International Journal of Software Engineering and Knowledge Engineering
Vol. 7, No. x (1997) xxx–xxx

GRAPH-BASED SOFTWARE PROCESS MANAGEMENT

PETER HEIMANN, CARL-ARNDT KRAPP, BERNHARD WESTFECHTEL
Lehrstuhl für Informatik III, RWTH Aachen
D-52056 Aachen, Germany

and

GREGOR JOERIS
University of Bremen, TZI/AG-KI, P.O. Box 330 440
D-28334 Bremen, Germany

Received (received date)
Revised (revised date)
Accepted (accepted date)

Software process dynamics challenge the capabilities of process-centered software engineering environments. Dynamic task nets represent evolving software processes by hierarchically organized nets of tasks which are connected by control flow, data flow, and feedback relations. Project managers operate on dynamic task nets in order to assess the current status of a project, trace its history, perform impact analysis, handle feedbacks, adapt the project plan to changed product structures, etc. Developers are supported through task agendas and provision of tools and documents. Chained tasks may be enacted in parallel (simultaneous engineering), and cooperation is controlled through releases of document versions.

Dynamic task nets are formally specified by a programmed graph rewriting system. Operations on task nets are specified declaratively by graph rewrite rules at a high level of abstraction. Furthermore, editing, analysis, and enactment steps, which may be interleaved seamlessly, are described in a uniform formalism.

Keywords: software process modeling, programmed graph rewriting, dynamic task nets

1. Introduction

Software processes are highly dynamic; only rarely can they be planned completely in advance [11]. Many changes have to be reacted to while a software process is being executed. For example, the product structure may evolve, feedbacks may occur to earlier steps in the lifecycle, the requirements may change, etc. These changes challenge the capabilities of a process-centered software engineering environment: Its model of the real-world process must be modified on the fly.

The DYNAMITE approach [14] specifically addresses software process dynamics. A software process is represented by *dynamic task nets* whose structure evolves during process enactment. A type-level net constitutes an ER-like schema. Instance-level nets are constructed by instantiating task and relation types. DYNAMITE

aggregates well-known concepts (taken e.g. from PERT charts and data flow diagrams) into a coherent framework which takes the specific requirements of software process management into account. For example, feedbacks are represented by a dedicated type of relation, task versions are introduced to keep feedback handling traceable, and simultaneous engineering is supported by pre-releases of intermediate results.

In particular, dynamic task nets are designed to support *project managers* who are provided with high-level, graphical views on ongoing software processes. They provide a detailed account of the current status of an ongoing project, a trace of the project history, impact analysis (e.g., for assessing the consequences of feedbacks), adjustment of plans in response to changes in the product structure, and facilities for controlling execution (e.g., suspension of certain tasks affected by a feedback).

Furthermore, *developers* are supplied with an agenda of tasks, and for each task a work context is maintained which comprises input documents, output documents, available tools, etc. The work context may be changed while a task is being executed (e.g., new versions of inputs may arrive in response to a feedback). Chained tasks — tasks connected by control flow relations — may be executed in parallel (simultaneous engineering).

Dynamic task nets are complex data structures on which sophisticated operations are performed. Structure and behavior are specified by a *programmed graph rewriting system*. Task nets are formally represented as attributed graphs. Graph rewrite rules describe transformations of these graphs at a high level of abstraction; analogously, queries may be encoded as graph tests. Graph tests and graph rewrite rules constitute the basic actions from which programs may be composed.

Using programmed graph rewriting for the specification of dynamic task nets provides the following advantages:

- Editing, analysis, and execution of task nets are described in a uniform formal framework. This is essential because these different kinds of activities may have to be interleaved seamlessly. For example, environments based on Petri nets (e.g., Process Weaver [10], MELMAC [5], or SPADE [1]) do not meet this requirement. While Petri nets provide a formal foundation for analysis and execution, editing has to be described outside the Petri net formalism.

- Since the specification is operational, executable code may be generated from the specification. In this way, a rapid prototype of a process-centered software engineering environment may be constructed [16]. To this end, we are using the PROGRES development environment [34].

Our work is located at the intersection of software engineering and knowledge engineering. It addresses an important discipline of software engineering, namely software process modeling. Furthermore, graph rewrite rules constitute a knowledge base of the software process. The shape of these rules heavily depends on the degree to which the software process is structured and understood. In case of

spontaneous, ad hoc processes, the rules allow for much freedom, but require many human decisions. In case of routine processes, the rules constrain the process more tightly, but reduce the need for human intervention.

This paper mainly focuses on the formal specification underlying DYNAMITE (Sec. 3). Before, dynamic task nets are introduced informally (Sec. 2). Section 4 compares related work, and a conclusion is given in Sec. 5.

2. Informal Description

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, development of a software system may follow different life cycle models (e.g., waterfall, spiral, or prototyping model). We distinguish between atomic and complex realizations. In case of an *atomic realization*, a task is not refined into subtasks; a *complex realization* consists of a net of subtasks.

Control flow relations impose an order on the tasks to be enacted; they resemble precedence relations in PERT charts. Enactment of chained tasks may overlap (simultaneous engineering). For example, implementation may start before the design is finished.

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.

Handling of a feedback depends on the state of the target task t. If t is still active, it may process the feedback, e.g. by producing a new version of an output document. If t has already terminated, it has to be reactivated. To keep the process traceable, a new *task version* t' is created. Enactment of t' may imply that new versions of dependent tasks have to be created as well.

Data flow relations are used to transmit data between tasks connected by hierarchical, control flow, or feedback relations. Data may be passed downwards (supertask input $\rightarrow$ subtask input), upwards (subtask output $\rightarrow$ supertask output), and horizontally (task output $\rightarrow$ task input).

During enactment, an active task may produce multiple *versions* of its outputs and consume multiple versions of its inputs. For example, errors may be fixed in a module body, and the module body may be adapted to a changed interface. The net keeps track of the sequence of produced and consumed versions (traceability). Versions provide the basis for properly controlled workspace management (note the link to software configuration management).

With respect to task nets, we have to distinguish between the instance level and the type level. At the *instance level*, a task net serves as a model of an actual soft-

