# Model-Driven Software Development with Graph Transformations: A Comparative Case Study

Thomas Buchmann, Alexander Dotor, Sabrina Uhrig, and Bernhard Westfechtel

Lehrstuhl Angewandte Informatik 1, University of Bayreuth
D-95440 Bayreuth
firstname.lastname@uni-bayreuth.de

**Abstract.** Significant achievements have been made in the design and implementation of languages and tools for graph transformation systems. However, many other competing approaches have been developed for model-driven software development. We present a case study in which we applied different modeling approaches in the construction of a tool for software process management. We compare these approaches with respect to the respective levels of abstraction on which models are defined, the language concepts offered, and the resulting modeling effort. The case study identifies the benefits and shortcomings of the selected modeling approaches, and suggests areas of future improvement.

## 1 Introduction

*Model-driven software development* promises to increase the productivity of software developers significantly with the help of high-level, executable models. In many application areas, the data maintained by the system to be developed may be represented as graphs in a natural way. Furthermore, graph modifications may be described declaratively by graph transformation rules. Thus, model-driven software development can be supported by generating executable code from graph transformation rules.

To date, several languages and tools for developing *graph transformation systems* are available and have been applied in diverse application domains (e.g., PROGRES [1], Fujaba [2], MOFLON [3], AGG [4], GenGed [5], DiaGen [6], VIATRA [7], and GReAT [8]). Significant advances have been achieved in language and tool development. Moreover, graph transformations have been applied successfully in various domains [9,10]. On the other hand, dispersal of the graph transformation approach outside the graph transformation community seems to have taken place only to a limited extent.

This paper presents a *case study* for the application of graph transformations. The subject of our study is a software process management system based on dynamic task nets (Sect. 2). We have realized this application with the help of three systems (Sect. 3): (1) GMF/EMF, i.e., a combination of the Graphical Modeling Framework for graphical editors and the Eclipse Modeling Framework, which both are not based on graph transformations; (2) PROGRES, a system for specifying programmed graph transformation systems; and (3) Fujaba, an object-oriented system where graph transformation rules have been incorporated into the UML. These solutions are evaluated and compared against each other in Sect. 4.

A. Schürr, M. Nagl, and A. Zündorf (Eds.): AGTIVE 2007, LNCS 5088, pp. 345–360, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Speaking in terms of the well-known model-view-controller design pattern, we focus exclusively on the *model*, i.e., the application logic. Furthermore, we study model-driven software development from the perspective of the *user* of the respective *modeling language*. Thus, we are interested in the language concepts, the levels of abstraction at which models are defined, expressiveness of the modeling languages, model size, readability, modeling effort, and efficiency of the code generated from the model.

An important issue addressed by our case study is the significance and role of graph transformations: We consider GMF/EMF to be a framework of industrial relevance which does not make use of graph transformations. Thus, it is fair to ask for the added value of graph transformations. We hope that this case study contributes to answering this question and thus to the mission of this workshop (applications of graph transformations with industrial relevance).

## 2 Dynamic Task Nets

*Dynamic task nets* represent software processes which evolve during execution. They were described earlier, e.g. in [11]. For this case study, we considered a "light" version of dynamic task nets which comprises only some of the core concepts. The case study goes beyond previous work only inasmuch as inconsistencies with respect to the underlying process meta model can be tolerated.

Figure 1, a screenshot taken from one of the prototypes which we constructed for this case study, shows an example of a task net. Each task is represented by a box containing its name (A . . . G) and its state of execution. States and transitions are defined

