Graph-Based Specification of a Management System for Evolving Development Processes*

Markus Heller[1] and Ansgar Schleicher[2] and Bernhard Westfechtel[3]

[1] Lehrstuhl für Informatik III, RWTH Aachen
Ahornstrasse 55, D-52074 Aachen
heller@i3.informatik.rwth-aachen.de
[2] DSA Daten- und Systemtechnik GmbH
Pascalstr. 28, D-52076 Aachen
Ansgar.Schleicher@dsa.de
[3] Lehrstuhl für Informatik III, RWTH Aachen
Ahornstrasse 55, D-52074 Aachen
bernhard@i3.informatik.rwth-aachen.de

Abstract. Development processes are inherently difficult to manage. Tools for managing development processes have to cope with continuous process evolution. The management system AHEAD is based on long-term experience gathered in different disciplines (software, mechanical, or chemical engineering). AHEAD provides an integrated set of tools for evolving both process definitions and their instances. AHEAD is based on graphs which are formally specified and manipulated by programmed graph transformations.

1 Introduction

Development processes in different disciplines such as software, mechanical, or chemical engineering share many features. Unfortunately, one of these common features is that they are hard to manage. Development processes are highly creative and therefore can be planned only to a limited extent. The tasks to be performed depend on the product to be developed, which is not known in advance. Alternative designs (variants) are explored to arrive at an optimal solution. Feedback may occur frequently — including not only spontaneous feedback raised by design errors in earlier steps, but also anticipated feedback which may be used to improve the design or to select among variants of the design. Finally, development methods such as concurrent or simultaneous engineering require sophisticated coordination between inter-dependent design activities.

In order to build effective tools for managing development processes, one must face the challenge of *process evolution*. While this has been recognized widely, current *management systems* can cope with process evolution only to a limited extent. In particular, this applies to workflow management systems [1] which were designed for repetitive business processes, e.g., by automating routine work in banks, insurance companies, administrations, etc. In such systems a high number of workflows are executed according to a common definition, ensuring that work is performed following a pre-defined

* The work presented in this paper was carried out in the Collaborative Research Center IMPROVE, which is supported by the Deutsche Forschungsgemeinschaft.

procedure. This approach cannot be transferred to development processes because it does not take process evolution into account: Developers would perceive themselves being tied in a straight-jacket so that they cannot perform their creative work as desired.

In this paper, we present the comprehensive evolution support [2] offered by *AHEAD* [3], an *A*daptable and *H*uman-Centered *E*nvironment for the M*A*nagement of *D*evelopment Processes. AHEAD is based on nearly 10 years of work on development processes in different engineering disciplines. So far, we have applied the concepts underlying the AHEAD system in mechanical, chemical and software engineering.

The AHEAD system is based on a formal specification for two reasons. First, the specification clearly defines the management data and the effects of commands provided by the system. Second, code is generated from a high-level specification, resulting in significant savings of implementation effort. As specification language, we selected PROGRES[4], a high-level language for specifying programmed graph rewriting systems. Development processes can be represented in a natural way as graphs, and their evolution can be specified by graph transformations.

2 Example

This section demonstrates how the AHEAD system supports the management of evolving development processes. For this purpose, AHEAD offers *dynamic task nets* [5]. Within dynamic task nets, tasks describe units of work to be done. Tasks are organized hierarchically, i.e., a task may be decomposed into a set of subtasks. Control flow relationships define the order of subtask enactment. Data flow relationships connect input and output parameters of tasks and allow to exchange documents between them. Feedback relationships model cycles within the process which may stem from planned iterations or occurring exceptional situations. Software processes evolve continuously and it is usually impossible to completely plan a process before its enaction. Therefore, dynamic task nets are designed to be editable and enactable in an interleaved fashion.

With respect to dynamic task nets, we have to distinguish between process definitions and process instances. A *process instance* represents a specific development process being executed. In contrast, a *process definition* describes a whole class of processes. It describes the structures of processes of this class, and the constraints that have to be met. Process instances refer to definitions; e.g., the task `ImplementUI` is an instance of the task type `Implement`.

2.1 Wide Spectrum Approach

Development processes span a *wide spectrum*, ranging from highly structured, repetitive, and well-known ones to ad hoc, one-shot, and unknown ones. In this subsection, we describe how AHEAD covers the whole spectrum of processes by offering flexible means for process definitions.