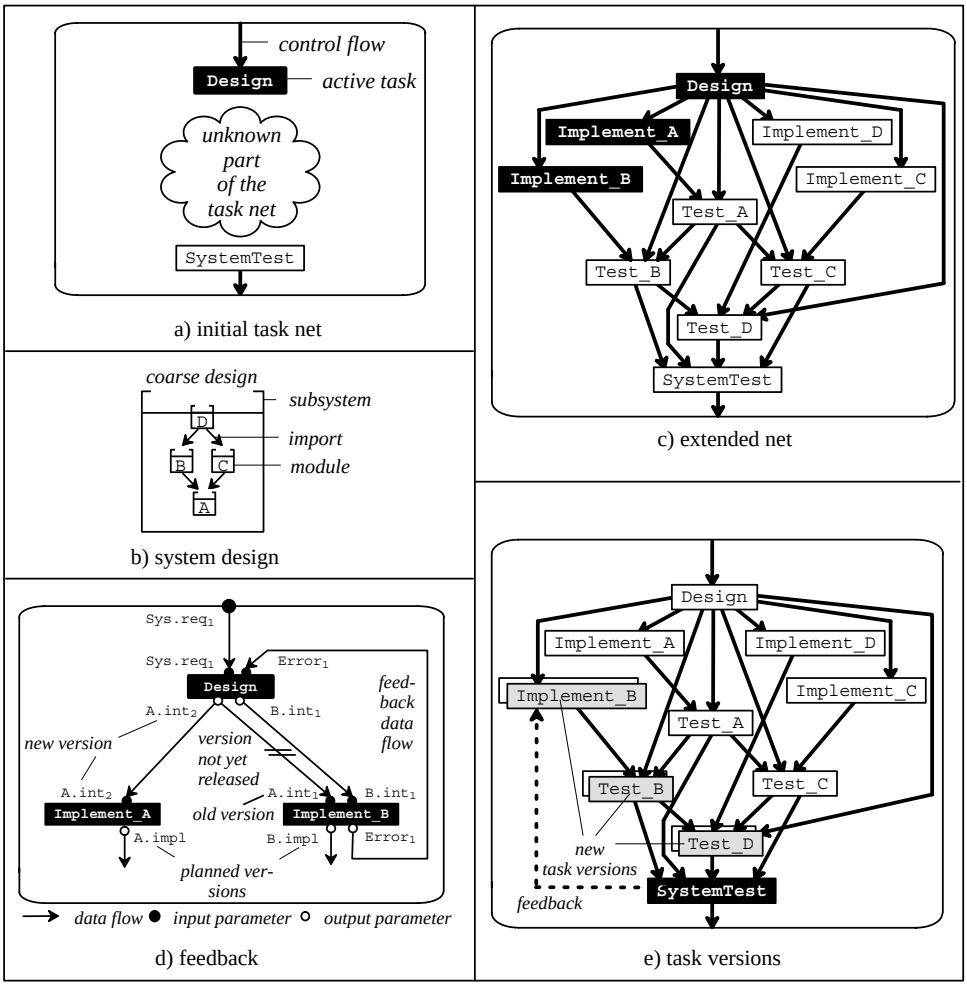

Fig. 1. Sample process.

ware process (process performance). At the *type level*, a task net defines the types of tasks and relations which are instantiated during enactment (process definition).

To illustrate the concepts introduced above, let us consider the development of a simple software system. Initially, it is merely known that the development starts with the design and ends with the system test. Thus, the task net of Fig. 1a is still incomplete. Note that the figure only shows control flow relations, which are used here to determine the start node and the end node of the task net.

The missing tasks may be filled in only after the coarse design has been determined (product dependent task net). In the following, we assume a simple software system consisting of four modules `A`,...,`D` (Fig. 1b). The resulting task net is displayed in Fig. 1c. All implementation tasks may be carried out in parallel. Module tests are arranged in bottom-up order.

In order to accelerate development and to detect design errors as early as possible, implementation tasks may be started before the design is completed (simultaneous engineering). In Fig. 1c, the implementation tasks for `A` and `B` are active because their interfaces have already been released by the design task. On the other hand, `C` and `D` may not be implemented yet.

Later on, the programmer of module `B` detects an error in the interface imported from `A` (e.g., a missing procedure). Thus, a feedback to the design is inserted into the task net. Along the corresponding data flow, an error report is passed to the design task. Figure 1d shows the design task and the implementation tasks for `A` and `B`, as well as their inputs, outputs, and connecting data flows. The design task has already produced the second version of the interface for `A` (denoted as $\texttt{A.int}_2$). So far, this version has only been released to the implementation task for `A`. After the required change has proven implementable, the release may also be granted to `Implement_B`.

If the target task of a feedback flow is no longer active, the error situation is more severe. Figure 1e shows the detection of an implementation error during the system test. In this case, we want to give the project manager the chance to later trace what has happened. Sometimes, the person who executed the implementation task for the first time is no longer available, and someone else has to be assigned. We therefore create a new version of the task instead of reactivating the old one. The new task version gets copies of the inputs of the original task version, plus the new error information. All subsequent tasks are potentially affected by the new task version. If they have been finished before a new input version arrives, new versions of these tasks are created as well.

The evolution of task nets is governed by an application specific schema. Figure 2 shows a schema for the system development task nets described above. According to this schema, a correct net must contain exactly one `Design` and one `SystemTest` task instance and at least one `Implement` and one `Test` task each. The schema also defines names and document types for required and optional inputs and output ports of tasks, and it regulates the structure of control, feedback and dataflow relations. While a task of type `Implement`, for example, has to deliver a module `Body` to a `Test` task, all feedbacks and their accompanying data flows are optional.

The task net of Fig. 2 introduces domain-specific structural information. In addition, the behavior of task (and relation) types needs to be defined according to the needs of a specific application domain. This includes the definition of states, state transitions, conditions, events and triggers (see Sec. 3).

3. Graph-Based Specification

3.1. *Motivation and overview*

A dynamic task net consists of many entities which are mutually interrelated. Based on our experience in building structure-oriented environments, the data

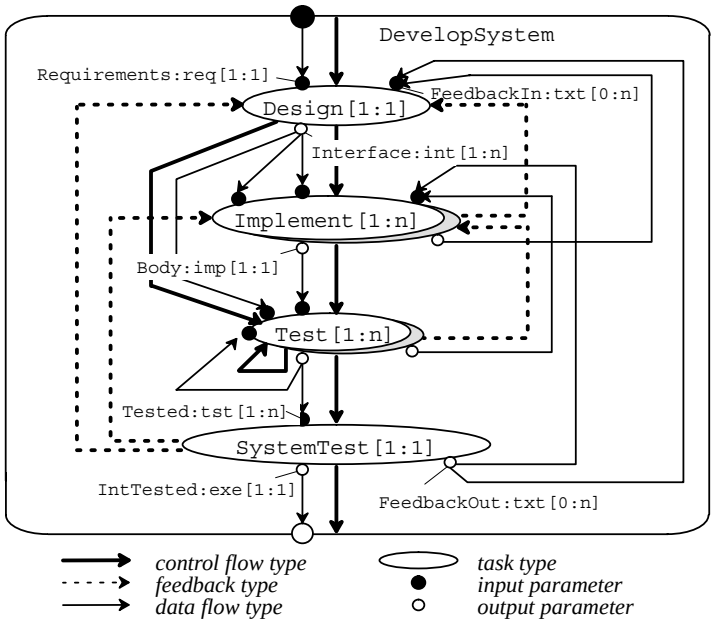

Fig. 2. Task schema.

model of attributed graphs has proven suitable for the internal representation of complex data structures [27]. An attributed graph consists of attributed nodes which are interconnected by labeled, directed edges. Figure 3 shows the internal graph structure representing a dynamic task net as shown in Fig. 1d. Tasks, parameters, tokens, and documents are modeled as graph nodes which are interconnected by various edges. N-ary relations, attributed relations, and relations participating themselves in relations are modeled by nodes and adjacent edges (e.g., control flows, feedback relations, data flows). The structure of task graphs will be explained in Sec. 3.2., thus the reader is encouraged to return to Fig. 3 later.