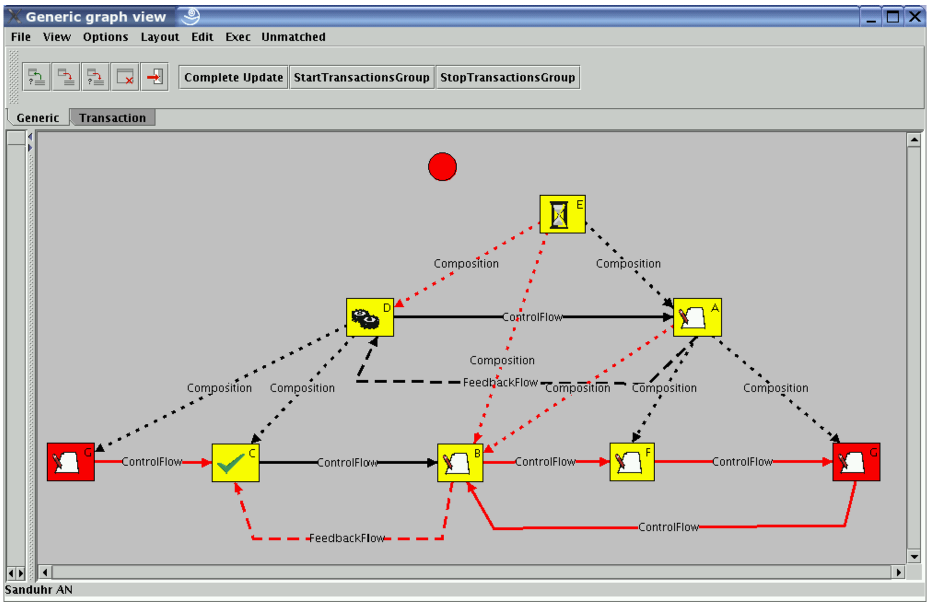


**Fig. 1.** Dynamic task net

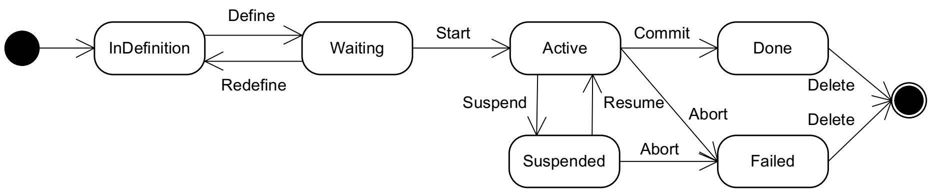


**Fig. 2.** State diagram

by the state diagram in Fig. 2. In Fig. 1, states are shown as icons (paper and pencil: `InDefinition`, sand-glass: `Waiting`, gearwheels: `Active`, tick: `Done`). Tasks are arranged in a hierarchy via composition relationships (dotted lines). Control flows (solid lines), which resemble precedence relationships in project plans, constrain the order of task execution. Finally, feedback flows (dashed lines) represent feedback to earlier steps in the process. Further concepts of dynamic task nets, e.g., data flows and task versions, are not covered here to keep the case study small.

The process meta model defines *constraints* for dynamic task nets which may be classified in two orthogonal dimensions: (1) Static constraints are defined as invariants which have to hold for each task net. Dynamic constraints are pre- and postconditions of operations which cannot be expressed as invariants. (2) Structural constraints define the rules which have to be followed when constructing task nets via edit operations. In contrast, behavioral constraints are concerned with state restrictions which have to be obeyed for the execution of tasks.

The following *static structural constraints* have to be satisfied[1]: (1) Each task must have a globally unique name. (2) Composition relationships must be free of cycles (likewise for control flows and feedback flows). (3) Each task may be contained in at most one parent task. (4) Each control flow must be either local, i.e., source and target must have the same parent, or balanced, i.e., the parents of source and target are different and are connected by a control flow (likewise for feedback flows). (5) Each feedback flow must be oriented in the opposite direction of a control flow, i.e., there must be a path of control flows from the target of the feedback flow back to the source.

All *static behavioral constraints* can be expressed via compatibilities of states of adjacent tasks. For each type of task relationship, a corresponding compatibility matrix is defined (see [12], pp. 91). Here, we discuss only compatibilities of states of tasks connected via a control flow. There are three types of control flows: If a control flow is sequential, the successor may start only after termination of the predecessor. A standard control flow is used to express that the successor can be terminated only after its predecessor. In the case of a simultaneous control flow, the successor may be activated only after its predecessor, and it may also be terminated only after its predecessor. These rules can be translated into legal and illegal state combinations. For example, the combination `Active` → `Done` is illegal for all types of control flows.