Figure 1a shows a *type-level definition* for an idealized software process. It consists of task types for system design, component implementation, and test. These task types are connected through control flow and feedback flow types. The task types own input

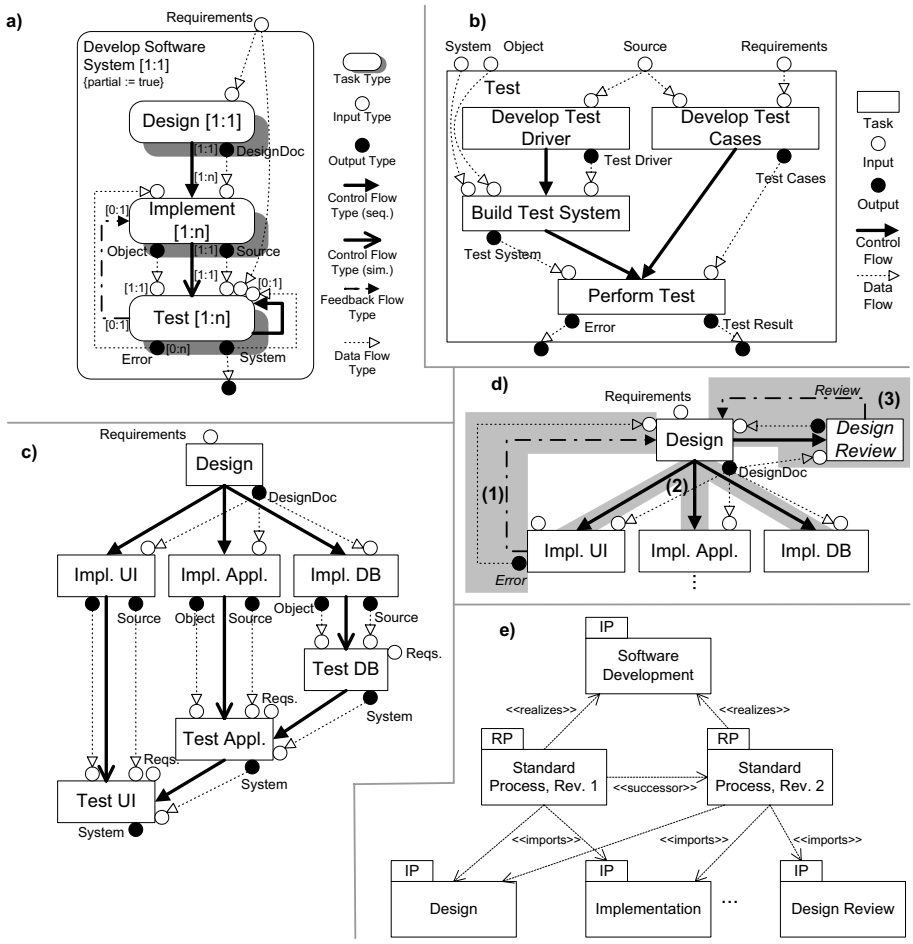


Fig. 1. Example process

and output parameter types which are connected by data flow types defining the potential channels for data exchange. The process definition is vague, which is modeled through the partial-flag. This means that the process modeler decides to permit *unconstrained types*: In addition to explicitly defined types, the process manager may instantiate pre-defined unconstrained types for those tasks which have not been anticipated by the process modeler. Permitting unconstrained types potentially leads to partially typed task nets but enhances flexibility.

Besides these *structural constraints*, the behavior of elements may be defined using *behavioral patterns* (some of which are shown in Table 1). For each structural element, a number of behavioral patterns are predefined. It is e.g. stated that a control flow may have standard, simultaneous or sequential semantics. Standard semantics means that the flow's source is to be terminated before its target. Simultaneous semantics additionally

model element	*property*	*values*
control flow	enactment order	standard, sequential, simultaneous
parameter	versioned	true, false
input	available	on start, later
input	consumption mode	automatic, manual

Table 1. Parametric definition of behavior

requires the source to be activated before the target. With sequential semantics the flow's source has to be terminated before the target is activated. For parameters behavioral patterns regarding the versioning of documents exchanged via a particular parameter, the time of availability of a document at a particular input parameter or the automation degree of input consumption are specified. One set of behavioral patterns is defined as the default behavior. E.g. for input parameters the default behavior is defined as (versioned:=true; available:=later; consumption-mode:=manual). The most unrestricted behavioral patterns are used to define the default semantics which are e.g. used for unconstrained types.

Assigning one or multiple of these behavioral patterns to particular types of the process definition results in parameterization of an enacted dynamic task net's behavior. The enactment order of related tasks may be adapted using different control flow semantics; the activation of a task can be delayed by requiring on-start availability of its inputs, etc. In the example of Figure 1a we have assigned sequential and simultaneous behavioral patterns to the control flow types.

Figure 1a defines a partially typed subprocess. In the current version of the process definition, we assume there is no refining definition for the design task type (untyped, i.e., unconstrained subprocess) and a type-level definition for the implementation task type whose description we omit. Testing is well understood, and a stable process has been defined as an instance-level process definition. The corresponding *instance pattern* is displayed in Figure 1b. It consists of tasks for test driver and test case development, test system building and test execution. This instance pattern can be directly implanted into a task net without interactive planning steps. To summarize, the definition of our sample process covers the whole spectrum from unconstrained processes on one end to instance patterns on the other end.

2.2 Consistency Control

We now turn to the description of flexible process instance support. Our main focus lies on *consistency control*. To enhance flexibility, the process instance may deviate from its definition, resulting in inconsistencies. The process manager retains control on where such inconsistencies are allowed and where they are prohibited.