During editing, analyzing, and enactment of a task net, complex transformations and queries have to be performed on the internal data structure. We have chosen programmed graph rewriting in order to specify theses complex operations on a high level of abstraction. In particular, we have chosen the specification language PROGRES which is based on programmed graph rewriting [33]. It combines concepts from database systems, knowledge-based systems, graph rewriting, and imperative programming languages into a coherent language: A graph schema defines the graph elements and the graph structure declaratively. It defines types of nodes, edges, and attributes. Derived attributes and relations can be defined in a schema as well. Graph transformations are specified by high-level graph rewrite rules which essentially replace a graph pattern within the current graph by another one. Rewrite rules can be combined to form more complex transformations. Thus, the operational programming style is supported. Graph rewrite rules are inherently

non-deterministic since usually more than one match of a graph pattern can be found when applying a graph rewrite rule. By means of backtracking non-determinism is taken into account, thus supporting the rule-based programming style.

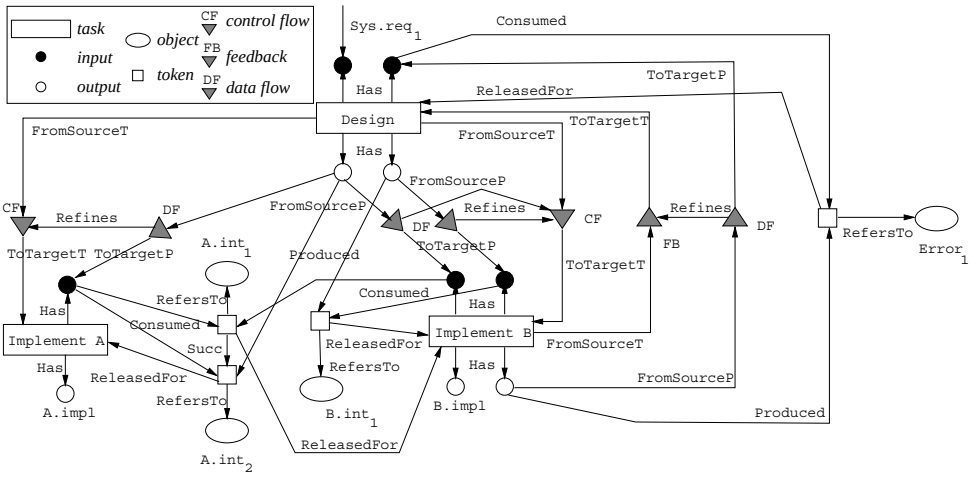

Fig. 3. Example of a task graph (cut-out).

Before delving into the details, let us survey the steps in which the PROGRES specification of dynamic task nets is going to be presented:

- The *base model* (Sec. 3.2) defines the constituents of task nets such as tasks, inputs, outputs, control and data flows, etc. Furthermore, it provides primitive operations which maintain structural integrity (e.g., when a control flow is created, it is checked that no cycle is introduced). The base model is domain-independent and can be adapted with respect to both structure and behavior.
- *Structural parameterization* (Sec. 3.3) addresses the definition of domain-specific types and structural constraints, such as illustrated e.g. in Fig. 2. The primitive operations of the base model are adapted such that domain-specific structural constraints are taken into account.
- *Behavioral parameterization* (Sec. 3.4) defines the behavior of dynamic task nets by cooperating state machines. To this end, a state transition diagram is introduced, conditions for executing transitions are defined, for each primitive operation legal states of its application are determined, and event handlers are used for defining reactions on state transitions and primitive operations.

3.2. *Base model*

A specification written in PROGRES consists essentially of two parts: The static structure of a graph is defined declaratively by means of a graph schema, while

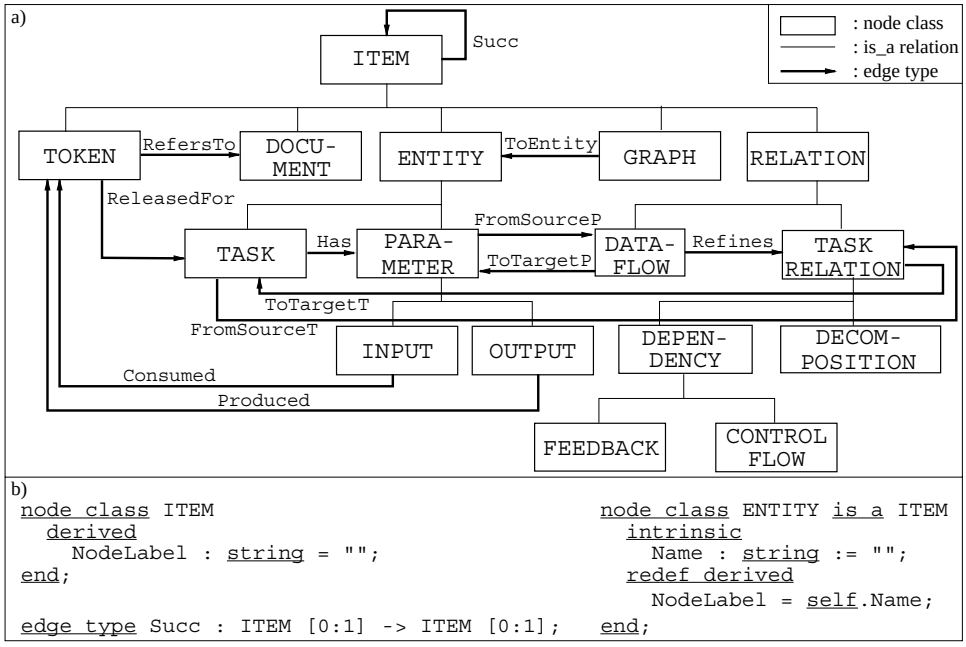

Fig. 4. Graph schema.

operations on a graph are defined by graph rewrite rules. Figure 4a illustrates the *graph schema* of the base model for dynamic task nets. It presents the graphical representation and hides the attribute declarations. Figure 4b shows the textual declarations for the node class `ITEM`, node class `ENTITY`, and edge type `Succ`. Node class `ITEM` serves as the root class. At this point of the class hierarchy, the concept of successor versions is introduced regardless of tasks, parameters, relations, or alike. New versions can be derived from all elements found in a task net. On the next level of the hierarchy, we mainly distinguish between entities and relations. The node classes `TOKEN`, `DOCUMENT`, and `GRAPH` describe nodes representing tokens, nodes representing software artifacts, and nodes aggregating all entities of a task net by means of `ToEntity` edges, respectively. `TASK` nodes have `PARAMETER` nodes which are either `INPUT` or `OUTPUT` parameters. Relations between entities are modeled as edge-node-edge constructs because edge types cannot carry attributes in PROGRES. Data flow relations are established between parameters and refine task relations connecting two tasks. Besides the hierarchical decomposition relation, we further distinguish feedback and control flow dependencies. The graph schema already fixes structural constraints on dynamic task nets. Data flow relations, for instance, can only be established between parameters and not between tasks. A task graph conforming to the schema has already been presented in Fig. 3.

The PROGRES type system distinguishes between node classes, node types, and node instances. Node types are instances of node classes and cannot be redefined. Node instances can only be established as instances of node types. Due to

the stratified type system, types are first order objects which may be supplied as parameters, and may be stored as values of node attributes. While the presented graph schema and the following graph productions of the base model are independent of any application domain of dynamic task nets (level of node classes), specific details are introduced with node types and refining graph productions (see Sec. 3.3 and Sec. 3.4).

