs, outputs, available tools, etc.) has to be restored.

R6: Traceability. Handling of feedback has to be recorded and packaged in an experience base [3]. By analyzing historical data concerning feedback, problem spots in the software process may be identified (e.g., feedback with long distances between source and target or a large number of iterated process steps).

The *DYNAMITE* approach (*DYNAMI*c *T*ask N*E*ts) [15, 17] meets these requirements. In DYNAMITE, software processes are represented by nets of tasks connected by control flow and data flow relations. Each task has an interface and a realization, the latter of which is either complex or atomic (task hierarchies). Tasks and relations are dynamically instantiated from corresponding types, resulting in evolving instance-level task nets. Planning, analysis, and enactment are seamlessly integrated (fine-grained interleaving).

Management of feedback is supported in the following way:

F1: Feedback is represented explicitly by *feedback relations* (R6). A feedback relation goes from a source task s to a target task t which is a transitive predecessor of s with respect to control flow relations. Often, a feedback relation comes with a parallel data flow, e.g., for transmitting a bug report.

F2: If the target of a feedback relation has already terminated, a new *task version* is created to handle feedback. In this way, traceability is supported (R6). Furthermore, each task version is an independent unit of work to be scheduled by project management.

F3: A task version inherits its work context from its predecessor version (R5).

F4: The semantics of data flow is defined such that product versioning is taken into account (R4). To correct errors and handle problems, a task may produce multiple versions of an output object which are transmitted sequentially along outgoing data flows.

F5: Dynamic task nets comprise terminated, active, and planned tasks (past, present, and future, respectively) and thus provide the foundation for traceability (R6), impact analysis, and status accounting (R3).

F6: DYNAMITE assists with feedback management, but does not automate it completely. Human intervention (R2) is required in many situations (e.g., to determine the target of a feedback relation, to assess its impact, etc.).

F7: Feedback management can be tailored to the needs of a specific application domain (R1). Structural constraints may be defined in a schema by introducing appropriate types of feedback relations. Different propagation patterns (R4) may be encoded by customized graph productions and transactions.

Previous papers have introduced our overall approach to managing software processes, focusing on the overall formal foundation in [14], on software engineering applications in [15], and the DYNAMITE environment in [17]. In contrast, the current paper specifically focuses on feedback handling and presents novel contributions not published so far, neither in own publications nor by others. In particular, we come up with a list of requirements, discuss the features built into DYNAMITE in order to meet these requirements, introduce a methodology for handling feedback, and also provide a formal specification in terms of a programmed graph rewriting system. We argue that feedback requires specific support through a process-centered environment, and in particular consider task versions, feedback relations, and versioned data flows eesential features not supported by other approaches.

The rest of this paper is structured as follows: Related work is compared in section 2. Sections 3 and 4 describe feedback management at an informal and formal level, respectively. A conclusion is drawn in section 5.

2 Related Work

DYNAMITE is almost unique in that it applies *graph rewriting* to software process management[1]. The key advantage is that editing, analysis, and enactment can be specified in a uniform formalism at a high level of abstraction. This general argument — which does not hold e.g. for approaches based on Petri nets (see below), where there is no formalism for describing edit operations affecting the current state of enactment — particularly applies to feedback handling, which requires sophisticated reasoning about and manipulation of task nets.

DYNAMITE is based on *instance-level task nets*, where tasks and relations are dynamically instantiated from types. In general, the structure of a task net is known only at runtime. Similar approaches have been developed in distributed systems (e.g., SDL [30] or Darwin [24]). However, these approaches do not address the specific problems occurring in software processes (e.g., creative processes, human intervention).

Only recently, some approaches which are based on instance-level task nets have been developed as extensions of software configuration management systems (ClearGuide [22] on top of ClearCase and APEL [10] on top of Adele). However, both ClearGuide and APEL do not address feedback handling specifically; e.g., there are neither feedback relations (F1) nor task versions (F2). As a consequence, requirements R4–R6 (change propagation, restoration of the work context, traceability) are not well supported.

Process-centered software engineering environments such as Process Weaver [11], SPADE [2], or MELMAC [13] are based on *populated copies*, i.e., a net template is copied, populated with tokens, and enacted. Instance-level nets (which would consist of fired transition instances and places carrying consumed/produced tokens) are not constructed. Teamware [36] and its successor Endeavors [4] essentially follow the populated copy paradigm as well and

[1] GRIDS[37], the only other approach we are aware of, mainly focuses on conceptual issues (3-dimensional approach to software process modeling) without addressing the semantics of software process management (execution support).

merely offer a special-purpose construct for replacing a placeholder at runtime with a set of parallel tasks with identical context (DYNAMITE is far more general; note the analogy to ER databases).

Populated copies fail to support impact analysis (R3), change propagation (R4), and traceability (R6). This even applies when nets may be modified while being enacted [2, 13]: only the current state of enactment is available (no traces, no planning). Feedback may be handled by cycles, but restoration of the work context (R5) is not specifically supported.

In *rule-based systems* such as e.g. Marvel [18], ALF [5], or MERLIN [28], the software process is modeled by facts and rules which are interpreted by a process engine. Rule applications are driven by forward or backward chaining. Plans are constructed by the process engine. If a step of the plan fails, a postcondition is raised which triggers rules for error handling.

The main drawback of rule-based systems consists in missing support for human intervention (R2). Plans are maintained automatically. They are normally hidden from the users and are displayed only on demand for explanation purposes (no interactive modifications). As we will demonstrate in section 3, human decisions may be required at many points in the process of feedback handling. In particular, this applies to impact analysis (R3) and change propagation (R4).

EPOS [23, 19] combines the net-based and the rule-based paradigm and maintains instance-level task nets similarly to DYNAMITE[2]. Task nets are displayed at the user interface and can also be manipulated interactively. Furthermore, an incremental planner constructs task nets from the product structure. In contrast to DYNAMITE, EPOS does not address feedback handling specifically. As in the rule-based systems cited above, feedback is triggered by asserting error conditions. Feedback is not represented explicitly in task nets (no feedback relations). Task versions are not maintained, resulting in lacking traceability (R6). In case of iterations, the old work context cannot be restored (R5). As a consequence, incremental improvements of previous results are not adequately supported (R4).

HFSP [20] applies attribute grammars to software process modeling. A software process is represented by a dynamically growing execution tree which is expanded by applying productions. Attribute equations define the data flows between task nodes and establish constraints on the order of task execution. A special mechanism called *Redo* [34] supports handling of feedback. `Redo A` replaces a subtree t of the enactment tree (rooted at `A`) with a tree t' of tasks to be executed anew. In contrast to feedback management in DYNAMITE, Redo is a destructive operation which erases a part of the execution history. Incremental changes (R4, R5) are poorly supported, and the process is no longer traceable (R6). Affected tasks are determined automatically without human intervention (R2).