In the beginning of our sample process it can be determined only that the new system needs to be designed. An initial top-level task net therefore contains no other tasks than a typed design task. The design process may then be described by a subordinate task net. As there is no process definition for design tasks available, the process manager creates an unconstrained ad-hoc task net (not shown). This task net comprises tasks for

creating a coarse design to identify the system's components and their interrelations and for providing a detailed design for each of these components (e.g. user interface, application logic, database system).

After a design has been produced by the complex design task, the top-level task net is completed. One implementation and one test task are created for each component and a bottom-up testing strategy is enforced (cf. Figure 1c). During the user interface's implementation an error in the design is detected which has to be resolved. Accordingly, a feedback flow is introduced into the net and an error report is created and sent to the design task (cf. (1) in Figure 1d). As this feedback flow, the error parameters and the data flow are not part of the process definition (cf. Figure 1a) but unconstrained types are allowed by it, a *weak consistency* results from this task net manipulation. By contrast, *strong consistency* implies that only constrained (explicitly defined) types are instantiated in a way that conforms with the process definition.

As the implementation tasks are sequentially dependent on the design task, they may only be active when the design task is terminated. But, as a result of the introduced feedback flow, the terminated design task has to be reactivated for error correction. This results in the implementation and design task being active at the same time which leads to a *behavioral inconsistency* (cf. (2) in Figure 1d).

As design errors may become costly the further the software process proceeds, the process manager makes the decision to let the next design version be reviewed. Accordingly, a design review task is created in the context of the design task. A feedback flow is used to model the planned iteration between the design and review tasks (cf. (3) in Figure 1d). As the design review task is typed but the type is not part of this process definition, a *structural inconsistency* emerges.

Let us summarize the general features of consistency control: By default, a process instance must be consistent with the process definition. However, the process manager may allow inconsistencies selectively by means of local "switches" which are provided for each subnet in a task hierarchy. By default, consistency enforcement is switched on, but the process manager may switch it off deliberately to tolerate temporary or even permanent inconsistencies. When consistency enforcement is switched off, the process manager may manipulate the respect subnet (being part of the process instance) such that it deviates from the process definition. Consistency enforcement may be switched on again only after all inconsistencies have been removed (either by operations applying only to the process instance or by migrating to a revised process definition, see below).

2.3 Definition-Level Evolution and Migration

In some cases, deviating process instances contain new and valuable process knowledge which should be made available to other process instances. This leads to an extension of the process definition, which is structured in a modular way through *packages*. Interface packages contain the task and parameter types of a task, while realization packages contain the refining definitions of one interface. In the context of our example, the process modeler decides to revise the top level process definition. The new *package version* contains a feedback flow type between implementation and design task types together with the necessary parameter and data flow types. The design review type is embedded into the definition and the behavioral description of control flow types between design

and implementation tasks is switched from sequential to simultaneous. The resulting package structure is shown in Figure 1e. The successor dependency between packages denotes the history of process definition extensions.

As mentioned in the beginning of this section, there might be multiple running instances of one process definition. The manager may select the instances to migrate on-demand and determine the time of migration. *Migration* means that an instance is upgraded to the new definition. In our example, the process manager decides to migrate the process instance of Figure 1d such that it is consistent with the new definition. Other running instances may also be migrated. Their consistency with respect to the new process definition version is very unlikely. Therefore, the inconsistencies induced by migration are again signaled to the manager, but migration can always be performed.

If the manager of another instance which is consistent with the original process definition decides to migrate to the new version, parts of the instance are inconsistent or incomplete. The missing design review task denotes an incompleteness of the task net, while the sequential control flows between design and implementation tasks are a behavioral inconsistency. The manager may selectively decide which incompletenesses and inconsistencies to remove from the net. E.g., he may create and embed a design review task into the net, removing the incompleteness.

Thus, instances may be migrated to new (versions of) definitions, but that does not necessarily imply that a consistent state will be reached after migration. Both temporary and persistent inconsistencies may be tolerated by turning off consistency enforcement.

3 Formal Specification

3.1 The AHEAD System

We have realized the process evolution approach described above in the AHEAD system (Figure 2). A process modeler creates an external process definition in a *modeling environment* for UML (in this paper, we used a different, more concise notation for demonstration purposes). The model packages are transformed automatically into an internal process definition which is hidden from the external user. Internally, processes are defined in *PROGRES* [4], a specification language for programmed graph transformations. Internal process definitions are based on a *process meta model* which has been defined once beforehand (again in PROGRES). The specifications for internal process definitions and the process meta model are translated by the PROGRES compiler into equivalent C (or Java) code which operates on a graph-based DBMS (GRAS). The generated code is executed by a *management environment* supporting process managers in planning, analyzing, and monitoring software processes, and a *work environment* which provides software engineers with tools managing agendas and workspaces.