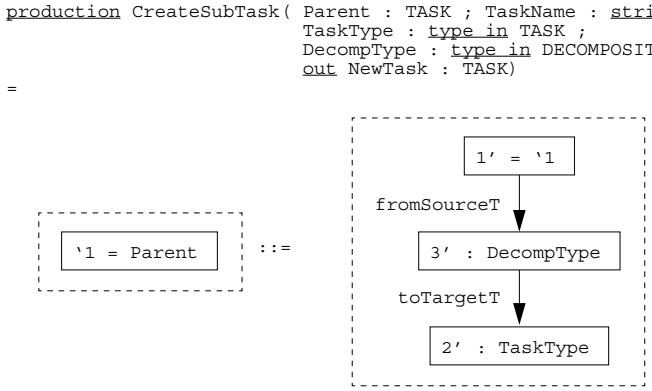

Fig. 5. Graph production for subtask creation.

We are now switching to the operational part of the specification, which describes how an attributed graph can be transformed and queried. A simple *graph production* is shown in Fig. 5. It describes the operation for introducing a new subtask w.r.t. a parent task into the net. The rule is applied with a node representing the parent task, a character string for the name and a node type for the new node, and a type for the new decomposition relation. While the left-hand side of the rule is already fixed by the input parameter `Parent`, the condition part must be still evaluated. We skip the detailed presentation of the condition part and refer to a later discussion in this paper. In case of successful evaluation, the `Parent` node is replaced identically and two new nodes and edges are introduced into the graph. The types of the new nodes are fixed by the parameters `TaskType` and `DecompType`. Furthermore, the intrinsic `Name` attribute of node `2'` is set to the value of the `TaskName` parameter, and the node `2'` is returned as output parameter. Another rule not shown in this paper describes deletion of a task. Together they are essential for dynamic instantiation and modification of a task net.

The next rule we present illustrates the token game of dynamic task nets. In contrast to the former graph rewrite rule, a more complex graph pattern has to be matched for the `Consume` operation (cf. Fig. 6). The operation is performed when a task consumes a token. As one can easily see, five nodes have to be found of which only node `'1` is fixed by an input parameter. The nodes have to be interconnected

by an edge and some path expressions (indicated by solid and double arrows), and some nodes must not be interconnected by edges and paths (indicated by crossed solid and double arrows). The restrictions at nodes `‘3` and `‘4` ensure that the operation can only be applied to the latest version of the task and the output parameter respectively. The path declaration for the `SrcPmtr` path is given in Fig. 6 as well. It connects an `INPUT` node with an `OUTPUT` by a path consisting of two edges (`ToTargetP`, `FromSourceP`) which are traversed in reverse direction.

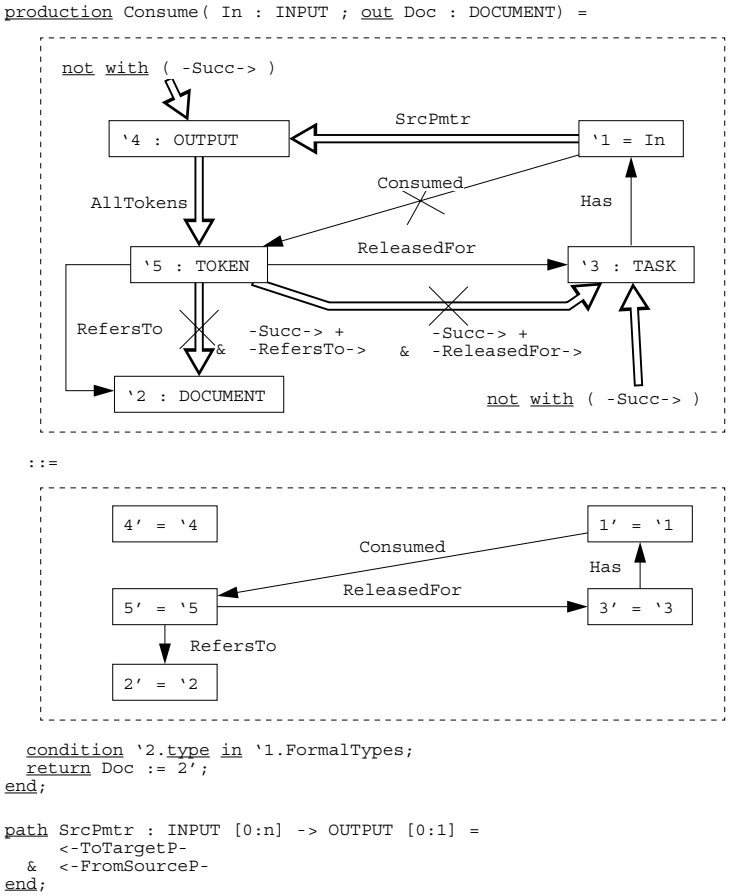

Fig. 6. Graph production for token consumption.

Thus the rule for consuming a token can be read as follows: Starting from the node representing the input parameter, a corresponding output parameter of the predecessor task is searched via the `SrcPmtr` path. Among all tokens produced by this parameter, a token has to be found for which the following structural constraints hold:

- The token has not yet been consumed by the input parameter (indicated by the crossed edge `Consumed`).

- The token has been released for the task having the parameter as input (indicated by the `ReleasedFor` edge).
- No other token following in the transitive `Succ` relation has been released for this task (indicated by the crossed path).
- The token refers to a `DOCUMENT` node by a `RefersTo` edge.
- No other token following in the transitive `Succ` relation refers to this `DOCUMENT`.

The condition part ensures that the document to be produced is type compatible with the formal type of the input parameter. After the successful pattern match and condition evaluation, the `In` node is associated with the `TOKEN` node `'5` by a `Consumed` edge.

The `Consume` rule demonstrates the customized data flow semantics of dynamic task nets. `Consume` does not destroy tokens; rather, they are preserved for traceability. Furthermore, multiple dependent tasks do not compete for tokens. Finally, propagation of tokens can be controlled through selective releases (see again Fig. 3).

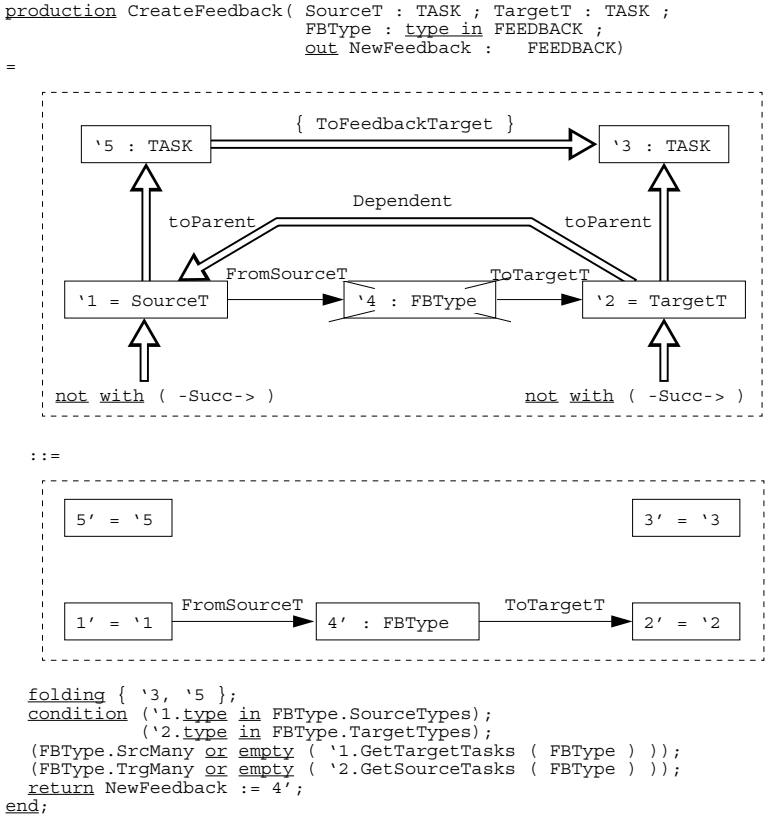

Fig. 7. Graph production for creating a feedback.