The state diagram of Fig. 2 defines *dynamic behavioral constraints* for state transitions: Each transition may be performed only in its source state and moves the task to which it is applied into its target state. In addition, there are some state constraints on

[1] There are no dynamic structural constraints.

edit operations. In general, dynamic task nets allow for seamless interleaving of editing and execution. However, tasks and their contexts (e.g., incoming control flows) must not be modified after termination, i.e., the history must not be changed.

In the screenshot of Fig. 1, inconsistencies are marked in red color[2]. The tasks on the left and on the right are marked as inconsistent because both have the same name (`G`). The composition relationships ending at `B` (bottom middle) violate the task hierarchy. The control flows connecting `B`, `F`, and `G` form a cycle (bottom right). The composition relationship from `E` (top) to `D` is behaviorally inconsistent: a child task must not be activated before its parent. Similarly, the control flow from `G` to `C` (bottom left) is inconsistent because the states of source and target are not compatible (`C` cannot terminate before `G`). Finally, the feedback flow from `B` to `C` is structurally consistent (e.g., it is balanced by the feedback flow from `A` to `D`), but behaviorally inconsistent: `B` cannot raise feedback even before it has started execution[3].

The following requirements have to be met by the process management tool to be constructed: The user is supplied with a graphical view of the task net which signals all inconsistencies. *Edit operations* are offered to build up and modify task nets by creating/deleting tasks and relationships and by changing task names. *Execution operations* are used to perform state transitions (`Start`, `Suspend`, etc.). With respect to constraint checking, the tool has to provide two working modes: In *enforcing mode*, the task net must not contain any inconsistency, and each command violating a static or dynamic constraint is rejected. In *permissive mode*, constraint violations are tolerated and marked (as shown in the screenshot above). The markings are updated after each command to provide immediate feedback to the user. Dynamic constraints are simply ignored in this mode.

In this case study, the commands for editing and execution perform rather simple transformations (insertion/deletion of tasks and relationships, changes of attribute values). The main challenge lies in the validation of constraints, which can be realized in different ways: In the case of *global validation*, constraints are checked on the whole graph (representing a task net). Since the user has to be provided with feedback on each command, global validation causes performance problems in the case of large graphs. In contrast, *incremental validation* checks only those parts of the graph which are affected by a change. The requirements of our case study call for incremental rather than global validation.

## 3 Models

In this section, we present alternative models used for the development of a process management tool based on dynamic task nets. Please recall that we are concerned with the model only and ignore the view and controller part of the application. The models are evaluated and compared against each other in Sect. 4.

[2] In gray-scale reproduction, dark boxes indicate inconsistent tasks, but inconsistent relationships are hard to identify.

[3] Feedback flows are inserted only on demand.

## 3.1 GMF/EMF

The Eclipse *Graphical Modeling Framework* (GMF [13]) supports the generation of a graphical editor for a custom model. The model is defined with the help of the *Eclipse Modeling Framework (EMF)*. This way each model is based on the *Ecore* (EMOF) meta model (a UML dialect and a variant of the OMG proposal for *Essential Meta Object Facility*[14]). Various ways exist to define an Ecore model: UML class diagrams, Java interfaces or directly via Ecore-XML (analogous to XMI). Please note that this paper deals only with the semantic model and not with the notational model, which is also required for building a graphical tool with GMF.

Figure 3 shows a UML diagram of the dynamic task net model. Each instance of `DTNDynamicTaskNet` consist both of `DTNConnection` and `DTNTask` objects. Each `DTNConnection` has one source and one target `DTNTask` object. `DTNConnection` is specialized to distinguish between the three types of connections in our dynamic task net case study: `DTNSubtaskFlow`, `DTNFeedbackFlow` and `DTNControlFlow`.