Wide spectrum processes are defined in the modeling environment, which is also used to evolve process definitions by versioning model packages. Consistency control is handled by the management environment, which offers commands for enabling and disabling inconsistencies and informs process managers about deviations from the process definition. In addition, the management environment supports migration of process instances to a modified process definition. In the following, we constrain the presentation to the level of PROGRES specifications since our approach to process evolution is

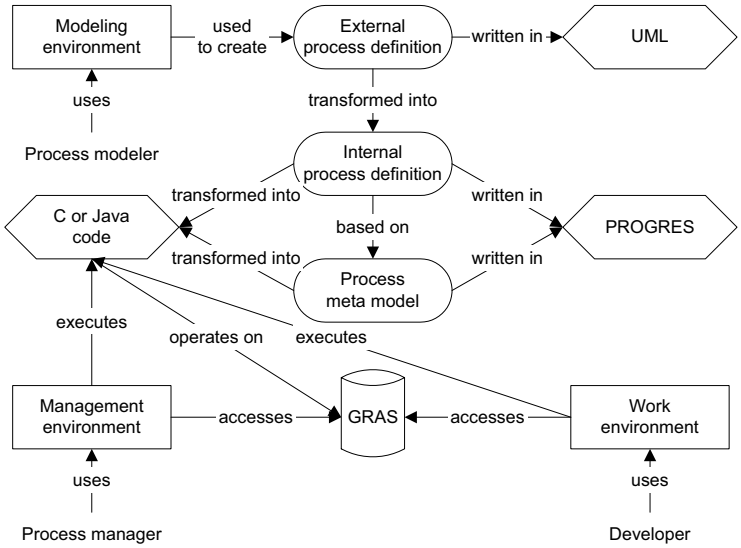


Fig. 2. Overview of the AHEAD system

formally defined at that level. For the UML modeling environment and the transformation from UML to PROGRES, see [6]. Since the overall specification covers far more than 100 pages, we can present only small fragments to illustrate the underlying principles. Furthermore, we can give only rather cursory explanations; for further technical details, the reader is referred to [2].

3.2 Process Meta Model

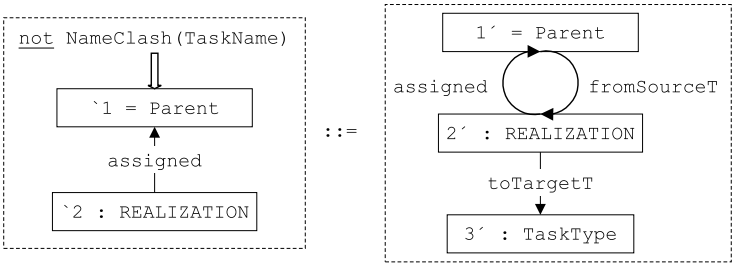
In the sequel, we sketch how the process meta model is defined in PROGRES (see [5] for a more comprehensive description). We start out in this subsection by presenting the meta model elements and their interrelations. In the next subsections we explain enhancements of this process meta model needed for wide spectrum process model definition, consistency control and process evolution, respectively.

The upper part of Figure 3 shows a cutout of the *graph schema* for dynamic task nets. The graph schema defines the components of dynamic task nets in terms of node classes, attributes, and edge types. Tasks are modeled by the node class `TASK`, which is a subclass of `ENTITY`. A task has an intrinsic `Name` attribute — inherited from `ENTITY` — as well as an additional `State` attribute. The process meta model defines a common state diagram which applies to all types of tasks, but which can further be adapted at the process definition level. Task relations are represented by nodes of class `TASK_RELATION` and edges of type `fromSourceT` and `toTargetT`, respectively. Finally, a node of class `REALIZATION` is used to represent the realization of a task being `assigned` to its interface (represented by the task node itself).

The lower part of Figure 3 gives two examples of base operations defined in the process meta model. With the help of the *graph rewrite rule* (*production*) `BaseCre-`

```
node class ENTITY is a ITEM;
  intrinsic Name : string;
end;
node class TASK is a ENTITY
  intrinsic
    State : {InDefinition, ..., Failed} := InDefinition;
end;
node class TASK_RELATION is a ITEM end;
edge type fromSourceT : TASK -> TASK_RELATION;
edge type toTargetT : TASK_RELATION -> TASK;
node class CONTROL_FLOW is a TASK_RELATION end;
node class FEEDBACK_FLOW is a TASK_RELATION end;
node class REALIZATION is a ENTITY, TASK_RELATION end;
edge type assigned : REALIZATION -> TASK;

production BaseCreateSubTask
  ( Parent : TASK; TaskName : string;
    TaskType : type in TASK; out NewTask : TASK ) =
```




```
  condition `1.State in {InDefinition, Active};
  transfer 3´.Name := TaskName;
  return NewTask := 3´;
end;

transaction BaseStart( Task : TASK ) =
    (Task.State in {Waiting, Suspended}
     and (Task.IsRoot or Task.=ToParent=>.State = Active))
  & Task.State := Active
end;
```

Fig. 3. Specification the process meta model

`ateSubTask`, a subtask is inserted into a task net. This is achieved by creating a new task node (node `3'` on the right-hand side) and connecting it to the realization node for the parent task. The base operation checks several constraints which are fixed parts of the process meta model. For example, insertion of the new task must not imply a name clash (see restriction on the left-hand side), and it can be performed only in certain states of the parent task (see condition part below left- and right-hand side).