3 Feedback Handling: Informal Description

Dynamic task nets are used to model real world processes when developing or maintaining complex software systems. In this section we give a brief informal introduction to the DYNAMITE approach and explain how feedback is handled. We use the process manager's point of view since he is the responsible person for net modifications, analysis, and enactment. While the first subsection is devoted to present the basic concepts, the second subsection considers

[2] CoMo-Kit[8], which was developed much later, basically follows the same approach.

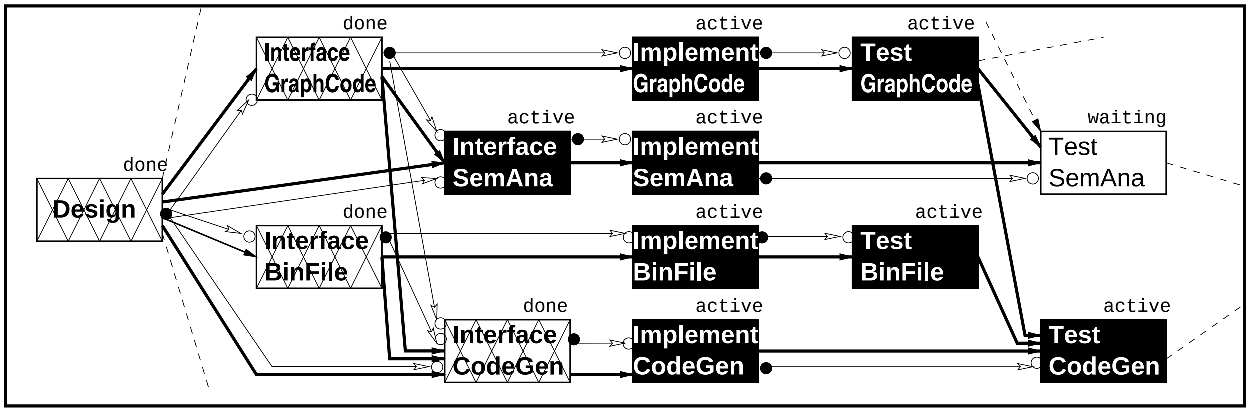


Figure 1: Dynamic task net for construction of a multi-pass compiler (cutout)

the methodology of feedback handling. In the last subsection we explain how appropriate feedback handling can be customized to the user's need.

3.1 Mechanisms

Figure 1 shows a graphical representation of a dynamic task net. Development activities are modeled by *tasks* (boxes) which are related by *control* and *data flow relations* (thick and thin arrows, respectively). A dynamic task net resembles a PERT chart and, additionally, offers execution semantics. A task can be subdivided into several subtasks (not shown in the figure). Therefore several levels of abstractions can be provided. It is up to the manager how many details are modeled and thus can be effectively used for controlling and guiding. The hierarchical structure is not only for structuring purposes. Enactment of the task net takes the hierarchy into account.

The example in figure 1 shows a cutout of a task net for constructing a multi-pass compiler. The cutout covers only a subset of its data structures (`GraphCode` for the intermediate code and `BinFile` for the compiled code) and a subset of its passes (`SemAna` for semantic analysis and `CodeGen` for code generation). After an initial coarse design (`Design`), module interfaces are defined (`Interface`), module bodies are implemented (`Implement`) and finally tested (`Test`). Tasks for defining interfaces and testing modules are performed in bottom-up order; implementation tasks may proceed in parallel. Even tasks connected by control flow relations (e.g. `ImplementGraphCode` and `TestGraphCode`) may be executed in parallel (simultaneous engineering). Finally, note that only the most important data flows are shown to keep the figure clear.

In order to guide and control development processes, execution semantics is defined by *cooperating state machines*. The enactment state of a task describes the behavior of activities abstractly and can be one of `waiting`, `active`, `suspended`, `planning`, `failed`, or `done` (visualized by different fill styles). A state transition is either done manually, or it is triggered by a user-defined enactment rule (see below). All tasks regardless of whether they are atomic or complex have a fixed state transition diagram whose transition conditions can be defined individually. The execution state determines which actions can be performed on a task (consumption of input documents, releasing of output documents, refinement of underlying task nets, etc.). Enactment takes into account that the task net can be refined, extended and modified even if the execution flow has reached an advanced stadium.

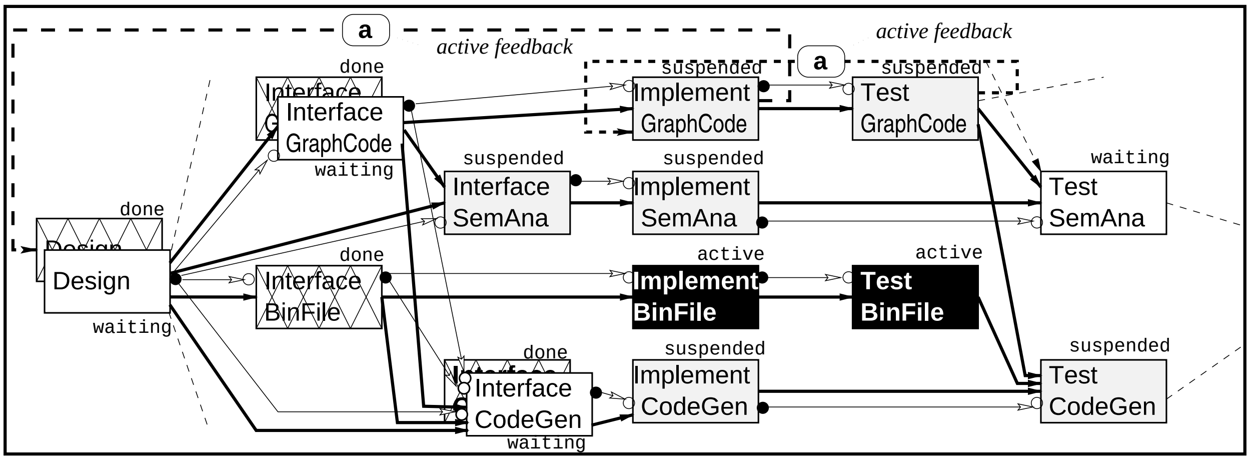


Figure 2: Multiple feedback relations

In order to model feedback situations, where the flow of control is redirected to earlier stages of the overall process, a special *feedback relation* is used. In our example, the test task for the `GraphCode` module observes that performance requirements are not met, resulting in feedback to the corresponding implementation task (cf. fig. 2). Since the implement task could not solve the problem, another feedback relation is introduced back to the design task. The consequences of feedback introduction like suspension or restart of dependent tasks are under manager's control. Thus, the feedback relation is a syntactical construct, which however can be used for analysis purposes and can be enriched with semantical constructs describing the behavior of affected tasks in the net (see below).

