DYNAMITE: Dynamic Task Nets for Software Process Management*

Peter Heimann, Gregor Joeris, Carl-Arndt Krapp, Bernhard Westfechtel
Lehrstuhl für Informatik III, RWTH Aachen
Ahornstr. 55, 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. Dynamic task nets take this into account. They are built and modified incrementally as a software process is executed. Dynamic task nets have been designed to solve important problems of process dynamics, including product-dependent structural evolution, feedbacks, and concurrent engineering. In order to describe editing and enactment (and their interaction) in a uniform way, task nets are formally defined by means of a programmed graph rewriting system.

1 Introduction

Managing the software development and maintenance process has been identified as a great challenge for several years [3]. Only rarely can complex software processes be planned completely in advance. 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 tasks be performed.
2. *Concurrent engineering*: In order to shorten development cycles, concurrent engineering [18] proposes methods to increase concurrency in the development process. To this end, cooperation between software engineers must be enhanced such that reasonable intermediate results may be delivered as soon as possible. As a result, each task is executed in a highly dynamic work context.
3. *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.

The *DYNAMITE* approach (DYNAMIC TASK NETS [22, 8]) has been designed to meet these requirements. DYNAMITE is characterized by the following features:

1. Editing and enactment of task nets are highly intertwined. This means that the structure of a task net evolves during execution.
2. The interface of a task (what) is separated from its realization (how).
3. Tasks are arranged in a hierarchy.
4. Within a subnet, tasks are connected by acyclic control flow relations. These relations are used to control the order in which tasks are executed.
5. In addition, feedbacks are represented by relations which are oriented oppositely to control flow relations.
6. Reactivation of tasks is modeled by creating new task versions. In this way, traceability of change processes is supported.
7. Furthermore, tasks are connected by data flow relations.
8. During enactment, a task may consume and produce sequential versions of inputs and outputs, respectively.
9. Enactment takes the hierarchy into account (i.e., the hierarchy is not flattened) and respects the abstraction principle introduced in 2.
10. The evolution of task nets is controlled by a schema which defines domain-specific types of tasks and relations.

Dynamic task nets are formally defined in *PROGRES*, a specification language which is based on programmed graph rewriting systems ([19], see e.g. [13, 7] for related approaches). The formal specification is developed using the PROGRES environment,

*This work has partly been supported by the German Research Council (DFG)

0270-5257/96 $5.00 © 1996 IEEE
Proceedings of ICSE-18

	DYNAMITE	MELMAC	SPADE	EPOS
formalism	graph rewriting	Petri nets	Petri Nets	EER
instantiation	instance-level nets	populated copies	populated copies	instance-level nets
abstraction	interface hides realization	no abstraction	interface hides realization	no abstraction
hierarchy	design and execution	design	design and execution	design and execution
horizontal relations	control flow, feedback, data flow	token flow (control and data)	token flow	data flow
interleaving of editing and enactment	fine-grained	coarse-grained	coarse-grained	fine-grained
support of feedbacks	explicit (feedback relations)	implicit (cycle)	implicit (cycle)	implicit (replanning)
concurrent engineering	pre-releases of intermediate versions	no specific support	no specific support	cooperating transactions
traceability	task versions	no specific support	no specific support	no specific support

Table 1: Comparison with other approaches

which provides tools for editing, analyzing, interpreting, and compiling PROGRES specifications [20]. Operations on task 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 task nets precisely and unambiguously on a high level of abstraction. Since the specification is executable, a software process management system may be generated from the specifaction.

2 Related work

Many different paradigms have been applied to software process management, including rule bases [12, 11], blackboards [14], process programs [17], events and triggers [2], state charts [9], and object orientation [5]. DYNAMITE follows a net-based paradigm. Nets allow for a natural, graphical representation of complex software processes. They suit the needs of project managers (planning and control of software projects), but they also support software engineers (agendas, work contexts).

Table 1 compares DYNAMITE to some well-established approaches (MELMAC [4], SPADE [1], and EPOS [10]). MELMAC and SPADE rely on Petri nets, EPOS is based on an EER approach, and graph rewriting provides the formal foundation for DYNAMITE. With respect to task instantiation, these approaches are classified into two categories:

1. In the first category, *instance-level nets* are maintained where each node represents a specific task instance. EPOS and DYNAMITE belong to this category.
2. In the second category, *type-level nets* are instantiated by creating copies which are populated with tokens. Here, each transition represents a task type rather than a task instance. Of course, individual task instances are created and managed, as well (e.g. for constructing an agenda). But instance-level nets are not maintained, neither for logging nor for planning.

Instance-level nets provide detailed and specific information on the status of a project. In this respect, they are superior to (copies of) type-level nets with a project-invariant structure.

Since DYNAMITE is based on instance-level nets, it differs considerably from MELMAC and SPADE. Furthermore, the table shows that there are still a lot of differences between DYNAMITE and EPOS, which is closest to our approach. In contrast to EPOS, interfaces and realizations are strictly separated in DYNAMITE, and abstraction is enforced (the interface hides the realization). Furthermore, DYNAMITE distinguishes between multiple types of horizontal relations (control flow, feedback, data flow), while EPOS only supports one type of relation (data flow). Concurrent engineering is supported in EPOS only between different workspaces (each of them containing

a task tree), but not between individual tasks within one workspace. In DYNAMITE, feedbacks are represented explicitly by relations which introduce cycles into the task net. In EPOS, feedbacks are triggered by asserting error conditions. There is no explicit relation between the triggering task and the triggered task; the task net does not contain cycles.