The *transaction* `BaseStart` is used to execute a state transition from the states `Waiting` or `Suspended` into the state `Active`. The body of the transaction consists of a sequence (`&`) of statements. If one statement fails, the whole transaction fails and leaves the task net unaffected. First, it is checked that the current task resides in a valid source state for the `Start` transition. Furthermore, the task must either be located at

the root of the task hierarchy, or its parent must be `Active`. If these conditions hold, the `State` attribute of the current task is assigned the value `Active`.

Please note that all base operations check and enforce both structural and behavioral constraints. Since planning and enactment may be interleaved, a structural operation such as `BaseCreateSubTask` has to check behavioral constraints with respect to the state of the parent task. Moreover, we would like to emphasize that only minimal behavioral constraints are enforced in the process meta model. For example, `Base-Start` merely checks the parent task's state. All constraints checked at this level are *hard constraints* built into the process meta model.

3.3 Wide Spectrum Approach

In the following, we describe how the wide spectrum of process definitions is represented in the PROGRES specification. First, we go into type-level process definitions; the representation of instance patterns will be discussed briefly at the end of this subsection.

The production `BaseCreateTask` presented above instantiates a domain-specific task type, but it does not check *domain-specific constraints* (e.g., whether an instance of that type is permitted in the realization of the parent task). In Subsection 3.4, we will describe how base operations are extended with these checks. Before that, we explain how the information used by these checks is represented in the graph schema.

Domain-specific types introduced in a process definition are represented by *node types*, which are instances of node classes. Node types may be passed as parameters (e.g., the type of a task to be created, see Figure 3), and they may be stored as attribute values (see below). To express domain-specific constraints, we make use of *meta attributes*, i.e., attributes which are attached to classes or types rather than node instances.

The upper part of Figure 4 shows a cutout of the graph schema for the process meta model, extended with meta attributes. At the level of the meta model, these attributes are initialized in the most "liberal" way. For example, the meta attribute `Declared-Parameters` of class `TASK` is a set-valued attribute (cardinality `[0:n]`) which contains all `PARAMETER` types, i.e., parameters of any type are allowed. The lower part of Figure 4 shows a cutout of a process definition, which introduces node types and redefines the values of meta attributes to express domain-specific constraints. For example, nodes of type `DevelopSoftwareSystem` may have only parameters of type `Requirements` (input) and `System` (output). Please note that both structural and behavioral constraints are represented by meta attributes (see e.g. the type `Design-ToImplement`, which is used to represent sequential control flows from `Design` to `Implement` tasks).

Instance patterns such as given in Figure 1b are not mapped onto the graph schema. Rather, a transaction is generated which contains a sequence of calls to base operations creating the instance pattern's elements (see [6], which also explains why we do not generate a production instead).

```
node class TASK is a ENTITY
  meta
    DeclaredParameters   :
       type in PARAMETER [0:n] := PARAMETER;
    DeclaredRealizations :
       type in REALIZATION [0:n] := REALIZATION;
  intrinsic
    State : {InDefinition, ..., Failed} := InDefinition;
end;
node class TASK_RELATION;
  meta
    SourceTypes : type in TASK [0:n] := TASK;
    TargetTypes : type in TASK [0:n] := TASK;
end;
node class CONTROL_FLOW is a TASK_RELATION
  meta
    EnactmentOrder :
       {standard, simultaneous, sequential} := standard;
end;
```

```
node type Task : TASK end;
node type Realization : REALIZATION end;
node type ControlFlow : CONTROL_FLOW end;
node type DevelopSoftwareSystem : TASK
  redef meta
    DeclaredParameters = {Requirements, System};
    DeclaredRealizations = {PhaseDevelopment};
end;
node type DesignToImplement : CONTROL_FLOW
  redef meta
    SourceTypes := {Design};
    TargetTypes := {Implement};
    EnactmentOrder := sequential;
end;
```

Fig. 4. Declaration of node types and meta attributes

3.4 Consistency Control

Now we can establish the connection between the base operations of Section 3.2 and the domain-specific constraints of Section 3.3. We use the term *soft constraints* to emphasize that they can be enforced selectively (in contrast to the process meta model's hard constraints). For checking soft constraints, we introduce *derived attributes* which signal inconsistencies. While the values of intrinsic attributes are assigned explicitly, the values of derived attributes are defined by evaluation rules and are updated automatically.