To deal with constraints, GMF supports *audit rules* which are based on OCL 2.0 [15]. Each *static constraint* of a task net (see Sect. 2) is defined by a corresponding audit rule. Figure 4 shows the OCL statement of the audit rule for detecting control flow cycles. First we select the `targetTask` of the connection for which the constraint is evaluated (`self`). Then we select the set that is reached via transitive closure of all tasks that can be reached via a connection (`outgoingEdges`) of type control flow (`oclIsTypeOf(DTNControlFlow)`). This set must not contain the source task of the connection (`excludes(self.sourceTask)`) – or the connection is part of a cycle. The closure operator, which is not included in the OCL standard and has been added in EMF as an extension, is indispensable for declaratively specifying some of the constraints defined for dynamic task nets. Note that the rule is declared as invariant, i.e. it is true iff the task net is cycle free.

In addition, GMF partially supports the specification of *dynamic constraints*. OCL constraints may be defined as preconditions of commands for creating relationships; thus, they are called *link constraints* in GMF. For example, there is a link constraint which forbids the insertion of an incoming control flow of a terminated task. For other

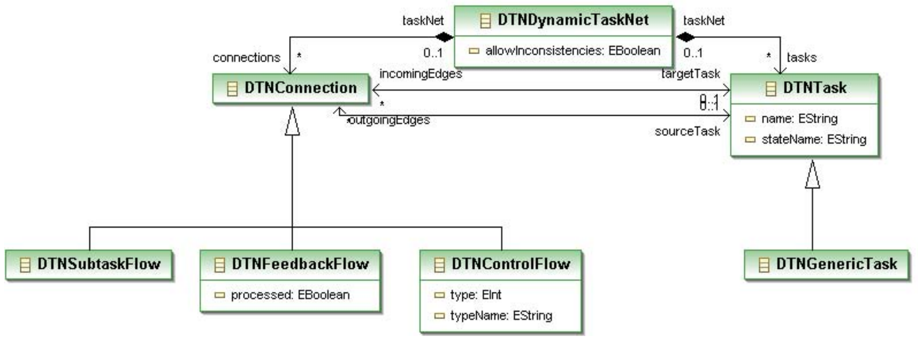


**Fig. 3.** EMF class diagram of the DTN-Model

```
self.targetTask
  ->closure(t|t.outgoingEdges
  ->select(e|e.oclIsTypeOf(DTNControlFlow)).targetTask)
  ->excludes(self.sourceTask)
```

**Fig. 4.** OCL expression for checking control flow cycles

types of commands, we wrote Java code for checking dynamic constraints (e.g., in order to preserve the history of task executions, a terminated task must not be deleted).

GMF supports the following mechanisms for validation: Audit rules for static constraints may be declared for *batch validation*, which has to be invoked explicitly by the user. All of these rules are checked on the complete model instance, and model elements are marked with constraint violations. In between two batch validations, rules declared for batch validation are not checked, and the markings are not updated.

Audit rules for static constraints may also be declared for *live validation*. These rules are checked immediately after each command execution; when some constraint violation is detected, the command is rolled back. Please note that live validation operates incrementally. Finally, link constraints are checked as preconditions before a command is executed. When a link constraint is violated, the respective command cannot be executed.

It is important to note that the validation mechanisms offered by GMF do not adequately support the validation modes required for our case study (see end of Sect. 2). We used batch validation for partially realizing the *permissive mode*. However, batch validation does not provide immediate feedback, and if it did, it would not provide fast responses when working on large model instances (due to global rather than incremental validation). Furthermore, live validation and link constraints cannot be used because constraint violations are not tolerated.

Likewise, we realized the *enforcing mode* only partially with live validation and link constraints. The audit rules for live validation were created by copying and modifying the rules for batch validation (the modifications are necessary to ensure that the rules are evaluated only when the enforcing mode is active). Unfortunately, live validation works incrementally, but not correctly: We would have had to customize the live validation by hand (by writing Java code) to make sure that all constraints on model elements affected by a change are actually re-evaluated. Furthermore, we defined link constraints in OCL for those commands which insert relationships, but we had to write Java code for those dynamic constraints which apply to other kinds of commands.