3 Informal description

In this section, we describe our approach to software process management in an informal way. In subsection 3.1, we discuss the structure of task nets. In subsection 3.2, we sketch how task nets may be adapted to a certain application domain. Subsection 3.3 describes task enactment in general terms. Finally, subsection 3.4 illustrates various applications of our approach. In particular, we show how structural evolution, concurrent engineering, and feedbacks are supported.

3.1 Task nets

A *task* is an entity which describes work to be done. The *interface* of a task specifies what to do. In particular, it describes inputs, outputs, preconditions, postconditions, start dates, due dates, etc. The interface serves as an abstraction which hides the realization.

The *realization* of a task describes how to do the work. In general, a given interface may be realized in multiple ways (note the analogy to module interfaces and realizations in programming-in-the-large). For example, consider development of a software system which consists of multiple subsystems. For each subsystem development task, an appropriate realization may be selected. S1 might be developed according to a waterfall model, while prototyping might be chosen for the development of S2. Finally, S3 might be delegated to a subcontractor. On the client's side, the realization would then describe all activities required to communicate with the subcontractor and to supervise his work.

We distinguish between atomic and complex realizations. In case of an *atomic realization*, a task is not refined into subtasks. Typical examples are editing and compiling of modules. A *complex realization* consists of a net of subtasks which are connected by various kinds of relations to be described below.

Control flow relations impose an ordering on the subtasks to be enacted; they resemble the ordering relations found in net plans. Fig. 1 shows a sample net of tasks connected by control flow relations. The subsystem consists of modules A,...,D, where B and C import from A and D imports from B and C (see fig. 2). Subsystem design is followed by concurrent

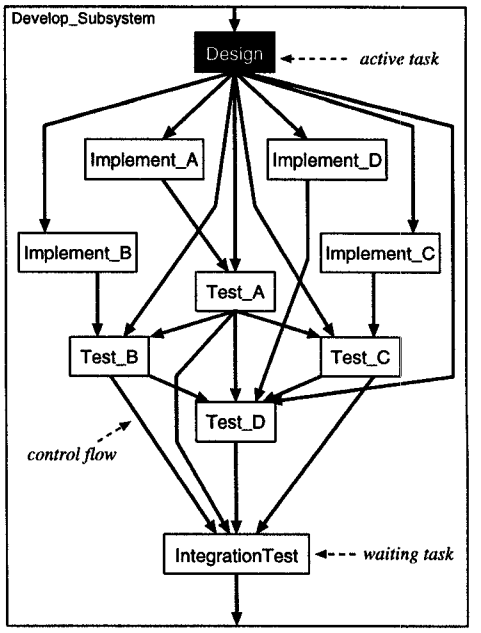


Figure 1: Sample task net (control flow view)

implementation of all modules. Subsequently, modules are tested in a bottom-up fashion. Finally, an integration test is performed.

Control flow relations span an acyclic, connected graph which acts as the skeleton of the task net. In order to represent feedbacks, *feedback relations* are introduced which are oriented in the opposite direction. For example, fig. 3 shows a feedback from the integration test to the implementation of module B.

Fig. 3 also illustrates task versions and successor relations. When a terminated task has to be reactivated later on, a new *task version* is created. Formally, task versions are connected by *successor relations*. In fig. 3, versioning is illustrated by drawing the successor on top of its predecessor. In general, a new task version is created either because of changing inputs or because of a feedback (see also subsection 3.4).

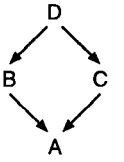


Figure 2: Subsystem architecture

Data flow relations are used to transmit data between tasks connected by hierarchical, control flow, feedback, or successor relations. There are three cases