Figure 7 shows a graph production for introducing a feedback relation into the net. A feedback can only be created between tasks which are transitively related by a control flow (path `Dependent`) and which have no successor versions. Furthermore, the negative node `‘4` ensures that a feedback does not exist yet. In the condition part, structural constraints are checked whose detailed explanation is given later in Sec. 3.3. When introducing relations into the task net, two balancing rules have to be obeyed concerning the hierarchical structure of task nets:

1. **Feedback within a subnet:** The introduction of a feedback 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 graph. The `folding` clause allows different nodes of the left-hand side (here nodes `‘3` and `‘5`) to be mapped onto the same node in the current graph. The path expression `ToFeedbackTarget` collapses in this case (indicated by curly brackets).

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

The balancing rules support the concept of abstraction. While the task interface essentially hides the realization, it cannot be avoided that subtasks have to be connected with elements outside the subnet. But because of the balancing rules, dependencies between subnets can be still seen at the interface level. The details become clear when zooming into the subnet.

```
transaction DeriveTask( OldTask : TASK ; out NewTask : TASK)
=
  use task : TASK;
      NewPmtr : PARAMETER
  do
      DeriveTaskItem ( OldTask, out task )
    & loop
        DeriveSourceDependency ( task )
      end
    & loop
        DeriveTargetDependency ( task )
      end
    & for all pmtr : PARAMETER := OldTask.-Has->
      do
          DeriveParameter ( pmtr, out NewPmtr )
        & choose
            DeriveSourceDataflow ( NewPmtr )
          else
            DeriveTargetDataflows ( NewPmtr )
          end
      end
    & NewTask := task
  end
end;
```

Fig. 8. Graph transaction for deriving a task.

Creation of a new task version is presented as the last example of the base model (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* where their application is determined by control flow constructs. The graph transaction for establishing a new new task version performs the following steps:

1. Create a new `TASK` node and establish a `Succ` edge between old and derived task by applying the `DeriveTaskItem` production.

2. As long as there is a control flow left, copy the incoming control flow relation and establish a `Succ` edge between the old and the new relation node.

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

4. For all parameters of the old task, create a parameter for the new task, copy the old incoming and outgoing data flow relations, and establish `Succ` edges between old and new parameters and old and new data flows.

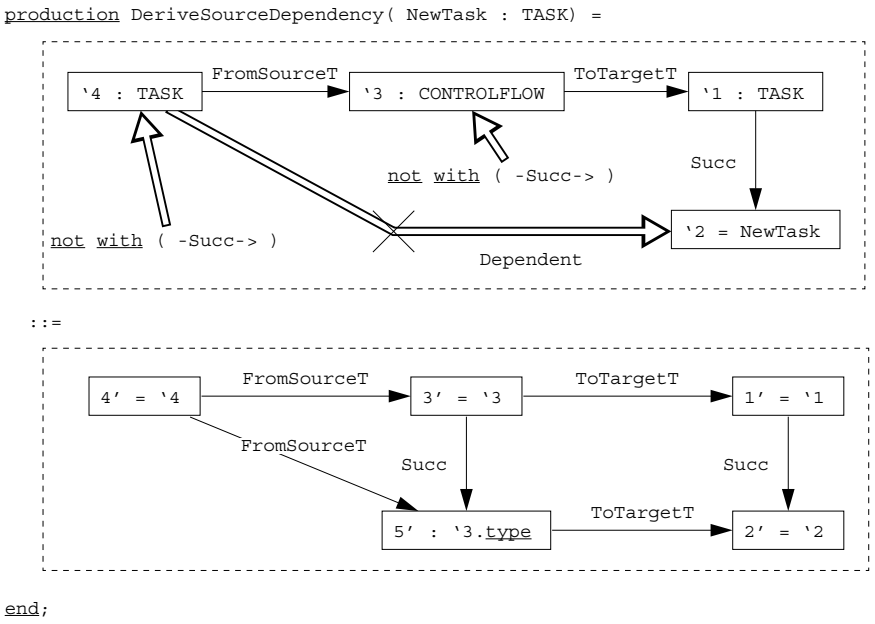

Fig. 9. Graph production for deriving control flow relations.

Due to space limitations, we present only one of the involved graph productions. The rule shown in Fig. 9 describes how incoming control flow relations are reconstructed for a new task version. From the corresponding old task, an incoming control flow relation is searched which has no outgoing `Succ` edge. If the source task of this control flow relation is not already connected with the new task version (indicated by a crossed path `Dependent`), a new control flow relation is established and is connected with the old relation by a `Succ` edge. The rule illustrates the

uniform use of the `Succ` edge. By analyzing all introduced `Succ` edges, important process steps can be reconstructed and traced.

3.3. *Structural parameterization*

The base model merely defines general notions such as task, input and output, control and data flow, etc. When a specific application domain is considered, the base model has to be augmented with domain-specific structural knowledge. In Sec. 2, we have sketched how this may be done with the help of a task schema (cf. Fig. 2). In the following, we will sketch briefly how a task schema may be encoded in PROGRES.

The stratified type system supports a clear separation between the base model and a specific model. While the base model is defined in terms of node classes, *node types* are introduced in order to represent the concepts of a specific application domain. For example, a task `Implement_A` is represented as an instance of the node type `Implement` which is in turn instantiated from the node class `TASK` of the base model.

Graph transformations have to be adapted such that domain-specific constraints are taken into account. In principle, the process modeler could write customized graph rewrite rules, taking the rules of the base model as a starting point. However, routine adaptations may even be performed without writing any code by extending the graph schema. By "routine adaptations", we refer to the mapping of an ER-like diagram such as the task schema of Fig. 2.

```
node class TASK_RELATION is a RELATION
  meta
    SourceTypes : type in TASK [0:n];
    TargetTypes : type in TASK [0:n];
    SrcOptional : boolean;
    TrgOptional : boolean;
    SrcMany : boolean;
    TrgMany : boolean;
end;
(* FEEDBACK is a TASK_RELATION *)
```

```
node type ImplementToDesign : FEEDBACK
  redef meta
    SourceTypes := Implement ;
    TargetTypes := Design ;
    SrcOptional := true ;
    TrgOptional := true ;
    SrcMany := false ;
    TrgMany := true ;
end;
```

Fig. 10. Structural parameterization through meta attributes.

The proceeding to be followed is illustrated by a simple example, namely the mapping of a feedback type from implementation to design. The left-hand side of Fig. 10 shows the declaration of the node class `TASK_RELATION` in which attributes are introduced to express domain, range, and cardinality of a certain relation type. The values of these attributes are type- rather than instance-specific; they are called *meta attributes* in PROGRES. The right-hand side of Fig. 10 shows a corresponding node type declaration where appropriate values are filled into the slots introduced in the node class declaration.

Our example reads as follows: A feedback of type `ImplementToDesign` goes from a task of type `Implement` to a task of type `Design`, it is optional at both ends, only one `Design` task is reachable from a given `Implement` task, but multiple feedbacks may arrive at a given `Design` task.