To conclude this subsection, let us briefly discuss how we realized the state machine of Fig. 2. Unfortunately, EMF does not provide modeling support for state machines. To improve maintainability (design for change), we applied the *state pattern* [16] and implemented state transitions in *Java*.

## 3.2 PROGRES

*PROGRES* [1], a specification language for programmed graph rewriting systems, supports a wide variety of language features for defining classes of attributed graphs, consisting of typed and attributed nodes which are connected by directed, binary relationships (edges) without attributes. Language features include multiple inheritance

```
node class + CONTROL_FLOW is a TASK_RELATIONSHIP
   ...
  derived
    BalancedControlFlow : boolean
      = card ( self.(      (    -ToSource->
                              &  <=Contains=
                              &  <-ToSource-
                              &  instance of CONTROL_FLOW )
                      and  (    -ToTarget->
                              &  <=Contains=
                              &  <-ToTarget-
                              &  instance of CONTROL_FLOW ) ) ) >= 1;
  ...
end;
```

**Fig. 5.** Textual specification of node class CONTROL_FLOW

on node classes, a stratified type system (nodes are instances of node types which are in turn instances of node classes), definition of derived attributes and relationships (the latter of which are called paths), graph transformation rules with flexible graph patterns, and control structures supporting non-determinism and transactional behavior. Some specification elements such as derived attributes and relationships may be specified both textually and graphically.

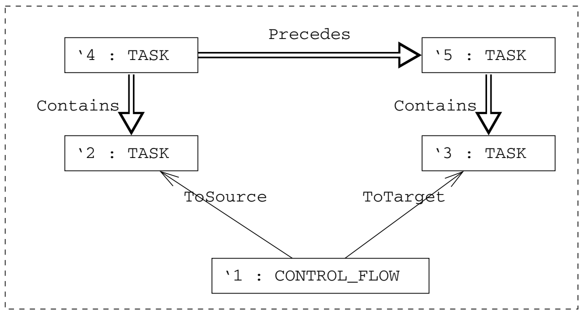
In our case study, constraint checking plays a crucial role. In the specifications we prepared for the case study, constraint checking is realized with the help of *derived attributes*. The user of the PROGRES language may define constraints in a declarative way with the help of equations. The underlying runtime system, including the database management system GRAS [17], provides for incremental evaluation of derived attributes. Thus, the user of the PROGRES language does not have to take care of the maintenance of the values of derived attributes.

The textual definition of derived attributes is illustrated in Fig. 5, which shows an excerpt of the declaration of the node class CONTROL_FLOW. The derived attribute BalancedControlFlow is used to check whether a control flow is balanced, and is defined by a textual expression in a similar way as an OCL constraint. Starting from the current flow, a navigation is performed to the source, the parent, and to its adjacent control flows on the next layer upward the task hierarchy; likewise for the target. The resulting sets of control flows are intersected. The control flow is balanced if the cardinality of the intersection is greater than 0, i.e., the intersection set is not empty.

Alternatively, constraints may be defined graphically rather than textually. In particular, a rule for a derived attribute may refer to a graphical *restriction*, i.e., a unary relation on nodes of a certain class. A node meets the restriction if it is part of a graph pattern defined in the body of the restriction. Figure 6 shows a graphical restriction for the balancing of control flows. We consider the graphical restriction easier to read than the corresponding textual expression shown in Fig. 5. Since textual and graphical notations are both offered by PROGRES, the user may select the notation which is more appropriate for the problem at hand.