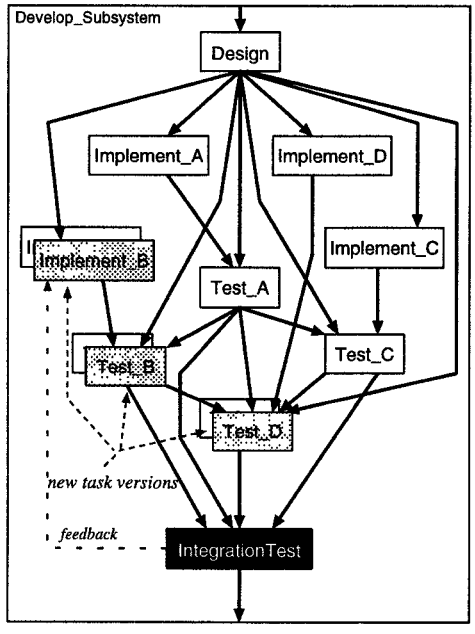


Figure 3: Feedback between tasks

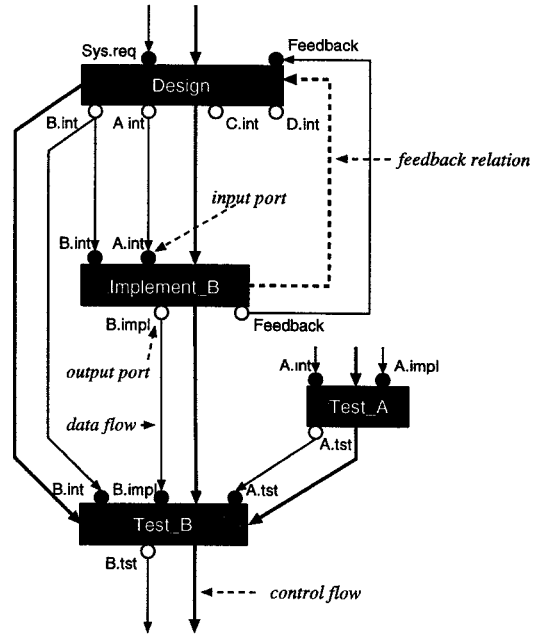


Figure 4: Sample task net (data flow view)

to be distinguished:

- Input → input: The input of a supertask is passed to a subtask.
- Output → input: The output of a subtask is propagated to another subtask.
- Output → output: The output of a subtask is passed upwards to the supertask.

To illustrate data flow relations, fig. 4 refines a cutout of the task net displayed in fig. 1. The design task receives the requirements definition from its supertask and creates interfaces for all modules. The implementation task for B consumes the export interface to be implemented and the interface of the imported module A; it creates an implementation for B. To illustrate feedbacks, a feedback relation goes back to the design; this relation is refined by a data flow carrying a bug report. Finally, the test task for B receives B's implementation and a tested implementation of A; it delivers a tested implementation of B.

3.2 Levels of modeling

Dynamic task nets need a computational representation. For this purpose we distinguish between three levels of modeling: generic model, specific model, and instances.

The *generic model* is built-in in and is independent of an application area. While this paper focuses on software engineering, the generic model can also be applied to other domains such as Computer Integrated Manufacturing (CIM) or office automation. It factors out all common properties of dynamic nets in different scenarios. It introduces the notions of task, input, output, data flow, etc. and provides a standard state transition diagram, which can be adapted on the next layer down.

The *specific model* is used to adapt the generic model to a specific application domain. Fig. 5 shows the structural part of a specific model for the nets described in the previous subsection. Task types are represented by ellipses. For each type, the model defines the minimum and maximum numbers of instances. 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 task instance may have. Besides the structural adaption of the task net, execution semantics can also be adapted on the level of the specific model. In this way, different execution strategies can be implemented for different project needs (see also subsection 3.3).

The structural part of the specific model constrains the task net which is created on the instance level. A task net is used to model the actual state and the planned steps of a development process.

3.3 Enactment

Enactment is defined such that *evolving task nets* are supported. Task nets need not be built completely before enactment starts. Rather, they can be modified incrementally during enactment. In particular, modifications to a task net do not require its enactment

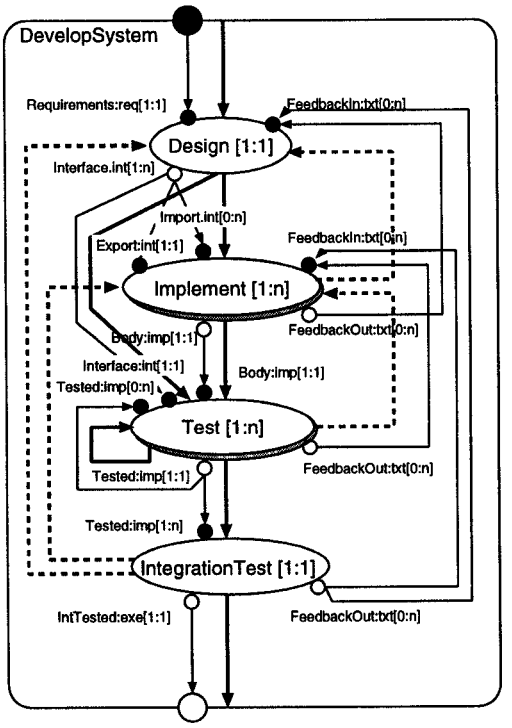


Figure 5: Model for task nets

to be suspended. Rather, subtasks not affected by the modification can continue execution while the net is being modified. This does not exclude suspension of the whole net if large modifications are intended.

Each task has an *enactment state* which is controlled by the state transition diagram shown in fig. 6. This diagram uniformly applies to all tasks, regardless of their position in the task hierarchy. It is a fixed part of the meta model which cannot be modified when defining a specific model for some application domain. However, there are still sufficient means to adapt the enactment behaviour. The conditions for state transitions may be redefined according to the needs of the application domain (see below).