When introducing feedback, traceability of the overall process can be lost very easily. In the example above, it is not clear how often the enactment cycle was restarted, which inputs were consumed with each restart, which following tasks were affected, etc. Due to the traceability requirement we have introduced the concept of *task versions*. If an already terminated task has to be restarted again, a new task version is established which can be provided with the same work context (same inputs, same outputs, same actor, etc.) as the original task (shown behind the new version) or which may be embedded into the task net differently. Thus, traceability can be provided by inspecting the different task versions.

3.2 Methodology

By means of the presented concepts, a manager can handle arbitrary feedback according to his needs without loosing traceability. Analyses on dynamic task nets are offered which support him in finding potentially affected tasks. We now consider by means of the example above how feedback is handled methodologically. Seven steps can be identified when the flow of control has to be redirected to earlier stages in the process[3]:

1. **Target determination.** After occurrence of an error or alike, the first step is to determine the target task of feedback. This can be either done by the actor of the source task himself or by the process manager who usually has a better overview of the overall process. In our example, the actors for testing and implementing the `GraphCode`

[3]The order of execution is not determined uniquely.

module introduce feedback, since they know approximately where the origin of the problem is.

2. **Source behavior.** After feedback introduction the behavior of the source task has to be defined. Several alternatives are possible: The source task may stay active, may be suspended, or may even be aborted. In our example the manager decided to suspend the test and implementation tasks for the `GraphCode` module after feedback introduction.

3. **Consequences to source successors.** Successors of the source task may be affected, as well. Analysis on the net supports the manager in deciding upon appropriate reactions. All tasks reachable by control flow relations can be highlighted. The manager may now decide to suspend a subset of these tasks in order to avoid wasted work. Tasks which are likely not affected may continue their work. In our example, the manager follows a "conservative" policy and suspends dependent test tasks (`TestCodeGen`).

4. **Target behavior.** As already mentioned, if the target task has already terminated, a new task version is introduced. In case it is still active, which can be possible if parallel execution of tasks is allowed, appropriate reactions have to be taken. Either the responsible actor is only informed that feedback is introduced or the actor is forced to process feedback immediately. In our example, the actor of the implementation task (`ImplementGraphCode`) is only informed that feedback from the module test is introduced. Because the design task was already terminated, a new task version is introduced and the actor can start with feedback processing.

5. **Consequences to target successors.** Target successors are potentially affected by feedback. Again, analysis on the task net helps to identify these tasks. Similar to source successors, the manager can perform appropriate reactions individually for each task. In our case, almost all transitively dependent tasks of `Design` are either suspended or iterated (new task versions) — except those tasks which are responsible for the `BinFile` module, since this module is supposed not to be affected.

6. **Feedback propagation.** After all consequences have been estimated and appropriate reactions have been performed by the manager, feedback can be propagated trying to accelerate the wave of changes through the net. To reach an orderly development status as soon as possible, the manager may want to enforce some rules concerning e.g. behavior of affected tasks and interaction with other active feedback relations. Note that a feedback relation is called *active* until it is terminated (see below). In fig. 2, this was visualized by a flag attached to the feedback relation.

   In our example, the manager wants to enforce the following policies:

   (a) Tasks along the control flow path between target and source task are not allowed to terminate as long as the feedback relation is still active.

   (b) While a feedback relation is active, no interfering feedback relations may be introduced. In particular, it is not allowed to create another feedback relation which "crosses" an active feedback.

7. **Feedback termination.** Active feedback relations influence the behavior of some tasks. Thus it is essential to determine the point when feedback can be considered as

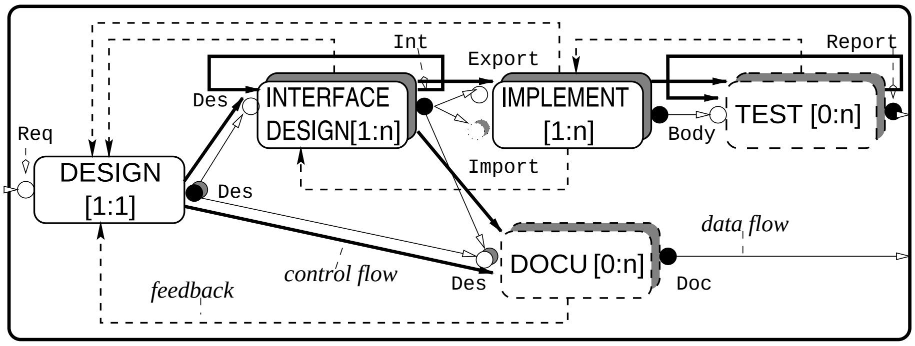


Figure 3: Task net schema

```
task_type Implement
    ...
  on feedback_rcv do
    UnreleaseOutput (...);
    ConsumeInput (...);
    NotifyActor (...);
  end;
  ...
end;
```

Figure 4: Task type and trigger definition

closed. Again, different policies can be used depending on the kind and severity of the feedback:

(a) Feedback can terminate if the flow of control reaches the feedback's source task again.

(b) Feedback can terminate if the feedback's target task terminated successfully.

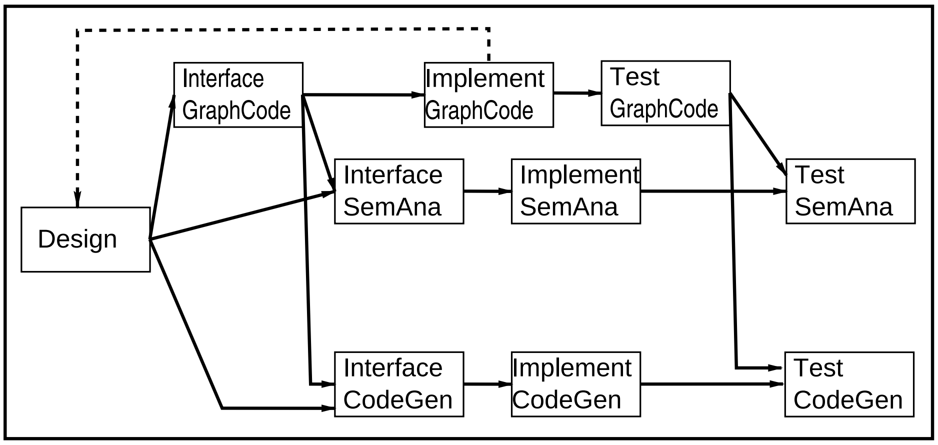
(c) The manager determines explicitly when feedback can be terminated.