The graph rewrite rule for creating a feedback relation has already been discussed in Sec. 3.2 (Fig. 7). Its condition part is used to check the values of meta attributes and therefore provides for the required domain-specific adaptation. The first and the second condition check the types of the source and the target task, respectively. The third and the fourth condition ensure that no cardinality overflow will occur (either many incoming or outgoing relations are allowed, or the current set of relations is empty).

3.4. *Behavioral parameterization*

Execution semantics of dynamic task nets (the behavior) is defined by means of *communicating state machines*. Instead of giving a process modeler the freedom to define for each task type an individual state machine, we fix all possible states and transitions for a task and leave only transition conditions open for user-defined adaptation. The reason for fixing a set of enactment rules is that behavioral parameterization can be done much easier if a core of enactment semantics is already provided by the process model. In this way, we believe that even users not familiar with complex programming languages can perform this parameterization step. Note that the base model of Sec. 3.2 still leaves the definition of behavior completely open. Thus, if the standard behavior proposed below does not meet domain-specific requirements, it can be modified or replaced without affecting the base model.

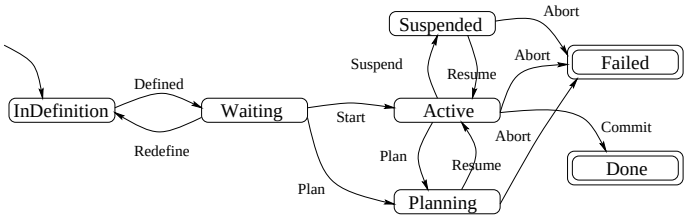

Fig. 11. Common state transition diagram.

Figure 11 presents the state transition diagram which is common to all tasks in a dynamic task net. Each task has an initial state `InDefinition` and two final states `Done` and `Failed` which indicate the successful and faulty completion of a task. After definition time, when all required parameters and incoming and outgoing dependencies have been defined, a task waits for its activation (i.e. `Start` transition) or waits for further refinement activities (i.e. `Plan` transition). Furthermore, a task may be suspended and subsequently resumed any time.

On the base of this state machine a core set of semantics for task net enactment is fixed. Rules and conditions define the interplay of state machines of neighboring tasks. In particular, they force and restrict transitions of tasks. The following three rules are examples of execution rules for tasks regardless of whether they are complex or atomic, or whether they are performed automatically or manually:

1. A task can only be committed if all predecessors have been successfully completed.

2. Suspension and abortion of a task is transitively done for all subtasks.
3. A task can only be activated if at least one of its predecessors has left the initial state.

The enactment state of a task represents the cumulative results of its behavior. It describes which operations of the base model can be applied to the task. To this end, every operation of the base model is enriched with state information about tasks involved in that operation. For instance, operations for consuming and producing a token cannot be applied to a task which is in state `InDefinition`. By combining the operations of the base model with state information on involved tasks, intertwined editing, analyzing, and enactment of dynamic task nets can be supported.

In order to handle dynamic, non-anticipated changes in the model, every state transition and every operation is followed by an event. Events are sent to tasks which may be affected by the state transition or by applying the operation. Appropriate reactions are either done manually by corresponding task actors who receive notifications or may be automated by applying pre-defined trigger definitions.

```
transaction Commit( Task : TASK) =
    (Task.State = Active)
  & for all task := Task.=Master=> ::
      task.State = Done
    end
  & Task.State := Done
  & SendCommit ( Task )
end;
```

```
transaction Abort( Task : TASK) =
    [...]
  & for all task := (Task.=CurrentChildren=>)
    do
      choose
        when (task.State in
              (Active or Planning or Suspended))
        then
          Abort ( task )
      end
    end
  & Task.State := Failed
  & SendAbort ( Task )
end;
```

Fig. 12. Sample state transitions.

After the brief informal presentation we can now proceed with the formal specification of task net enactment. Nodes representing tasks carry an intrinsic attribute `State` which describes the current enactment state of the task. A transition is defined by a transaction which checks the model inherent transition conditions, assigns the new state, eventually forces other operation applications on child tasks, and sends an event to all nodes in the context. Sample transactions are presented for the `Commit` and `Abort` transition (cf. Fig. 12). While the former transaction checks enactment rule 1 from above, the latter implements rule 2. Events are sent by calling an event-handler which is a graph transaction as well. The consequences of the event handling, however, do not affect the state transition (see below).

```
transaction ConsumeOp( Input : INPUT ; out Doc : DOCUMENT) =
    (Input.<-Has-.State in (Active or Planning))
  & Consume ( Input, out Doc )
  & SendConsume ( Input.<-Has- )
end;
```

Fig. 13. Consume operation with state information.

An operation of the base model is combined with state information by means

of a transaction which calls the operation after evaluating state conditions. When applying the `Consume` operation, for instance, the task must be in state `Active` or `Planning` (cf. Fig. 13). Furthermore, a `Consume` event is sent to tasks in the context.

```
transaction SystemTestStart( Task : SystemTest) =
  use Doc : DOCUMENT
  do
      for all task := (Task.=Master=>)
      do
        (task.State = Done)
      end
    & Start ( Task )
    & for all inP := Task.=HasInput=>
      do
        Consume ( inP, out Doc )
      end
  end
end;
```

Fig. 14. Refined `Start` transition.

So far, we have introduced domain-independent standard behavior. By means of the `SystemTest` task, we present the refining mechanism for state transitions. The same mechanism applies for operations. Figure 14 shows the `Start` transition for a `SystemTest` task. It is important to note that the original transition operation (`Start`) is still invoked. This ensures conformance to the inherent enactment rules. Additionally, some other conditions are checked and a operation is performed automatically with this transition. In particular, all predecessor tasks have to be successfully completed and for all input parameters the token is consumed.

```
transaction SendStart( Task : TASK) =
    [...]
  & for all task := Task.=Dependent=>
    do
      choose
        [...]
      else
        when (task.type = SystemTest)
        then
          Abort ( task )
      else
        skip
      end
    end
end;
```

Fig. 15. `Start` event handler.

As the last topic concerning the semantics for net enactment, we present the event-trigger mechanism. Assume that a process modeler wants to abort the task for the system-test each time a predecessor task starts again (due to a feedback, for instance). To this end, he defines a trigger for a `SystemTest` task type in case of receiving a `Start` event from a predecessor (cf. Fig. 15). For the task `Task` originating the `Start` event, all successor tasks are considered. In case an active `SystemTest` is found, the abortion is performed. Note that the application of this PROGRES transaction is always successful (through the `skip` statement as the last `choose` alternative). This ensures that the transaction calling this event-handler is not affected by the consequences of the event (decoupled transaction).

While behavioral parameterization was presented on the level of PROGRES, we are currently developing a user-friendly language for parameterization which can be transformed to PROGRES.

4. Related Work

Although software process modeling is still a young discipline, researchers have already transferred, adapted, and extended virtually any *paradigm* deemed appropriate (rules [17], blackboards [26], imperative programs [29], Petri nets [5], events [9], objects [8], state machines [20], attribute grammars [21], etc.; see [11] for a representative sample). In the following, we discuss a subset of these paradigms which we consider most relevant and influential. During this discussion, the reader should keep in mind that a specific approach may combine multiple paradigms. The section is concluded with some remarks on the theory and practice of graph rewriting.