For consistency checking, we attach Boolean attributes to the nodes of task nets (Figure 5). Consistency calculation is performed in an object-oriented manner. Every language element can calculate its own consistency within its context. In the node class `ITEM`, which serves as the root of the class hierarchy, attributes are defined for checking structural and behavioral consistency. As default, these attributes evaluate to true; the evaluation rules are redefined in subclasses of `ITEM`. These evaluation rules refer to

```
node class ITEM
  derived
    StructurallyConsistent : boolean = true;
    BehaviorallyConsistent : boolean = true;
end;
node class ENTITY is a ITEM ... end;
node class TASK is a ENTITY
  intrinsic
    AllowInconsistencies : boolean := false;
  ...
end;
node class TASK_RELATION is a ITEM ... end;
node class CONTROLFLOW is a TASK_RELATION
  derived
    TypesConsistent  : boolean =
      self.<-fromSourceT-.type in self.SourceTypes and
      self.-toTargetT->.type in self.TargetTypes;
    StatesConsistent : boolean =
      [self.EnactmentOrder = sequential ::
         self.-toTargetT->.State in
           {Active, Suspended, Done, Failed} =>
         self.<-toSourceT-.State in {Done, Failed}
       | ... ];
  redef derived
    StructurallyConsistent = self.TypesConsistent and ...;
    BehaviorallyConsistent = self.StatesConsistent and ...;
  ...
end;

transaction CreateSubTask
  ( Parent : TASK; TaskName : string;
    TaskType : type in TASK; out NewTask : TASK ) =
    BaseCreateSubTask(Parent, TaskName, TaskType, out newTask)
  & [ not Parent.AllowInconsistencies ::
        NewTask.StructurallyConsistent and ...
    | true ]
end;

transaction Start( Task : TASK ) =
    BaseStart(Task)
  & [ not Task.ToParent.AllowInconsistencies ::
       (for all cf in AdjacentControlFlows(Task)
          cf.BehaviorallyConsistent
        end) and ...
    | true ]
end;
```

Fig. 5. Specification of consistency control

the values of meta attributes. For example, in the class `CONTROL_FLOW` structural consistency is determined using the derived attribute `TypesConsistent`, which checks whether the type of the source and target tasks are contained in the sets `SourceTypes` and `TargetTypes`, respectively. Similarly, the behavioral consistency depends on the derived attribute `StatesConsistent`, which checks whether the states of source and target are compatible with the `EnactmentOrder` of the control flow type. For example, in the case of a sequential control flow the target may only be active, suspended, failed, or done if the source is terminated. Thus, there are two derived

attributes signaling the overall behavioral and structural consistency of an element and an arbitrary number of additional derived attributes performing fine-grained checks. By retrieving the values of these latter attributes the cause of a particular inconsistency can be identified.

In `TASK`, an intrinsic attribute `AllowInconsistencies` is defined whose default value is false. To allow for inconsistencies in a certain task net, the process manager changes its value to true. After all inconsistencies have been removed, `AllowInconsistencies` may be switched off again.

The base operations of the process meta model are embedded into *wrappers* which can check soft constraints. The base operation is called first. Afterwards, the *AllowInconsistencies* flag is checked which is attached to the surrounding task net's root node. If inconsistencies are not allowed, but have been introduced by the base operation, the operation is rolled back, and the wrapper transaction fails. Figure 5 demonstrates this by the wrappers for creating a subtask and starting a task.

In the example of Figure 1d, the process manager may create an unconstrained feedback flow from implementation to design when inconsistencies are still disabled (weak consistencies are allowed). After that, the `AllowInconsistencies` flag has to be set to true. Only then may the `Design` task be reactivated, resulting in a behavioral inconsistency with respect to the sequential control flow to the source of the feedback flow. Moreover, insertion of a (typed) `DesignReview` task results in a structural inconsistency. These inconsistencies are signaled by the respective derived attributes.

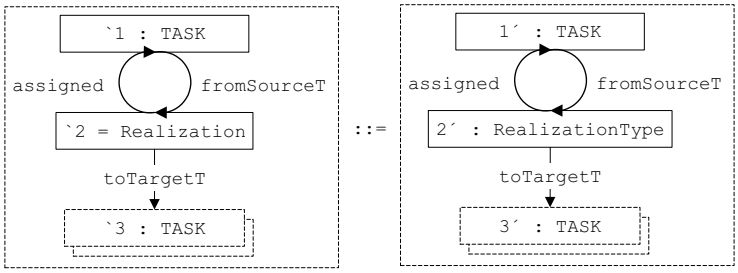
3.5 Definition-Level Evolution and Migration

Due to the lack of space, we cannot elaborate on the technical details of definition-level evolution and migration. Rather, we briefly describe the basic ideas.

Evolution of process definitions is supported at the level of packages. PROGRES does provide packages, but it supports neither package versions nor type versions. Thus, package versions as shown in Figure 1e are simulated by appending version numbers to package names (likewise for node types). From the perspective of PROGRES, definition-level evolution implies an extension of the graph schema and (for instance patterns) addition of new operations (i.e., no loss of information). To make definition-level evolution effective, the extended external process definition is transformed in a batch process into an extended internal process definition in PROGRES, which is compiled into C in turn. The source code has to be compiled and linked to produce an AHEAD system for the extended process definition. Subsequently, migration be initiated.