3.3 Customization

The basic concepts for feedback handling are very general and offer only little semantical support. A manager is responsible for appropriate reactions according to the introduced methodology. In situations, however, where more information about appropriate feedback handling is available, the DYNAMITE approach offers more sophisticated mechanisms. Customization can be done in a user-friendly notation which may be compiled into the PROGRES specification (see sect. 4).

We call a net as presented in figure 1 on the instance level. If there is no knowledge about the software process available a priori, a manager can build up task nets of arbitrary structure. In particular, feedback can be introduced by a manger wherever he likes. In cases where possible net structures are known a priori, a *net schema* is used to express this knowledge. A net schema is on type level and consists of task types, decorated with

```
on CreateFeedback =
```




```
do
   case
      Active, Planning               : Suspend;
      Done, Abort                    : Restart;
      InDefinition,Waiting,Suspended :  -
   end;
end;
```

Figure 5: Feedback propagation pattern

cardinalities and connected by relation types. A manager may now decide that compiler construction should be done according to the schema shown in figure 3. Feedback types constrain the feedback relations which may be created at runtime. For instance, within a task net which is an instance of this schema, a feedback relation starting from a test task can only go to an implementation task. If there is little knowledge about potential feedbacks available, a general feedback type may be introduced which connects tasks of arbitrary types.

It may be tedious and cumbersome to manually apply the methodology introduced in subsection 3.2. Our approach allows to define task type specific reactions by means of trigger definitions. A *trigger* determines the behavior of a task in case a certain event arises in its context. In figure 4 reactions are defined for implementation tasks when receiving a feedback. If an implementation task receives a feedback, the release of its output is revoked, an incoming error report is consumed, and the actor of the implementation task is notified that feedback has to be processed.

The application of basic operations can be restricted by user-defined *conditions*. Different policies for feedback processing, for instance, can be defined by such a condition. It is associated to a task schema and is evaluated whenever feedback is introduced into a task net which is an instance of that schema. The condition decides whether feedback is introduced or not and it is defined in a first-order logic language. The syntax is similar to PROGRES and is not shown for this reason. An example of a condition will be given later (cf. fig. 10).

For well understood development processes (as for the process of compiler construction), *propagation patterns* offer appropriate mechanism to specify and catch a certain situation in order to react accordingly. While type-level triggers are suitable to specify type specific behavior for a task, they are weak in expressing appropriate reactions on specific net structures which are known a prioi. To this end, propagation patterns can be specified. A propagation pattern specifies a certain situation of the overall process and defines which subsequent actions have to be done. Figure 5 shows a pattern which specifies the situation for feedback handling presented in figure 2. When introducing a feedback relation from the implementation task of the `GraphCode` module back to the `Design` task, all specified tasks within the pattern are collected. Depending on its current enactment state, a task is suspended, restarted, or not affected at all. Note that no task is specified for the `BinFile` module. This reaction cannot be expressed by type-level triggers because they cannot handle task instances individually.

4 Feedback Handling: Formal Specification

In this section we present a small cutout of the formal specification which formalizes the concepts presented in section 3. We use the formal specification language *PROGRES* [33, 31] which is based on programmed graph rewriting. The language provides a uniform formalism to express edit, analysis, and enactment steps on dynamic task nets. Since PROGRES specifications are executable, interactive tools for process management can be generated from a specification. A generated tool can be linked into an existing development environment.

A PROGRES specification defines a graph schema and schema-preserving operations. PROGRES is based on *attributed graphs*. Nodes are decorated with attributes; edges represent binary, directed relations which do not carry attributes. The graph schema defines node classes, which are organized into a multiple inheritance hierarchy, and edge types (no inheritance). Attributes are classified into intrinsic and derived ones. Edges and paths represent intrinsic and derived relations, respectively. Graph productions describe graph transformations at a high level of abstraction. They can be composed into transactions by means of control structures which take nondeterminism into account (backtracking).

Note that PROGRES has not been designed specifically for process modeling. Chapter 3 in [25] provides an overview of its applications to software engineering in general. In particular, we have used the language also for modeling products (software configuration management [35]) and resources [16].

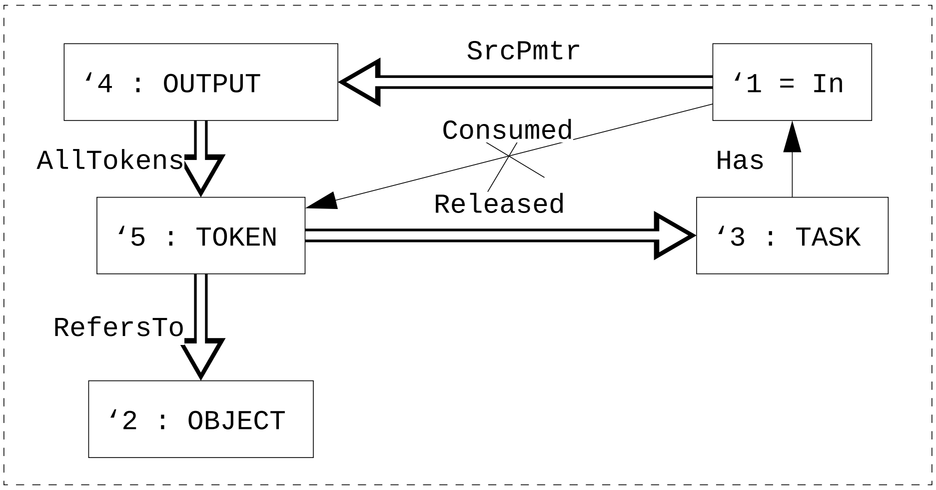
In the first subsection, we present some specification details for the basic constructs (i.e., the generic kernel of DYNAMITE). The second subsection describes how customization can be realized in the DYNAMITE approach using PROGRES.

4.1 Basic Concepts

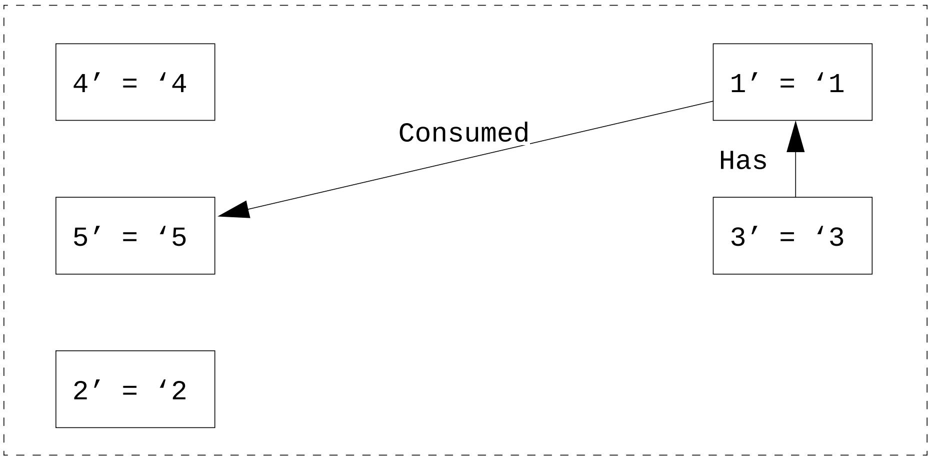
The generic kernel of DYNAMITE defines the elements of dynamic task nets as well as basic operations. In the following, we will focus on mechanisms for feedback handling (F1–F4, see sect. 1). For a general introduction to DYNAMITE, the reader is referred to [15].