`In_Definition` serves as initial state where a task is defined, i.e. its interface is described (inputs to be consumed, outputs to be produced, etc.). The `Defined` transition indicates that the task definition is completed and may now be enacted. If necessary, `Redefine` may be used to perform the inverse transition. As soon as its activation condition is fullfilled, `Start` may be used to move the task from `Waiting` to `Active`. Enactment may be interrupted by performing the `Suspend` transition into the state `Suspended`; `Resume` reverts this transition. There are two final states. `Done` indicates that the task has been completed successfully (transition `Commit`). In contrast,

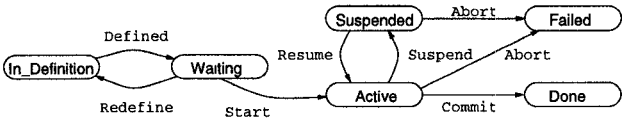


Figure 6: State transition diagram

`Failed`, which is reached by an `Abort` transition, means that the task could not be completed successfully.

Default conditions are provided for all state transitions. These defaults may be overridden as required. In this way, a broad spectrum of behaviors may be defined according to the requirements of a certain application domain. For example, in a very formal setting, the start condition may state that a task may only be started when all predecessors (with respect to control flow relations) have been terminated successfully. On the other hand, in a concurrent engineering setting we may relax this condition such that pre-releases of certain (not necessary all) inputs are available. These adjustments may be performed individually for each task type. Therefore, the enactment behaviour can still be *customized* although the state transition diagram is fixed.

Finally, let us comment on the role of the *task hierarchy* with respect to enactment. In some process management systems, the hierarchy has no meaning for task enactment (see also section 2), i.e. enactment operates on a flat net of atomic tasks. This approach is not consistent with our distinction between interface and realization. If the hierarchy is flattened, there is no abstraction, i.e. a task does not hide its realization any more. We want to retain the property that a task interacts with its neighbors only through its interface. Therefore, the hierarchy has to be preserved for enactment.

In particular, abstraction implies that a subtask T1 in subnet N1 has no direct connection to a subtask T2 in a different subnet N2. Communication has to be performed via their supertasks. This might appear a bit awkward, but cannot be avoided without violating the abstraction. However, passing data up and down the hierarchy may be automated. Communication via supertasks cannot be avoided without violating abstraction.

3.4 Evolving nets, concurrent engineering, and feedbacks

During execution, the structure of task nets usually changes. In particular subnets may depend on the contents of the documents produced and therefore cannot be planned in advance. The development of e.g. a simple software subsystem starts with the ini-

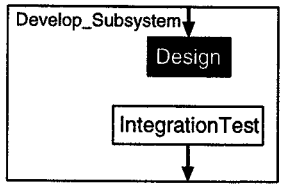


Figure 7: Initial net (control flow view)

tial net of fig. 7. After the `Design` task has produced a coarse description of the architecture such as given in fig. 2, the task net is extended as shown in fig. 1. For every module, one implementation and one test task is inserted. Control and data flows are derived from import relations and the test strategy. In this example, the process engineer has chosen a bottom up test strategy. In general, changes are not constrained to extensions. If the architecture gets modified later on, this will induce a change of the task net. If the bottom up test strategy is replaced with a top down strategy, this again will lead to a net modification.

We do not enforce a strictly phase-oriented approach, but instead allow concurrent engineering. In our example, the depending implementation tasks may start before the `Design` task is completed. During enactment, an active task can produce several versions of its outputs, and conversely consume several input versions. The net keeps track of which tasks have produced and consumed which versions. In fig. 8, the `Design` task has first delivered an incomplete version $\texttt{A.int}_1$ of the interface definition of module A. This version has been used to start implementing modules `B` and `C`. Since then, `Design`, `Implement_B`, and `Implement_C` have been active concurrently. Later on, the `Design` task has created a completed version $\texttt{A.int}_2$. Fig. 8 shows the data flow view after `Implement_C` has read the new version, but while `Implement_B` is still using the old one. The state transition conditions make sure that `Implement_B` cannot be terminated successfully in this situation.

The development of complex technical documents does not always proceed smoothly. If a task detects an error in one of its input documents, a feedback flow is added to the net (dashed arrow in fig. 3). Along this flow, the bug is reported to the producer of the faulty document. The specific model (see subsection 3.2) governs which feedbacks are allowed. Actual flows are, however, added to the task net only when needed so the net does not get cluttered. If the task which produced the erroneous document is still active, it uses the new bug report input to create a corrected version of the output document. If the task has been terminated, the situation is more severe. In this case, we

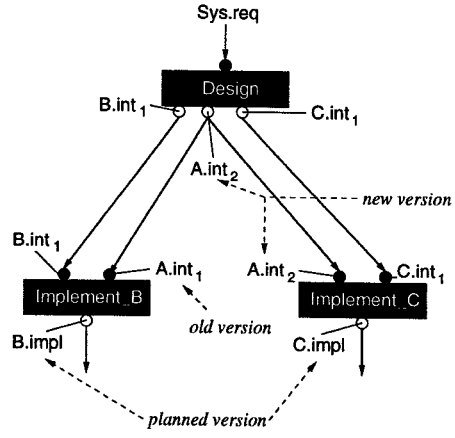


Figure 8: Different document versions

want to give the project manager the chance to later trace what has happened. Furthermore, the person responsible for the original task might be no longer available (she might even have left the company), and we have to assign a different actor this time. Therefore, we do not reactivate the old task, but create a new version instead. Inputs to this new version are the feedback flow and copies of the input flows of the original task. Subsequent tasks that may be affected by the corrected output version can be automatically deduced. For the terminated ones among them, new versions are created as well.