4.1. *Net-based approaches*

Systems such as PROCESS WEAVER [10], MELMAC [5], and SPADE [1] are based on different variants of *Petri nets*. A software process is modeled as a hierarchical collection of Petri nets. To prepare enaction, a template is copied and populated with tokens. Enactment is modeled by the well-known token game.

Petri nets have been criticized for their inflexibility. Therefore, several mechanisms have been devised to support modification of Petri nets during enactment. Due to late binding (PROCESS WEAVER and SPADE), the definition of a subnet must be available only when it is called during enactment. MELMAC offers special transitions called modification points. When a modification point is fired, enactment of a specific subnet is suspended, the subnet is modified, and enactment is resumed. SPADE models the definition, enactment, and modification of process models as a higher order Petri net (reflexion). In any case, modification of an enacted process model is considered a serious disruption, while DYNAMITE supports seamless interleaving of editing, analysis, and enactment.

Another difference concerns the way instantiation is handled. In MELMAC, SPADE, and PROCESS WEAVER, enacted nets are *populated copies* of net templates. Thus, the nets at the definition level and the enactment level have the same shape. In contrast, DYNAMITE distinguishes between *type-level nets* and *instance-level nets* (Fig. 2 and 1, respectively). Populated copies resemble type-level rather than instance-level nets (e.g., only one transition per task type). There is no equivalent to instance-level nets which provide the basis for tracing, analysis, enactment, and planning of software processes.

With respect to its ER-like instantiation mechanism, DYNAMITE resembles some approaches developed in the field of distributed systems (e.g., SDL [32], Estelle [4], and Darwin [25]). However, these approaches do not address the specific problems occurring in software process management (e.g., feedbacks, simultaneous

engineering, human intervention, etc.).

Teamware [38] and its successor Endeavors [3] heavily rely on late binding and interpreters to support flexibility in software process management. However, they essentially stick to the populated copies approach and therefore suffer from the problems outlined above. A small step towards instance-level nets is taken by the following feature: A set node acts as a placeholder which is replaced at runtime with a set of parallel tasks with identicall predecessors and successors (e.g., parallel implementation tasks following a design task). DYNAMITE supports net modifications during enactment in a much more general way.

4.2. *Rule-based approaches*

In rule-based approaches, a software process is modeled as a collection of activities, and rules constrain their execution order. For example, in Marvel [17] each rule consists of a precondition, an activity, and a set of alternative postconditions, depending on the outcome of the activity. When the user attempts to execute an activity whose precondition is not fulfilled, Marvel's process engine constructs a *plan* through backward chaining and tries to execute the plan such that the requested activity can eventually be performed. Forward chaining is also supported, i.e., rules can be fired automatically when their preconditions hold. In particular, Marvel has been applied successfully to automatic tool chaining, e.g., compiling and linking of programs. As noted in [2], forward and backward chaining are less useful in human-intensive processes, where the execution of activities needs to be controlled by the user.

The plans maintained by Marvel differ from dynamic task nets in two ways. First, plans are internal data structures belonging to the runtime stack of the process engine. They are presented to the user only for explanations and cannot be manipulated manually. Second, plans are made for a short term to schedule tool invocations such that an issued command can eventually be executed. In contrast, dynamic task nets are plans for rather long-lasting activities (e.g., several months for developing a non-trivial subsystem).

With respect to these issues, EPOS [19, 24] is much closer to DYNAMITE. In EPOS, a software process is modeled as a hierarchy of task nets. Execution and planning are interleaved: When the interpreter reaches a placeholder for a composite task, the planner is called in order to refine it. The planner can construct a plan from the product structure and incrementally adjust the plan after the product structure has been changed. A plan is considered a proposal which can still be rejected and modified by the user. Furthermore, task nets are more long-term than in Marvel.

DYNAMITE offers several improvements over EPOS. First, DYNAMITE provides a richer set of modeling constructs. In particular, EPOS does not distinguish between control and data flows, feedbacks are not represented explicitly through relations (feedbacks are handled by asserting error conditions which trigger re-execution of tasks), and task versions are not supported at all. Second, DYNA-

MITE offers more sophisticated semantics of enactment and specifically supports simultaneous engineering. Finally, traceability is not addressed in EPOS; changes to task nets are performed by overriding.

Only recently, CoMo-Kit [6, 7] has been proposed for planning and enacting software processes. Since CoMo-Kit strongly resembles EPOS in many respects (hierarchical task nets, input/output dependencies, incremental replanning, interleaving of planning and enactment), the arguments given above hold for CoMo-Kit as well.

4.3. *State-based approaches*

In state-based approaches, a software process is modeled as a collection of cooperating state machines. For example, entity process models [18] attach to each entity (document) produced in the software process a state machine which is coupled to the state machines of its neighbors by sending and receiving events. Entity process models are formalized by *statecharts*, based on the semantics implemented in STATEMATE [12].

STATEMATE assumes a static hierarchy of processes, i.e., the cooperating state machines are known in advance. ESCAPE [20] combines the EER data model with state charts and can handle dynamic instantiation of processes. In particular, the state chart specifications may abstract from the instances which are actually present at runtime (e.g., through universal quantifiers in conditions on state transitions). ESCAPE is compiled down into a rule-based language which serves as the foundation of the process-centered software engineering environment MERLIN [30].

In DYNAMITE, we content ourselves with using simple transition diagrams; so far, we did not need the increased modeling power offered by statecharts. Furthermore, we strive for using a uniform state transition diagram for all types of tasks (with customizable conditions and event handlers), avoiding the complexities involved in the definition of the cooperation between heterogeneous state machines. Since the definition of communicating state machines is rather awkward at the PROGRES level, we are developing a process modeling language to be compiled into PROGRES code (in the spirit of the ESCAPE/MERLIN approach).

4.4. *Event-based approaches*

Many process modeling languages offer events and triggers as a mechanism for defining process models. For example, in APPL/A [36] triggers can be defined in order to react on operations on relations. In Adele/TEMPO [9], trigger definitions may be attached to entity and relationship types. A software process is modeled as a graph of workspaces, and triggers are used to define cooperation policies.

Triggers are useful for reactive programming, but they are rather low-level constructs and have to be employed with care. In DYNAMITE, we have tried to make disciplined use of triggers, mainly in order to define reactions on state transitions.

4.5. *Graph rewriting*

Since the first publications on graph grammars at the end of the 60's, a rich variety of theoretical approaches have been developed, including e.g. NLC grammars, algebraic approaches, hyperedge replacement, and logic-based approaches. An excellent survey of the current state-of-the-art is given in a recent handbook [31].

However, there is wide discrepancy between theory and practice of graph rewriting. Rather than operating in "paper and pencil mode", we are using a full-fledged development environment which is based on a sophisticated language and supports rapid prototyping. The family of development environments based on graph rewriting is still small, and we believe that PROGRES is one of its leading members (see [35] for a comparison and survey).

To conclude this section, we briefly discuss two approaches which apply graph rewriting to software process modeling. GRIDS [39] integrates different perspectives of software engineering into a coherent model. These perspectives correspond to the components of a software systems, views on these components, and processes. Each perspective is mapped onto an axis of a 3-dimensional grid; a node of the grid aggregates some component, view, and process. The main purpose of the model is to aid understanding. In contrast, DYNAMITE focuses on planning and enactment and takes care of all the details involved in managing the dynamics of software processes.