Data flows (F4). The semantics of data flows is defined such that product versioning is taken into account. While a task is active, it may receive multiple feedback relations which require changes to its outputs. Thus, a task may produce multiple versions of output objects. These are represented by tokens which are arranged in a list (traceability, R6). A dependent task may consume the most recent, released token (note that tokens may be released to individual tasks). Since the token history is stored, it is possible to compare old and new inputs (incremental change propagation, R4).

```
production Consume( In : INPUT ; out Obj : OBJECT) =
```




```
  ::=
```




```
  condition ‘2.type in ‘1.FormalTypes;
            (‘3.ConsumeCond) and (‘1.ParaConsumeCond);
  return Obj := 2’;
end;
```

Figure 6: Production for consuming a token

Figure 6 shows a *graph production* which describes the operation of consuming a token by a task via an input parameter. When applying a graph production, the following steps have to be performed:

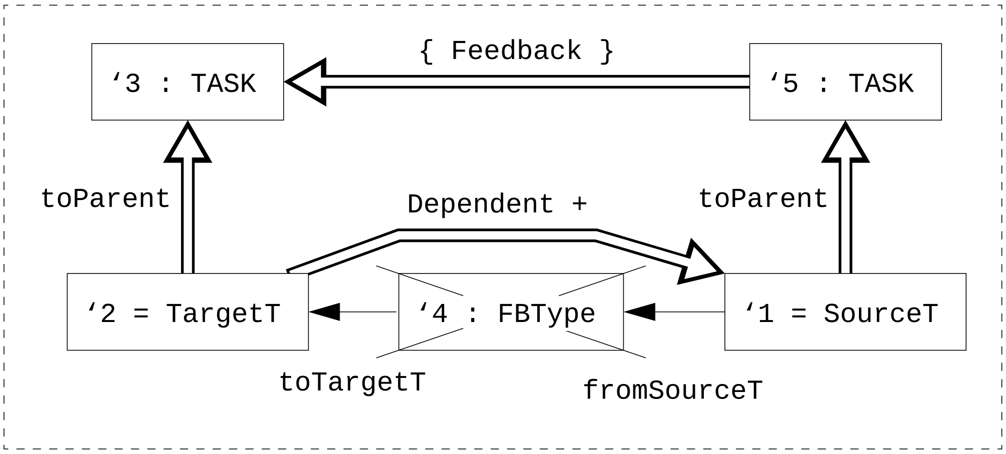
1. Pattern match of the left-hand side of the production (upper dashed box) within the current graph.

2. Evaluation of attribute conditions stated below the left-hand and right-hand side.

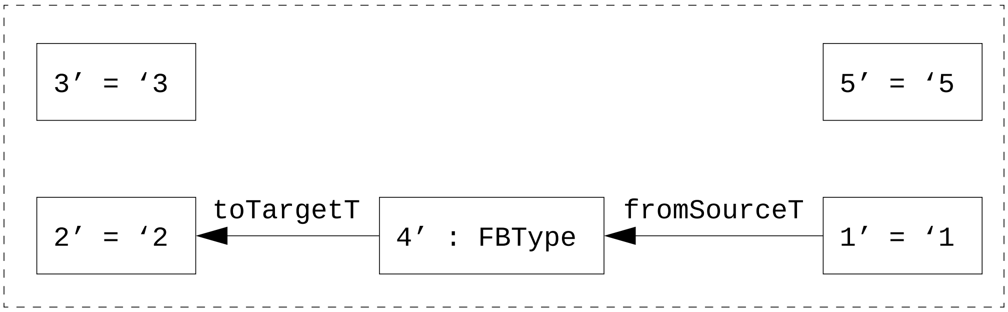
3. If the pattern could be found in the current graph and if the condition holds, replace the left-hand side with the right-hand side.

In the example, the production has an input parameter `In` of node class `INPUT` and an output parameter `Obj` of class `OBJECT`. The left-hand side states that starting from the node

```
production CreateFeedback( SourceT : TASK ; TargetT : TASK ;
                           FBType : type in FEEDBACK ;
                           out FB : FEEDBACK)
=
```




```
  ::=
```




```
  folding { ‘3, ‘5 };
  condition (‘5.FeedbackCond) and (‘3.FeedbackCond);
            (‘1.FeedbackOutCond) and (‘2.FeedbackInCond);
            (‘1.type in FBType.SourceTypes)
            and (2.type in FBType.TargetTypes);
  return FB := 4’;
end;
```

Figure 7: Production for creating a feedback

`‘1` (fixed by the production parameter `In`), a path must be found (indicated by a double arrow) which ends at node `‘4` of class `OUTPUT`[4]. Furthermore, from node `‘4` a `TOKEN` node has to be found via the `AllTokens` path. This node must be connected by a `Released` path with a task node, which in turn possesses the input node `In` (indicated by the `-Has->` edge). The crossed edge ensures that the token was not already consumed by the task's input parameter.

The first condition in the condition part requires that the type of the document to be consumed (node `‘2`) be compatible with the formal parameter type of the input parameter `In`. In the second condition, user-defined restrictions on executing the `Consume` operation are evaluated. These restrictions are formalized as (redefinable) derived attributes. For example, `Consume` may be allowed only if the consuming task resides in the state `active`[5].

[4]Due to the lack of space, we omit the definitions of path expressions.

[5]A concrete example in PROGRES will be given later on (cf. fig. 10).

```
transaction DeriveTask( OldTask : TASK ; out NewTask : TASK)
=
  use task : TASK;
      NewPmtr : PARAMETER
  do
(1)       DeriveTaskItem ( OldTask, out task )
(2)     & MarkTaskAsOld ( OldTask )
        & loop
(3)         DeriveInControlFlow ( task )
          end
        & loop
(4)         DeriveOutControlFlow ( task )
          end
        & for all pmtr : PARAMETER := OldTask.-Has->
          do
              CreateParameterItem
              ( task, pmtr.type, pmtr.Name, out NewPmtr )
            & choose
(5)             DeriveSourceDataflow ( pmtr, NewPmtr )
              else
                DeriveTargetDataflows ( pmtr, NewPmtr )
              end
          end
        & NewTask := task
  end
end;
```

Figure 8: Transaction for establishing a new task version

When applying the `Consume` production, the graph will be expanded. All nodes of the left-hand side are replaced identically (indicated by `X' = 'X` statements). The only edge `-Has->` is preserved and the edge `-Consumed->` is introduced. Note that already existing `-Consumed->` edges to previous tokens are not removed (traceability); the current input of the dependent task is the most recent token referred to by an outgoing `-Consumed->` edge.