4 Formal specification

In this section, we will be concerned with the developer's view of an environment for dynamic task nets. To define the semantics of such nets, we use the specification language PROGRES, which is based on *programmed graph rewriting systems*. Furthermore, the PROGRES environment provides tools for generating an end-user environment from a specification.

4.1 Overview

PROGRES is a specification language which is based on attributed graphs. A PROGRES specification — a specification written in PROGRES — consists of two parts. The *graph schema* defines types of nodes, edges, and attributes. Subsequently, operations are specified which conform to the graph schema. This is done on a high level of abstraction. Instead of describing a graph transformation in terms of elementary operations such as creation and deletion of single nodes and edges, a *graph rewrite rule* — also called production — describes replacement of a whole subgraph in a declarative way. The PROGRES compiler takes care of all algorithmic details for performing graph transformations efficiently.

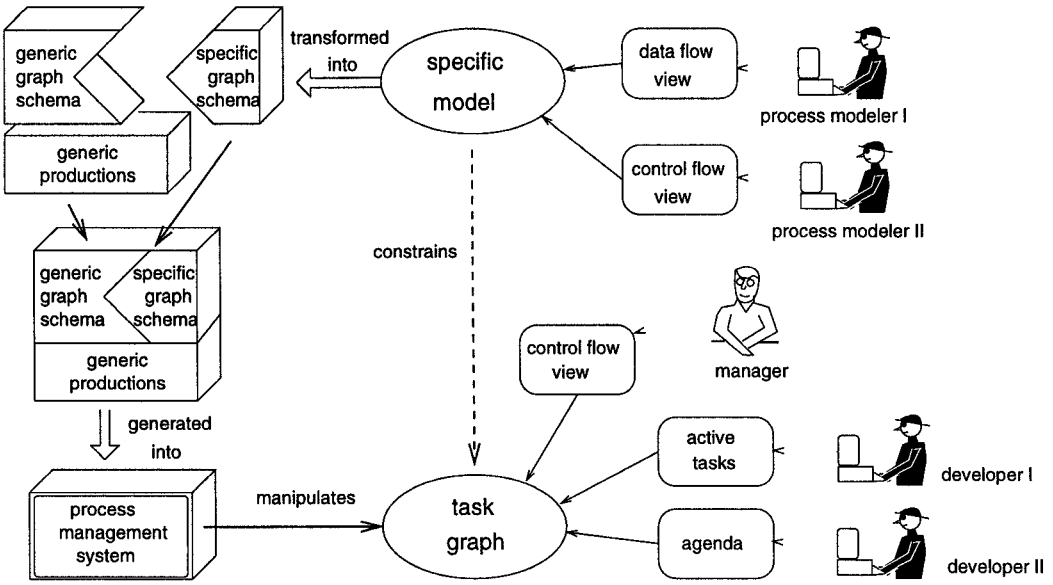


Figure 9: Generic approach for dynamic task nets

We follow a *generic approach* to the design and implementation of an environment for dynamic task nets which is described in fig. 9. Recall that we distinguish between three levels of modelling: generic model, specific model, and instances (subsection 3.2). The generic model is defined by a graph schema and a set of productions which are independent of a specific application domain. The specification of this model contains formal generic parameters which are used to parameterize generic productions. By supplying corresponding actual parameters, the productions are adapted to a specific application domain. To this end, a domain specific part is added to the generic graph schema (upper left corner of fig. 9). In the simplest case, we may adapt the generic model without writing 'code' (productions). In general, we may have to write domain specific productions as well (not shown in the figure).

The PROGRES specification is used to generate a process management system adapted to a specific application domain (lower left corner of fig. 9). The process management system manipulates instance-level task nets, represented by task graphs.

Different *views* on task graphs are offered according to varying user roles (lower right corner of fig. 9). As we will show below, task graphs are complex internal data structures which cannot be presented directly to the users of a process management system. More high-level and condensed presentations are needed which hide the complexity of the underlying task graphs. For example, a control flow view of a task net as shown in fig. 1 may be presented to a project manager. Furthermore, a developer may be supplied with an agenda, i.e. a textual list of tasks to be carried out.

For similar reasons, we want to shield process modelers from the inherent complexity of PROGRES specifications. For example, a diagram as shown in fig. 5 may be used to define a specific model in a convenient way (upper right corner of fig. 9). Such a diagram can be transformed into the domain specific part of the PROGRES specification. Hence, process modelers need to use PROGRES only for further modifications which go beyond simple standard adaptations.

In the remainder of this section, we describe the specification of the DYNAMITE model in a more detailed way.

4.2 Graph schema

The graph schema serves as a database schema which defines types of objects, relations, and attributes. In PROGRES, objects are modeled as nodes, and properties of objects are represented by node attributes. Edges represent binary relations between nodes and do not carry attributes. N-ary relations, attributed relations, and relations participating themselves in relations are modeled by nodes and adjacent edges.