Project flow graphs [13] represent software documents, their versions, dependencies between documents, compatibilities between versions, tools with inputs and outputs, and actors. Project flow graphs are mainly concerned with the consistency of product configurations. The way a software process is modeled is primarily inspired by classical software configuration management. A novel contribution consists in the application of vector clocks — originally developed for distributed systems — to control the consistency between document versions. There are only weak relations to DYNAMITE. For example, tasks are only defined implicitly by granting access rights, there are no control flow and feedback relations, no communicating state machines, etc.

5. Conclusion

We have presented a graph-based approach to managing software processes. We have applied DYNAMITE to several sample processes, including the ISPW6/7 example [22] which, however, does not demonstrate the dynamics for which DYNAMITE has been designed (the task net is essentially static). The sample process presented in this paper has also been encoded in PROGRES and covers 120 pages (including the domain-independent part of the specification).

DYNAMITE forms an important part of an overall *administration model* [28]. In addition to process management, the administration model covers management of products and resources. The former addresses versions and configurations of doc-

uments (requirements definitions, designs, module implementations, etc; see [37]). The latter is concerned with both human and technical resources and regulates the mapping from abstract resources with required properties to matching actual resources [15]. Integration of these submodels is currently under way.

Based on the DYNAMITE model, we are developing a *process-centered software engineering environment* with the help of rapid prototyping [16]. The PROGRES development environment [34] is used to generate an end-user prototype directly from the specification. Before the prototype generator was available, we had to translate specifications manually into an ordinary programming language such as Modula-2 or C. For the specification of our sample process, the generator creates about 110,000 lines of code in C (including the code for the domain-independent part of the specification).

However, even now development of an environment from the underlying specification still requires substantial effort. In particular, user-friendly *views* have to be offered which hide the complexity of the internal data structures; i.e., we have to bridge the gap between the external representations of Sec. 2 and the internal representations of Sec. 3. Furthermore, the views have to be tailored to the needs of different user roles. For example, a project manager requires graphical, global views of the overall process, while a developer is primarily interested in those cutouts which are relevant for his own work.

As a final remark, let us stress that PROGRES is a fairly general specification language which has not been designed specifically for process modeling. In particular, our experience has shown that parameterization (Sec. 3.3 and 3.4) is both time- and space-consuming. Therefore, we are designing an environment for process modelers [23] which provides high-level support above the PROGRES level (definition of task schemas, state transition diagrams, conditions for state transitions, event handlers, etc.). These definitions will be compiled into PROGRES where further adaptations can be performed, if required.

References

1. S. C. Bandinelli, A. Fuggetta, and C. Ghezzi. Software process model evolution in the SPADE environment. *Transactions on Software Engineering*, 19(12):1128–1144, Dec. 1993.
2. N. S. Barghouti, D. S. Rosenblum, D. C. Belanger, and C. Alliegro. Two case studies in modeling real, corporate processes. *Software Process — Improvement and Practice*, 1(1):17–32, Aug. 1995.
3. G. A. Bolcer and R. N. Taylor. Endeavors: A process system integration infrastructure. In *Proceedings of the 4th International Conference on the Software Process*, Brighton, England, Dec. 1996. IEEE Computer Society Press.
4. S. Budkowski and P. Dembinski. Introduction to Estelle. *Computer Networks and ISDN Systems*, 14:3–23, 1987.
5. W. Deiters and V. Gruhn. The FUNSOFT net approach to software process management. *International Journal of Software Engineering and Knowledge Engineering*, 4(2):229–256, 1994.
6. 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.
7. B. Dellen, K. Kohler, and F. Maurer. Knowledge based techniques to increase the flexibility of workflow management. *Data & Knowledge Engineering Journal*, to appear, 1997.
8. G. Engels and L. Groenewegen. SOCCA: Specifications of coordinated and cooperative activities. In Finkelstein et al. [11], pages 71–102.
9. J. Estublier and R. Casallas. The Adele configuration manager. In W. Tichy, editor, *Configuration Management*, pages 99–134, New York, 1994. John Wiley and Sons.
10. C. Fernström. PROCESS WEAVER: Adding process support to UNIX. In *Proceedings of the 2^{nd} International Conference on the Software Process - Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, Feb. 1993.
11. A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Advanced Software Development Series. Research Studies Press (John Wiley), Chichester, UK, 1994.
12. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, Oct. 1996.
13. G. Heidenreich, M. Minas, and D. Kips. A new approach to consistency control in software engineering. In *Proceedings of the 18th International Conference on Software Engineering*, pages 289–297, Berlin, Mar. 1996. IEEE Computer Society Press.
14. 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.
15. P. Heimann, C.-A. Krapp, M. Nagl, and B. Westfechtel. An adaptable and reactive project management environment. In Nagl [27], pages 504–534.
16. 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.
17. 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.
18. W. S. Humphrey and M. I. Kellner. Software process modeling: Principles of entity process models. In *Proceedings of the 11th International Conference on Software Engineering*, pages 331–342, Pittsburgh, PA, May 1989. IEEE Computer Society Press.
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. G. Junkermann. A dedicated process design language based on EER models, statecharts and tables. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, pages 487–496, Rockville, Maryland, June 1995. Knowledge Systems Institute.
21. 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, PA, May 1989. IEEE Computer Society Press.
22. M. Kellner et al. Software process modeling example problem. In *Proceedings of the 6th International Software Process Workshop*, pages 19–30, Hakodate, Japan, Oct. 1990. IEEE Computer Society Press.
23. C.-A. Krapp. Parametrisierung dynamischer Aufgabennetze zum Management von Softwareprozessen. In J. Ebert, editor, *Softwaretechnik '96*, pages 33–40, Koblenz, Germany, Sept. 1996.
24. C. Liu and R. Conradi. Automatic replanning of task networks for supporting 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.
25. 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.
26. C. Montangero and V. Ambriola. OIKOS: Constructing process-centered SDEs. In Finkelstein et al. [11], pages 131–152.
27. M. Nagl, editor. *Building Tightly-Integrated Software Development Environments: The IPSEN Approach*. LNCS 1170. Springer-Verlag, Berlin, 1996.
28. M. Nagl and B. Westfechtel. A universal component for the administration in distributed and integrated development environments. Technical Report AIB 94-8, Technical University of Aachen, 1994.
29. L. Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, pages 2–13, Monterey, CA, 1987. IEEE Computer Society Press.
30. 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.
31. G. Rozenberg et al. *Handbook on Graph Grammars*, volume 1. World Scientific, Singapore, 1996.
32. A. C. Sarma. Introduction to SDL-92. *Computer Networks and ISDN Systems*, 28:1603–1615, 1996.
33. A. Schürr. Introduction to the specification language PROGRES. In Nagl [27], pages 248–379.
34. 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.
35. A. Schürr, A. Winter, and A. Zündorf. Developing tools with the PROGRES environment. In Nagl [27], pages 356–369.
36. S. M. Sutton, D. Heimbigner, and L. J. Osterweil. APPL/A: A language for software process programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.
37. B. Westfechtel. A graph-based system for managing configurations of engineering design documents. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):549–583, Dec. 1996.
38. P. Young. *Customizable Process Specification and Enactment for Technical and Non-Technical Users*. PhD thesis, University of California Irvine, 1994.
39. 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*, pages 48–59, Berlin, Mar. 1996. IEEE Computer Society Press.