Feedback relations (F1). Figure 7 shows a graph production for creating a feedback relation. A feedback relation can only be introduced between tasks which are in a transitive control flow relation (`Dependent +`). Again, type compatibilities and user-defined restrictions are checked in the condition part (see subsection 4.2 for a detailed discussion).

This rule is a good example for presenting the balancing rules when constructing hierarchical task nets. Two different cases must be distinguished:

1. **Feedback within a subnet:** The introduction of a feedback relation between two tasks within a subnet is not further restricted. In this case, the path `toParent` yields for the source and target node the same father node in the current graph. The `folding` expression allows that different nodes of the left-hand side (here nodes `'3` and `'5`) can be mapped onto the same node of the current graph. The path expression `Feeback` collapses in this case (indicated by curly brackets).

2. **Feedback between subnets:** If the feedback relation crosses the subnet border, the father nodes of source and target in the left-hand side are mapped onto two different nodes of the current graph. In this case, a feedback relation can only be introduced if a feedback relation was already established between the father nodes.

```
transaction Feedback( Sender, Receiver : TASK ; FBType : type in FEEDBACK)
=
  use Feedback : FEEDBACK
  do
      CreateFeedback ( Sender, Receiver, FBType, out Feedback )
    & SendCreateFeedback ( Sender, Receiver, Feedback )
  end
end;

transaction SendCreateFeedback( SenderT, ReceiverT : TASK ; Feedback : FEEDBACK)
=
    choose
      when (ReiceiverT.type = Implement)
      then
          UnReleaseOutput ( ReceiverT )
        & Consume ( Report ( Feedback ) )
        & NotifyActor ( ReceiverT )
    else
      when (ReceiverT.type = Design)
      then
          ...
    end
  & choose
      when (SenderT.type = Implement)
      then
          ...
    else
      when (SenderT.type = Design)
      then
          ...
    end
end;
```

Figure 9: Trigger mechanism

Task versions (F2) and restoration of the work context (F3). Creation of a new task version is presented as the last example of basic operations (cf. fig. 8). To preserve the old work context, complex graph manipulations have to be carried out which cannot be expressed by a single graph production. In PROGRES, graph productions may be composed into a *graph transaction*. Application of productions is determined by control structures such as sequence (`&`), exhaustive loop (`loop`), iteration over all elements of a set (`for_all`), and alternatives (`choose`).

The complex graph transaction for establishing a new task version performs the following steps[6]:

1. Create a new task node and establish a new `successor_of` relation between old and new task by applying the `DeriveTaskItem` operation.

2. Mark the predecessor task as old, i.e., set the corresponding node attribute.

3. As long as there is a control flow left, restore the incoming control flow relation.

4. Proceed in the same way for outgoing control flow relations.

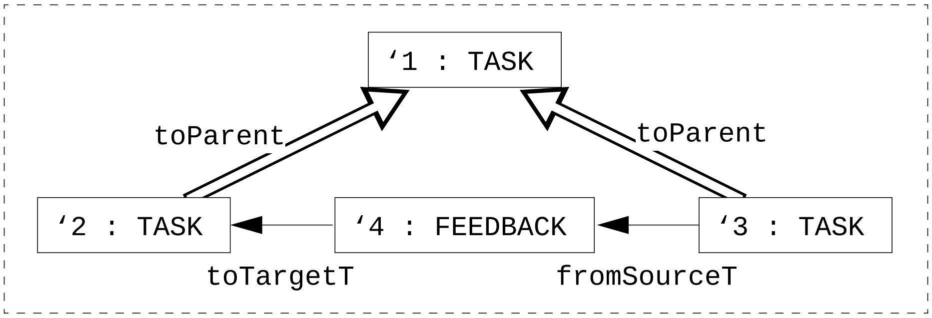
5. For all parameters of the old task, create a parameter of the new task, associate it with the new task version, and restore the old incoming and outgoing data flow relations.

[6]Due to the lack of space, we do not discuss the graph productions invoked.

```
node type Develop : TASK
    ...
  redef derived
    FeedbackCond = HasNoActiveFeedback ( self );
    ...
end;

function HasNoActiveFeedback : ( Task : TASK) -> boolean =
  not (Task is HasActiveFeedback)
end;

restriction HasActiveFeedback : TASK =
  ‘1 in
```




```
  condition ‘4.active;
end;
```

Figure 10: Customized evaluation rule

The transaction shows how parameter passing is performed between graph operations and how operational programming constructs can be combined with declarative graph productions.

4.2 Customization Constructs

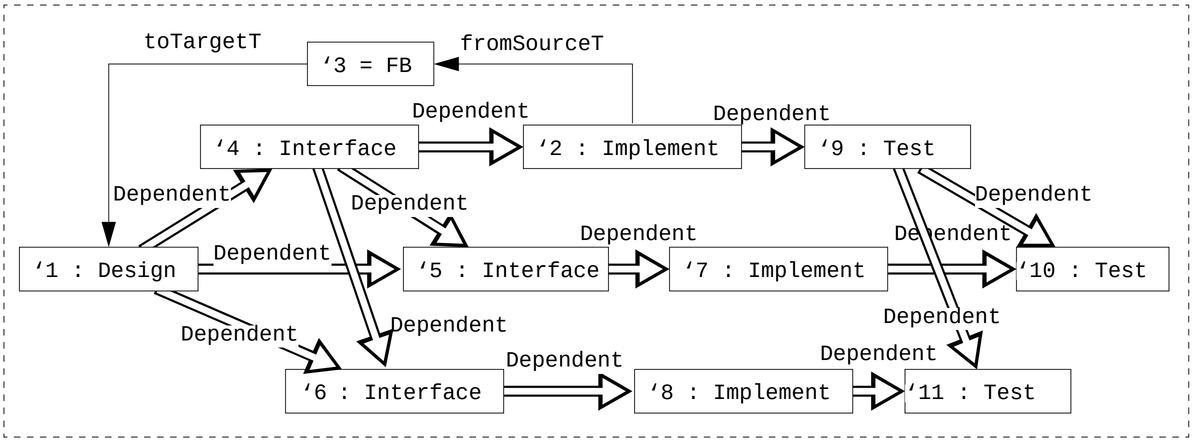
The basic concepts are powerful enough to support manual process management. In this section we present the formalization of the mechanisms for customizing and automating process management. The discussion follows the line of subsection 3.3.