Fig. 10 displays a graphical schema for the generic model. Each box represents a *node class*. Node classes declare attributes for nodes and are organized into a multiple inheritance hierarchy. A subclass inherits from its superclasses all attributes and all incoming and outgoing *edge types*, which are drawn as labeled

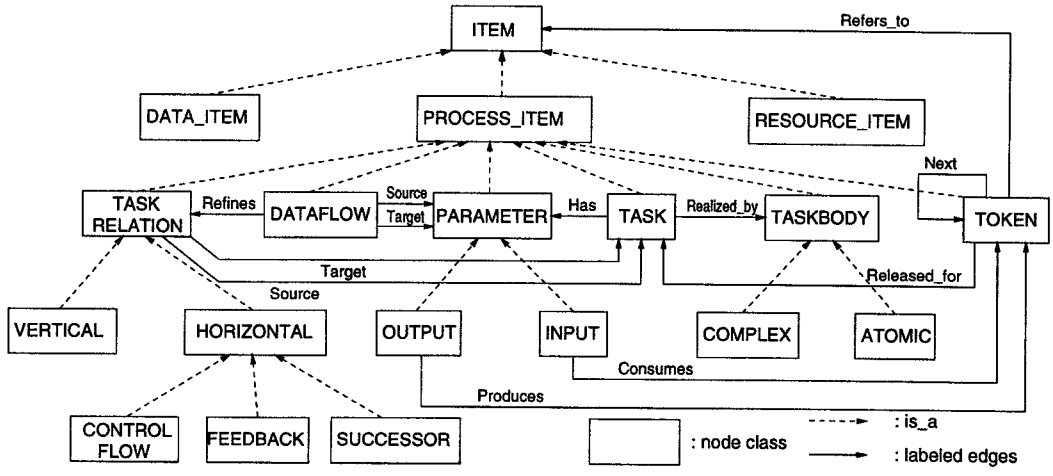


Figure 10: A PROGRES graph schema for the meta model

solid arrows. For the sake of readability, attributes are not shown in fig. 10.

Node class 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. A TASK is realized by a complex or atomic TASKBODY and has some input and output parameters. Control and data flow relations between process items are modeled as nodes. Vertical relations exist between a task and all its task children. Horizontal relations connect sibling tasks. Data flow relations refine the corresponding task relations. The data flow is modeled by a token game. Each token refers to some item. Arbitrary items are permitted by the meta model as inputs of tasks, including products, tasks and resources. Released_for edges indicate to which tasks a certain token has been released. In order to keep the development information about produced and consumed data, the tokens are not deleted. They are collected in a token list linked by Next edges.

In order to adapt the generic model, specific types of tasks, parameters, documents, etc. have to be defined. In PROGRES, *node types* are defined as instances of node classes. PROGRES has a stratified type system: nodes are instances of node types, which in turn are instances of node classes. The generic model and the specific model are defined on the level of node classes and node types, respectively.

As outlined in the previous subsection, we follow a generic approach: The generic model introduces formal parameters which are replaced by actual parameters in the specific model. In our specification, we model formal generic parameters as type-level attributes (called *meta attributes* in PROGRES). A meta attribute is attached to a node class or node

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

Figure 11: Cutout of specific model

type; its value is type- rather than instance-specific. On the level of the generic model, uninitialized meta attributes are introduced. On the level of the specific model, values are assigned to meta attributes. These values serve as actual parameters bound to formal generic parameters.

As an example, fig. 11 shows the specification of the task Implement, which is introduced as part of the specific model. Recall that process modelers have a more user friendly view on the specifc model (cf. fig. 5). The type-level attribute In of the type Implement is initialized with the types Export, Import and FeedbackIn. This means that an im-

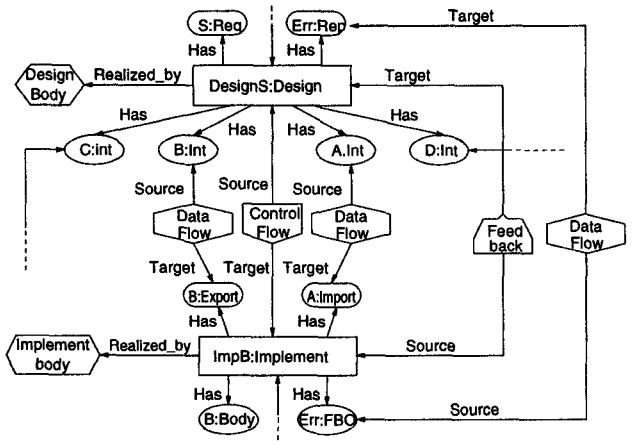


Figure 12: Cutout of a task graph

plementation task receives the export interface, import interfaces, and error reports resulting from a feedback as input parameters. All possible realizations for the `Implement` task are listed in the attribute `Realizations`. The meta attributes cannot be changed on the task 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 to specify the cardinalities of the parameters.