In the specifications of the case study, we separated graph transformations from constraint checking. An example is given in Fig. 7. Tasks and their relationships form an overall process which is represented by a node of class PROCESS. All elements of some process are connected to the process node by Has edges (declared outside the

```
restriction + BalancedControlFlowRestriction : CONTROL_FLOW =
  `1 in
```



```
end;
```

**Fig. 6.** Graphical specification of control flow balancing

```
node class + PROCESS is a NODE
  intrinsic
    InconsistenciesAllowed : boolean := true;
  redef derived
    Consistent = for all element := self.Has ::
                    element.Consistent
                 end                           ;
  methods
    ...
    transformation + EditCreateControlFlow
      ( sourceTask, targetTask : TASK ;
        controlFlowType : type in CONTROL_FLOW ;
        out taskRelationship : controlFlowType)
    =
        self.CheckPreconditionOfEditOperation ( targetTask )
      & self.AuxCreateTaskRelationship
             ( sourceTask, targetTask, controlFlowType,
               out taskRelationship )
      & (self.InconsistenciesAllowed or self.Consistent)
    end
    ...
end
```

**Fig. 7.** Creation of a control flow

class `PROCESS`). The process node carries an intrinsic attribute for controlling whether inconsistencies are allowed, and a derived attribute which evaluates to true when all elements are consistent. The derived attributes attached to the process elements refer to other derived attributes such as e.g. `BalancedControlFlow` (Fig. 5).

All operations for creating or deleting process elements are attached as methods to the node class `PROCESS`. For example, when a control flow is created, the graph transformation rule which actually inserts the control flow is embraced by actions dedicated to checking constraints. The transformation `EditCreateControlFlow` is an atomic transaction, i.e., either all of its steps succeed, or it fails and leaves the host graph unchanged. `CheckPreConditionOfEditOperation` checks a dynamic precondition: The target task of the control flow would be affected by this edit operation. If inconsistencies are not allowed and the target task has already terminated, the check fails, and the transaction is rolled back. Please note that this check cannot be postponed: It cannot be recognized after the fact that the in-context of a terminated task was modified after termination. In the next step, the control flow is created by a graph transformation rule which simply creates the control flow node and its adjacent edges without performing any further constraint checking. Finally, after the control flow has been inserted, it is checked whether inconsistencies are allowed. If this is not the case, it has to be checked whether any inconsistencies have been introduced into the process. Please note that access to the derived attribute triggers all necessary re-evaluations at runtime. If the overall process is no longer consistent, the check fails, and the transaction is rolled back.

The PROGRES specification meets all of the requirements imposed by the case study. In particular, it realizes both the permissive and the enforcing mode with incremental validation. From the specification, executable code is generated which is hooked into the UPGRADE framework [18] to produce a graphical tool for software process management. The screen shot of Fig. 1 was taken from this tool.

So far, we have not discussed how we realized the state machine of Fig. 2. Unfortunately, PROGRES does not provide modeling support for state machines. We simply added a state attribute to the `TASK` and wrote methods for the state transitions which check their preconditions (legal source state) and invariants (compatibility with states of neighbor tasks).

## 3.3 Fujaba

Fujaba [19] is an environment for developing executable models with the help of class, story, and state diagrams. It is being developed jointly at multiple sites and has been used in numerous research projects. Fujaba strongly supports graphical modeling, while PROGRES offers a mix of graphical and textual modeling elements. Fujaba's most distinctive feature are the so called story diagrams, a combination of activity and communication diagrams, from which Fujaba is able to generate executable code. We used the CASE tool Fujaba in our case study to design and implement the application logic of our process management tool. Fujaba has been integrated into various user interface tool kits such as GEF, GMF, and UPGRADE, but user interface issues go beyond the scope of this paper.

While Fujaba does not support OCL constraints, constraints may be expressed graphically by *story patterns* with embedded path expressions. With the help of story patterns, constraints may be written in an intuitive way; in some cases, they are much easier to understand than OCL constraints. An important difference to the OCL constraints as supported in GMF/EMF consists in the use of story patterns: In Fujaba, story patterns are embedded in story diagrams and thus belong to the dynamic rather than the static model.

In the case of Fujaba, we fully realized both the permissive and the enforcing mode of operation required for the process management tool. We prepared two versions of the Fujaba model: The first one performs incremental validation, the second one resorts to global validation. In contrast to PROGRES, Fujaba does not support incremental re-evaluation of derived data. Thus, the user of Fujaba must explicitly program incremental validation. The additional modeling effort can be determined by comparing the model versions with incremental and global validation, respectively.