Structural restrictions (cf. fig. 3) are expressed in the PROGRES graph schema. Type-valued attributes are introduced which restrict the application of edit operations when trying to introduce a subtask of wrong type, to violate the cardinality, or to violate the task order. An example for the use of type-level attributes was given in figure 7. The input parameter `FBType` denotes the node type of the feedback relation to be created. The attributes `SourceTypes` and `TargetTypes` constrain legal source and target types for a given feedback type. In the conditions, these attributes are queried in order to check the structural constraints of a specific schema.

Since PROGRES does not provide an event-trigger mechanism (cf. fig. 4 in subsection 3.3), we have to simulate it using synchronous control flow. To this end, all basic operations are wrapped into a transaction which first applies the basic operation and subsequently calls a specific event handler (cf. fig. 9 for the `CreateFeedback` operation)[7]. Within

[7] Pre-events are handled analogously.

```
test GraphCode_FBPattern( FB : FEEDBACK ; out DepTasks : TASK [1:n])
=
```




```
  condition ‘4.Name = "GraphCode";
            ‘5.Name = "SemAna";
            ‘6.Name = "CodeGen";
  return
    DepTasks := ‘4 or ‘5 or ‘6 or ‘7 or ‘8 or ‘9 or ‘10 or ‘11;

end;

transaction GraphCode_FBHandler( FB : FEEDBACK) =
  use taskset : TASK [1:n];
      NewTask : TASK
  do
      GraphCode_FBPattern ( FB, out taskset )
    & for all t := taskset
      do
        choose
          Transition ( t, Suspend )
        else
          DeriveTask ( t, out NewTask )
        else
          skip
        end
      end
  end
end;
```

Figure 11: Pattern mechanism

the event handler, the user-defined trigger definitions are compiled which determine the behavior of tasks in the context. In our example, according to the type of the receiver task (`ReceiverT`), the corresponding trigger routine is executed. After that, the same procedure is performed for the originator of the feedback relation (`SenderT`). By means of this mechanism, we specified event handlers reacting on structural modifications, state transitions, consumption/production of tokens, etc.

Conditions are an important mechanism when customizing the execution behavior of dynamic task nets. The application of an operation is controlled by user-defined conditions (cf. subsection 3.3). For example, in the graph production for creating a feedback relation (cf. fig. 7) the derived attributes `FeedbackOutCond`, `FeedbackInCond`, and `FeedbackCond` are checked on the source task, the target task, and the parent task(s), respectively. For the complex task of software development this condition is refined by the function `HasNoActiveFeedback` which requires that no active feedback relation is allowed when introducing a new one (cf. fig. 10). Thus, a very restrictive policy is followed which allows only one active feedback relation at a time. Less restrictive policies may be specified by similar restrictions. The function is evaluated as follows: It is checked for a development task whether the restriction HasActiveTask does not hold. The restriction states that starting from node '1, two child tasks have to be found which are connected by a FEEDBACK relation. This feedback relation must be still active, which is expressed in the condition part of the restriction.

The last mechanism we want to present is the specification of propagation patterns (cf. fig. 5 in subsection 3.3) as shown in figure 11. If enough information about the process to perform is available, propagation patterns at the instance level are used to specify situations for which appropriate reactions have to be performed. In our approach, a situation is a certain graph pattern found in the current graph. In PROGRES, graph patterns are specified as graph tests (cf. fig. 11). The GraphCode_FBPattern test searches for a feedback relation from an implementation task for a GraphCode module back to the Design task and collects all dependent tasks which are affected by the feedback. This pattern is searched, every time a feedback relation is introduced within the net of compiler construction. The transaction FBHandler_GraphCode is the trigger routine for feedback introduction. Every time a feedback relation is introduced, it is checked whether the graph pattern could be found. In the negative case, nothing happens. In the positive case, the test returns all affected tasks in the taskset variable. These tasks are subsequently either suspended or — in case they were already terminated — a new task version is introduced for them. If a task is not started yet, no operation is done (skip statement).

5 Conclusion

We have described how feedback may be handled in dynamic task nets. A general overview of DYNAMITE is given in [15]. The underlying formal specification is also discussed in [14]. Its generic (domain-independent) kernel currently consists of about 100 node classes and 130 graph rewrite rules. It has reached a size of about 80 pages. Several domain-specific process models have been encoded on top of the generic model, including e.g. the ISPW6 example.

We are developing a process-centered software engineering environment which is based on the formal specification [17]. To this end, we are using (and improving) the PROGRES development environment which offers tools for editing, analyzing, and interpreting PROGRES specifications, and for compiling them into end-user prototypes. While the specification itself can be compiled automatically into source code, considerable effort is required to design and implement customized user interfaces for software engineers and managers (as the reader will have noticed, the internal graph representation is hardly readable and needs to be converted into a suitable external representation). An initial version of the DYNAMITE environment was presented at ICSE '97 in conjunction with the PROGRES environment [32].

Furthermore, experience has shown that some aspects of coding a process model in PROGRES are time-consuming and cumbersome. Thus, additional tools are required above the PROGRES level in order to generate "routine parts" of a PROGRES specification [21].

In this paper, we have focused on feedback handling at the instance level, i.e., within *software production processes*. We have considered neither the *meta process* of developing process models (see e.g. [26, 27]) nor the interactions between meta process and production process. Several authors apply the term "feedback" to the control loop between meta process and production process (see e.g. feedback modifications in MELMAC [13]).

In particular, implementation of the control loop requires some means to change the model of an ongoing production process. Late binding and reflection have been recognized as important requisites to perform such changes [1, 4]. However, consistency maintenance has not been addressed satisfactorily yet. Approaches developed in the database field do address consistency maintenance, but either force all data under one schema (schema evolution) or restrict applications to view data under one schema at a time (schema versioning [29]).

Future research will be targeted at developing a more general approach. In case of dynamic task nets, the past (terminated tasks) must not be changed (traceability), the present (active tasks) may be changed selectively, and the future (planned tasks) should be changed. Updates implied by schema changes will be specified by graph productions. In contrast to automatic approaches developed in the database field, human intervention is essential to control the updates.

Unlike [7], we have not yet considered the interaction between a real-world process and its enacted model (the latter is represented as an instance-level task net, the former is not represented at all). Effectively, their work requires to add another level of modeling (descriptive model of the actual real-world process) and mechanisms to detect and resolve inconsistencies with respect to the model of the enacted process. Definition of flexible processes (process definition domain [9]) and supporting changes to process definitions during enactment (integration between process definition domain and process enactment domain) may obviate the need for such a challenging approach.