After we have described the abstract structure of task graphs in terms of node classes and node types, we give an idea of how such graphs look like. A *task graph* simply consists of attributed nodes and labeled edges. Fig. 12 shows a cutout of a graph representing the task net shown in fig. 1. For the sake of readability, we omit the node attributes. The presented graph is a snapshot since it evolves dynamically over time. The nodes are of different classes and types (indicated by symbols and type identifiers, respectively). For instance, the graph contains a node `ImpB` of type `Implement`, which in turn is an instance of node class `TASK` defined in the schema of the meta model.

4.3 Graph operations

Operations on the task graph are specified on the level of the meta model. Since they refer to values of meta attributes and receive node types as actual parameters, they are implicitly adapted to a specific model by defining specific node types and assigning values to meta attributes (see above).

Operations on the task graph are specified by means of graph rewrite rules (called productions in PROGRES). A *graph rewrite rule* describes a graph transformation and consists essentially of a left-hand side, which defines the graph pattern to be replaced, and a right-hand side, which defines the replacing graph. In this way, operations on the complex data structure representing the task net can be easily specified without regarding algorithmic details concerning the pattern match, graph replacement, etc. These details are implemented within the PROGRES system.

A comprehensive discussion of transformations on task graphs is beyond the scope of this paper ([15]). We present an example of an edit operation, which changes the structure of the task net. Fig. 13 presents an operation to insert a feedback from a successor `s` to a predecessor `p` into the task net. A lot of steps are performed in a single graph rewrite rule:

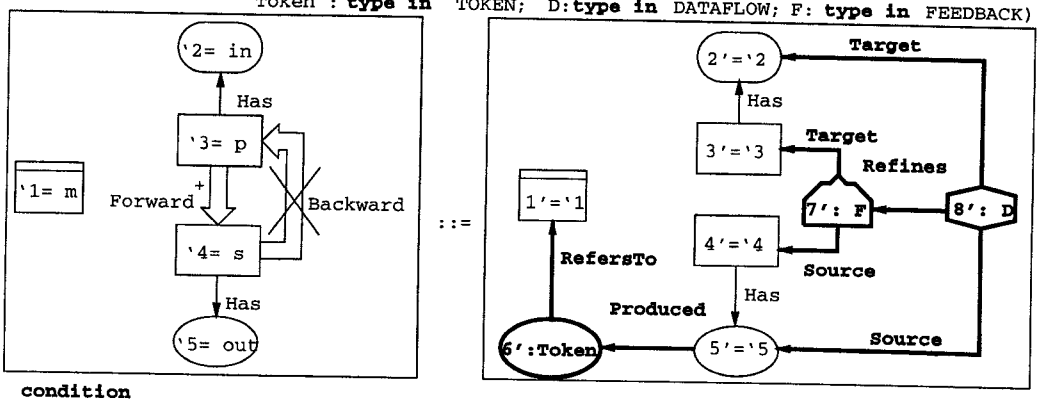
1. The rule checks whether `s` is a (directe or transitive) successor of `p`. Furthermore, there must not yet exist a feedback relation from `s` to `p`.
2. The rule is only applicable if `s` resides in state `Active`.
3. Furthermore, the rule checks whether any constraint of the specific model is violated.
4. A feedback relation from `s` to `p` is created.
5. Furthermore, a refining dataflow is created which transmits an error message `m`.

The rule receives tasks `s` and `p`, the message `m`, the parts to be connected (`out` and `in`), and the types of token, data flow, and feedback relation as input parameters (`Token`, `D`, and `F`, respectively).

The *left-hand side* of the rule consists of nodes, edges and paths. In this example, all nodes are fixed by input parameters, e.g. node `'2` is fixed by parameter `in`. A path is indicated by a double arrow and is used to navigate through the task graph. For instance, the path expression `Forward` searches a path from a `TASK` node to another `TASK` node via a reverse `Source` edge, a node instance of class `CONTROLFLOW` and a `Target` edge. The `+` operator indicates the transitive closure of the path. The crossed path `Backwards` ensures that no feedback relation exists between the two tasks.

Conditions on attribute values are stated below the left-hand and right-hand side. First, the `State` attribute of `s` must have the value `Active`. Furthermore, the applicability of the graph transformation is checked against the specific model using the type parameters `D` and `F`. For example, a feedback between `s` and `p` can only be inserted into the task graph if the feedback type `F` is defined such that its `TargetTask` and `SourceTask` 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

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




```
   condition
      (s.State = Active) and
      (p.type in F.TargetTask) and (s.type in F.SourceTask)) and
      (in.type in D.TargetParameter) and (out.type in D.SourceParameter) and
      (out.FormalType <=> in.FormalType) and (m.type in out.FormalType)
end;

path Forward : TASK -> TASK =
   (<-Source- & instance of CONTROLFLOW & -Target-> )
end;

path Backward : TASK -> TASK =
   (<-Source- & instance of FEEDBACK & -Target-> )
end;
```

Figure 13: Graph rewrite rule for inserting a feedback into the task graph

in parameter and the output m is of the right type (out.FormalType).

If all conditions evaluate to true, the left-hand side is replaced by the *right-hand side*. In this example, the graph rewrite rule describes a graph extension, i.e. all elements of the left-hand side are replaced identically, and new elements (indicated in bold face) are inserted into the task graph.