Migrating a subnet to a new definition may involve insertion of new task net elements as well as deletion or conversion of old task net elements. Corresponding *base operations* for insertion and deletion are available anyhow. The base operations for conversion upgrade task net elements to new types. Since PROGRES does not offer type changes of nodes, conversion has to be simulated by deleting the old node and inserting a new one, preserving the embedding. As an example, Figure 6 shows the graph rewrite rule for migrating a task realization. Please note that the condition part excludes realization of terminated tasks from migration because historical data should not be changed post mortem.

```
production BaseMigrateRealization
  ( Realization : REALIZATION;
    RealizationType : type_in REALIZATION;
    out NewRealization : REALIZATION ) =
```




```
  condition `1.State in {InDefinition, Active, Suspended};
  transfer 2´.Name := `2.Name;
  return NewRealization := 2´;
end;
```

Fig. 6. Base operation for migrating a realization

In addition, *complex operations* are provided to (partially) automate the migration process. A lot of user interactions are required if the process manager has to call the base operations on each individual element. Please note that the base operations require that the target type be passed explicitly, which makes the migration even more tedious. Therefore, AHEAD assists the process manager by a set of transactions which select the elements to be migrated and try to infer the respective target type. Unfortunately, it may happen that the target type may not be inferred in a unique way, in particular in the case of unconstrained types. Thus, currently migration normally requires user interactions.

3.6 Discussion

To conclude this section, we discuss a technical issue concerning the way we have applied the PROGRES language to process modeling. Our overall approach is illustrated on the left-hand side of Figure 7. The process meta model is encoded in PROGRES at the level of node classes, the process definition adds domain-specific parts of the specification at the level of node types, and process instances are represented as node instances (of node types) in the host graph at run time. This solution is called *specification mapping* (i.e., mapping of the process definition into the specification).

The right-hand side of Figure 7 shows an alternative realization: the *host graph mapping* (i.e., mapping of the process definition into the host graph). Following this alternative, the host graph would be composed of two parts representing process definitions and instances, respectively. Accordingly, the PROGRES specification would consist of two analogous parts dealing with graphs and operations for process definitions and instances, respectively.

Specification and host graph mapping have complementary advantages and drawbacks:

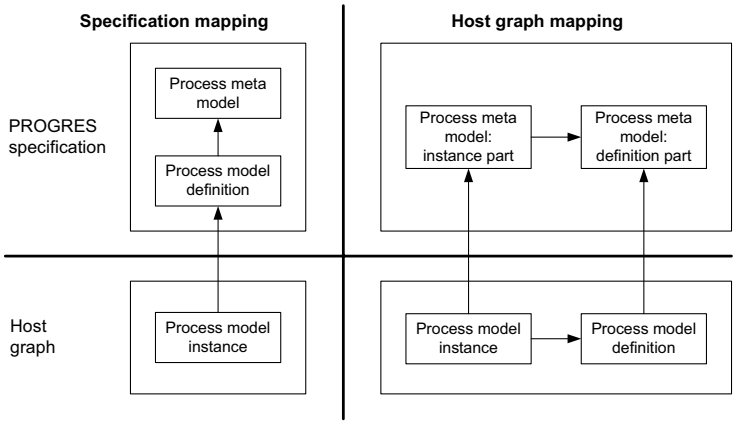


Fig. 7. Realization alternatives

Uniformity In the case of the specification mapping, process definitions and instances are represented in different ways. In contrast, the host graph mapping would handle process definitions and their instances in a uniform way.

Modeling effort The specification mapping reduces the modeling effort considerably because all features of PROGRES may be exploited for process definitions. In contrast, for the host graph mapping features which are already available in PROGRES would have to be simulated.

Efficiency In the case of the specification mapping, the process definition is compiled into C code, which can be executed efficiently. In contrast, the host graph mapping would require time-consuming interpretation of the process definition.

Flexibility In the case of the specification mapping, process evolution at the definition level requires two batch compilation steps (UML $\rightarrow$ PROGRES $\rightarrow$ C). In contrast, the host graph mapping would allow for changes to the process definition on the fly, without any disruption.

This discussion shows that selecting one of these alternatives is a classical engineering design decision: In neither case, a solution is obtained which is optimal with respect to all aspects discussed above. We have favored specification mapping because it is more efficient and because it reduces the modeling effort. On the other hand, the host graph mapping would handle process definitions and instances uniformly, and it would be more flexible. We consider host graph mapping a viable alternative in the case of rather simple process definitions; otherwise, the modeling effort would become too high.

4 Related Work

AHEAD differs from commercial *workflow management systems* [1] considerably. Essentially, in a workflow management system a process instance is created by (conceptually) copying the definition (a “process program”) and executing it. In contrast,

in AHEAD a task net is built up only at runtime; planning and enactment may be interleaved seamlessly. The standards of the Workflow Management Coalition (see `www.wfmc.org`), having partially been adopted by the OMG [7], do not address process evolution.

AHEAD is based on a process meta model which bears some similarities to other meta models, e.g., the SPEM metamodel [8] defined by the OMG as an extension of the UML meta model. In SPEM, activities are modeled as operations, and software processes are primarily modeled with the help of activity diagrams. AHEAD follows a different approach and makes use of class diagrams to support process evolution through dynamic instantiation of task classes and associations [6].

The need for a *wide spectrum* approach to process management was recognized as a research challenge in [9]. It is addressed in some research prototypes of workflow management systems which provide a wide range of control flow types (e.g., Mobile [10] and FLOW.NET [11]). However, the main focus still lies on highly or medium-structured processes.