References

[1] S. Bandinelli and A. Fuggetta. Computational Reflection in Software Process Modelling: The SLANG Approach. In *Proc. of the* 15^{th} *International Conference on Software Engineering*, pages 144–154, Baltimore, Maryland (USA), May 1993.

[2] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Software Process Model Evolution in the SPADE Environment. *IEEE Trans. Software Eng.*, 19(12):1128–1144, Dec. 1993.

[3] V. R. Basili, G. Caldiera, and H. D. Rombach. Experience Factory. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons, 1994.

[4] G. A. Bolcer and R. N. Taylor. Endeavors: A Process System Integration Infrastructure. In *Proceedings of the 4th International Conference on the Software Process*, pages 76–89, Brighton, England, Dec. 1996. IEEE Computer Society Press.

[5] G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart, and J. Lonchamp. ALF: A Framework for Building Process-Centered Software Engineering Environnments. In Finkelstein et al. [12], pages 153–185.

[6] R. Conradi, editor. *Proceedings of the 7th International Workshop on Software Configuration Management*, LNCS 1235. Springer-Verlag, 1997.

[7] G. Cugola, E. di Nitto, A. Fuggetta, and C. Ghezzi. A framework for formalizing inconsistencies and deviations in human-centered systems. *ACM Transactions on Software Engineering and Methodology*, 5(3):191–230, July 1996.

[8] B. Dellen, K. Kohler, and F. Maurer. Integrating software process models and design rationales. In *Proceedings of the 11th Knowledge-Based Software Engineering Conference*, Syracuse, New York, 1996. IEEE Computer Society Press.

[9] M. Dowson and C. Fernström. Towards requirements for enactment mechanisms. In B. C. Warboys, editor, *Proceedings 3rd European Workshop on Software Process Technology (EWSPT '94)*, LNCS 772, pages 90–106. Springer Verlag, 1994.

[10] J. Estublier, S. Dami, and M. Amiour. High level process modeling for SCM systems. In Conradi [6], pages 81–97.

[11] C. Fernström. PROCESS WEAVER: Adding Process Support to UNIX. In *2^{nd} International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, Feb. 1993. IEEE Computer Society Press.

[12] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology.* Research Studies Press, Taunton, Somerset, England, 1994.

[13] V. Gruhn. *Validation and Verification of Software Process Models.* PhD thesis, Universität Dortmund, 1991.

[14] P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. A Programmed Graph Rewriting System for Software Process Management. In *Proceedings Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA '95)*, volume 2 of *Electronic Notes in Theoretical Computer Science*, 1995.

[15] P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. DYNAMITE: Dynamic Task Nets for Software Process Management. In *Proceedings of the 18th International Conference on Software Engineering*, pages 331–341, Berlin, Mar. 1996. IEEE Computer Society Press.

[16] P. Heimann, C.-A. Krapp, M. Nagl, and B. Westfechtel. An adaptable and reactive project management environment. In Nagl [25], pages 504–534.

[17] P. Heimann, C.-A. Krapp, and B. Westfechtel. An Environment for Managing Software Development Processes. In *Proceedings of the 8th Conference on Software Engineering Environments*, Cottbus, Germany, Apr. 1997. IEEE Computer Society Press. To appear.

[18] G. T. Heineman, G. E. Kaiser, N. S. Barghouti, and I. Z. Ben-Shaul. Rule Chaining in MARVEL: Dynamic Binding of Parameters. *IEEE Expert*, 7(6):26–32, Dec. 1992.

[19] M. L. Jaccheri and R. Conradi. Techniques for Process Model Evolution in EPOS. *Transactions on Software Engineering*, 19(12):1145–1156, Dec. 1993.

[20] T. Katayama. A Hierarchical and Functional Software Process Description and its Enaction. In *Proceedings of the 11th International Conference on Software Engineering*, pages 343–352, Pittsburgh, Pennsylvania (USA), May 1989. IEEE Computer Society Press.

[21] C.-A. Krapp. Parametrisierung dynamischer Aufgabennnetze zum Management von Softwareprozessen. In *Tagungsband Softwaretechnik '96*, pages 33–40, Koblenz, Germany, Sept. 1996.

[22] D. Leblang. Managing the software development process with ClearGuide. In Conradi [6], pages 66–80.

[23] C. Liu and R. Conradi. Automatic Replanning of Task Networks for Support Process Model Evolution in EPOS. In I. Sommerville and M. Paul, editors, *Proceedings of the European Software Engineering Conference '93*, LNCS 717, pages 434–450, Garmisch-Partenkirchen, Sept. 1993. Springer-Verlag.

[24] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the European Software Engineering Conference (ESEC '95)*, LNCS 989, pages 137–153, Barcelona, Spain, Sept. 1995. Springer Verlag.

[25] M. Nagl, editor. *Building Tightly-Integrated Software Development Environments: The IPSEN Approach.* LNCS 1170. Springer-Verlag, Berlin, 1996.

[26] M. N. Nguyen and R. Conradi. Classification of Meta Processes and their Models. In *Proceedings of the 3rd International Conference on the Software Process*, pages 167–175, Washington, 1994. IEEE Computer Society Press.

[27] M. N. Nguyen and R. Conradi. Towards a Rigorous Approach for Managing Process Evolution. In C. Montangero, editor, *Proceedings 3rd European Workshop on Software Process Technology (EWSPT '96)*, LNCS 1149, Nancy, France, 1996. Springer Verlag.

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

[29] J. F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information Software and Technology*, 37(7):383–393, July 1995.

[30] A. C. Sarma. Introduction to SDL-92. *Computer Networks and ISDN Systems*, 28:1603–1615, 1996.

[31] A. Schürr. Introduction to the Specification Language PROGRES. In Nagl [25], pages 248–279.

[32] A. Schürr. Developing graphical (software engineering) tools with PROGRES. In *Proceedings 19th International Conference on Software Engineering*, pages 618–619, Boston, MA, 1997. IEEE Computer Society Press.

[33] A. Schürr, A. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In *Proceedings of the European Software Engineering Conference (ESEC '95)*, LNCS 989, pages 219–234, Barcelona, Spain, Sept. 1995. Springer Verlag.

[34] M. Suzuki, A. Iwai, and T. Katayama. A Formal Model of Re-execution in Software Process. In *Proc. of the Second International Conference on the Software Process*, pages 84–99, Berlin, Germany, Feb. 1993. IEEE Computer Society Press.

[35] B. Westfechtel. A Graph-Based System for Managing Configurations of Engineering Design Documents. *International Journal of Software Engineering and Knowledge Engineering*, 6(4), Dec. 1996.

[36] P. Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users.* PhD thesis, University of California Irvine, 1994.

[37] A. Zamperoni. GRIDS – graph-based, integrated development of software: Integrating different perspectives of software engineering. In *Proceedings of the 18th International Conference on Software Engineering*, 1996.