5 Conclusion

We have presented a model for managing task nets which takes various aspects of process dynamics into account (including structural evolution, feedbacks, and concurrent engineering). The model has been defined formally by a graph rewriting system which covers more than 60 pages of PROGRES. We are currently implementing a prototype which is partially generated from the PROGRES specification [20]. We will use this prototype as a test bed for evaluating the DYNAMITE model, but we do not expect the prototype to be usable for actual software production.

We have evaluated the DYNAMITE model by studying scenarios in different application domains (not only software engineering, but also mechanical engineering). So far, these studies have been performed with paper and pencil, since no tools are yet available. Although we have not used actual data from real software processes, we believe that we have investigated realistic examples. These examples demonstrate that concepts such as structural evolution of task nets, explicit feedbacks, task versions (for traceability), or pre-releases of intermediate results (concurrency) are actually needed in practice. We believe that previous work has not addressed these problems in a satisfactory way. On the other hand, empirical evidence of the usefulness of our approach is still missing.

DYNAMITE is a successor to CoMa (Configuration Manager), a version and configuration management system for engineering design documents [21, 23]. The CoMa system primarily supports product management, but it also provides support for process management. CoMa follows a product-centered approach to process management (for the sake of flexibility, DYNAMITE separates between products and processes, see section 3). The CoMa process model is less powerful, but it already includes some essential features of DYNAMITE (in particular structural evolution and concurrent engineering). The first version of the CoMa system was completed in 1993. The experiences gained from the CoMa system have heavily influenced the DYNAMITE model.

The research presented in this paper is embedded into a more comprehensive research activity which aims at developing an integrated environment for

managing products, processes, and resources. The overall conceptual model is described informally in [16]. DYNAMITE formalizes a part of this model. Further work on formalizing the overall model is currently under way.

References

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

[2] N. Belkhatir, J. Estublier, and W. Melo. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. In Finkelstein et al. [6], chapter 8, pages 187–222.

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

[4] W. Deiters and V. Gruhn. Managing Software Process in MELMAC. *ACM Software Engineering Notes*, 19(6):193–205, 1990.

[5] G. Engels and L. Groenewegen. SOCCA: Specifications of Coordinated and Cooperative Activities. In Finkelstein et al. [6], chapter 4, pages 71–102.

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

[7] H. Göttler. *Graph Grammars Used in Software Engineering*, volume 178 of *Informatik Fachberichte.* Springer-Verlag, New York, Berlin, etc., 1988. (in German).

[8] 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.

[9] W. S. Humphrey and M. I. Kellner. Software Process Modeling: Principles of Entity Process Models. In *Proceedings of the* 11^{th} *International Conference on Software Engineering*, pages 331–342, May 1989.

[10] M. L. Jaccheri and R. Conradi. Techniques for Process Model Evolution in EPOS. *IEEE Trans. Software Eng.*, 19(12):1145–1157, December 1993.

[11] G. Junkermann, P. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment. In Finkelstein et al. [6], chapter 5, pages 103–129.

[12] G. Kaiser, P. Feiler, and S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(5):40–49, May 1988.

[13] A. De Lucia, A. Imperatore, M. Napoli, G. Tortora, and M. Tucci. The Tool Development Language TDL for the Software Development Environment WSDW. In 5^{th} *International Conference on Software Engineering and Knowledge Engineering*, pages 213–221, 1993.

[14] C. Montangero and V. Ambriola. OIKOS: Constructing Process-Centered SDEs. In Finkelstein et al. [6], chapter 6, pages 131–151.

[15] M. Nagl, editor. *The IPSEN Project.* Lecture Notes in Computer Science. Springer-Verlag, 1996. to appear.

[16] M. Nagl and B. Westfechtel. A Universal Component for the Administration in Distributed and Integrated Development Environments. Technical Report 94-08, RWTH Aachen, D-52056 Aachen, 1994.

[17] L. Osterweil. Software processes are software, too. In *Proc. of the* 9^{th} *International Conference on Software Engineering*, pages 2–13, Monterey, Carlifornia (USA), 1987. IEEE Computer Society Press.

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

[19] A. Schürr. Rapid Programming with Graph Rewrite Rules. In *USENIX Symposium on Very High Level Languages*, pages 83–100. USENIX Association, 1994.

[20] A. Schürr, A. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer and P. Botella, editors, *Proc. of the* 5^{th} *European Software Engineering Conference (ESEC)*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234, Sitges, Spain, 1995. Springer-Verlag.

[21] B. Westfechtel. Using Programmed Graph Rewriting for the Formal Specification of a Configuration Management System. In G. Tinhofer, editor, *Proceedings WG' 94 Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 164–179. Springer-Verlag, 1994.

[22] B. Westfechtel. A Graph-Based Model for Dynamic Process Nets. In *Proceedings of the* 7^{th} *International Conference on Software Engineering and Knowledge Engineering SEKE '95*, pages 126–130, Skokie, Ilinois, 1995. Knowledge Systems Institute.

[23] B. Westfechtel. Integrated Product and Process Management for Engineering Design Applications. *Integrated Computer-Aided Engineering*, 3(1), 1996.