There are only a few other approaches which are capable of managing *inconsistencies*. In PROSYT [12], users may deviate from the process definition by enforcing operations violating preconditions and state invariants. However, eventually consistency has to be re-established; otherwise enactment has to be aborted. In contrast, the approach described in [13] does allow for potentially persistent inconsistencies, which raise exceptions that are handled manually or automatically. However, all of the approaches we are aware of do not address weak consistency in the presence of partial process knowledge.

A key and unique feature of our approach consists in its support for *round-trip process evolution*. In contrast, most other approaches are confined to *top-down evolution*. In [10, 14, 15], the process definition has to be created beforehand, while we allow for enacting partially known process definitions. Furthermore, both structural and behavioral consistency must be maintained during migration. This is not required in our approach, which is more flexible.

Finally, there are a few approaches which are confined to *instance-level evolution* (e.g., [16]). A specific process instance is modified, taking the current enactment state into account. However, the process definition remains unaffected. Our approach is more general since it covers both instance- and definition-level evolution (the latter of which may be used to propagate changes to more than just one instance).

5 Conclusion

In this paper, we have demonstrated the use of graph technology for the specification of process evolution support in the AHEAD management system. Development processes can be represented in a natural way as graphs, and their evolution can be specified by graph transformations. In our specification, we have exploited virtually all features offered by the specification language PROGRES, including the stratified type system, meta attributes, derived attributes, graph rewrite rules, transactions, and backtracking.

Finally, we have compared rather briefly two ways of using PROGRES for process modeling, namely specification mapping (i.e., mapping of the process definition

into the PROGRES specification) and host graph mapping (i.e., mapping of the process definition into the host graph). We have deliberately pursued the specification mapping alternative to reduce modeling effort and to increase efficiency. However, we still consider comparison and evaluation of different styles of specification an important research topic to be addressed in the future.

References

1. Lawrence, P., ed.: Workflow Handbook. John Wiley, Chichester, UK (1997)
2. Schleicher, A.: Management of Development Processes — An Evolutionary Approach. Dissertation RWTH Aachen, Deutscher Universitäts-Verlag, Wiesbaden, Germany (2002)
3. Jäger, D., Schleicher, A., Westfechtel, B.: AHEAD: A graph-based system for modeling and managing development processes. In Nagl, M., Schürr, A., Münch, M., eds.: Proc. of the Intl. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE). LNCS 1779, Springer (1999) 325–339
4. Schürr, A., Winter, A., Zündorf, A.: Graph grammar engineering with PROGRES. In Schäfer, W., Botella, P., eds.: Proc. of the European Software Engineering Conference (ESEC '95). LNCS 989, Springer (1995) 219–234
5. Heimann, P., Krapp, C.A., Westfechtel, B., Joeris, G.: Graph-based software process management. Intern. Journal of Software Eng. and Knowledge Eng. **7** (1997) 431–455
6. Jäger, D., Schleicher, A., Westfechtel, B.: Using UML for software process modeling. In Nierstrasz, O., Lemoine, M., eds.: Software Engineering — ESEC/FSE '99. LNCS 1687, Springer (1999) 91–108
7. Object Management Group Needham, Massachusetts: Workflow Management Facility Specification. Version 1.2 edn. (2000) http://www.omg.org.
8. Object Management Group Needham, Massachusetts: Software Process Engineering Metamodel Specification. Version 1.0 edn. (2002) http://www.omg.org.
9. Sheth, A., et al.: NSF workshop on workflow and process automation. ACM Software Engineering Notes **22** (1997) 28–38
10. Heinl, P., Horn, S., Jablonski, S., Neeb, J., Stein, K., Teschke, M.: A comprehensive approach to flexibility in workflow management systems. In Georgakopoulos, D., Wolfgang, Wolf, A.L., eds.: Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC '99). Volume 24-2 of ACM SIGSOFT Software Engineering Notes. (1999) 79–88
11. Joeris, G., Herzog, O.: Towards flexible and high-level modeling and enacting of processes. In Jarke, M., Oberweis, A., eds.: Proc. of the Intl. Conf. on Advanced Information Systems Engineering (CAiSE'99). LNCS 1626, Springer (1999) 88–102
12. Cugola, G.: Tolerating deviations in process support systems via flexible enactment of process models. IEEE Transactions on Software Engineering **24** (1998) 982–1001
13. Murata, T., Borgida, A.: Handling of irregularities in human centered systems: A unified framework for data and processes. IEEE Transactions on Software Engineering **26** (2000) 959–977
14. Joeris, G., Herzog, O.: Managing evolving workflow specifications. In: Proc. of the Intl. Conf. on Cooperative Information Systems (CoopIS'98), IEEE Comp. Soc. Press (1998) 310–321
15. Kradolfer, M., Geppert, A.: Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In: Proc. of the Intl. Conf. on Cooperative Information Systems (CoopIS'99), IEEE Comp. Soc. Press (1999) 104–114
16. Reichert, M., Dadam, P.: ADEPT$_{flex}$ — supporting dynamic changes without loosing control. Journal of Intelligent Information Systems **10** (1998) 93–129