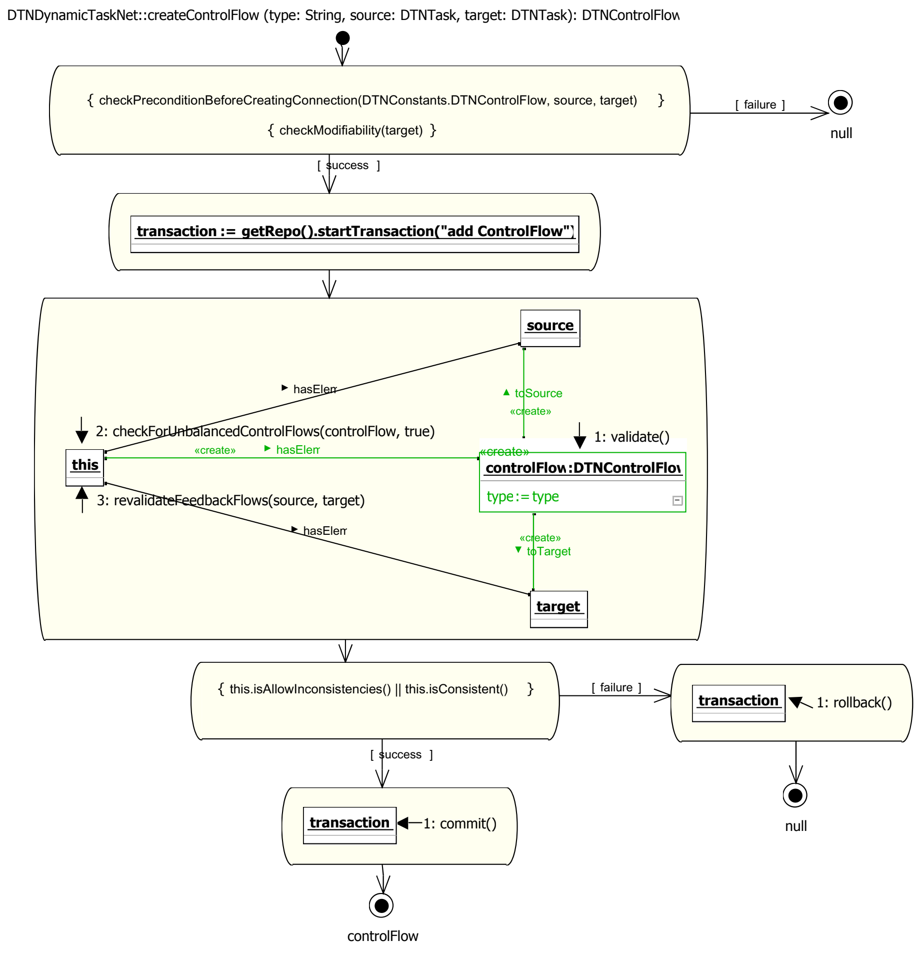

DTNDynamicTaskNet::createControlFlow (type: String, source: DTNTask, target: DTNTask): DTNControlFlow
{ checkPreconditionBeforeCreatingConnection(DTNConstants.DTNControlFlow, source, target) }
{ checkModifiability(target) }
[ failure ]
null
[ success ]
transaction := getRepo().startTransaction("add ControlFlow")
source
hasElem
toSource
«create»
2: checkForUnbalancedControlFlows(controlFlow, true)
1: validate()
this
«create»
hasElem
controlFlow:DTNControlFlow
type:=type
3: revalidateFeedbackFlows(source, target)
hasElem
«create»
toTarget
target
{ this.isAllowInconsistencies() || this.isConsistent() }
[ failure ]
transaction
1: rollback()
null
[ success ]
transaction
1: commit()
controlFlow

**Fig. 8.** Story diagram for creating a control flow

Figure 8 shows the story diagram for creating a control flow in the case of *incremental validation*. The story diagram is structured in a similar way as the corresponding PROGRES transaction (see Fig. 7). First, it is checked whether insertion of the control flow would result in a duplicate relationship and whether the dynamic constraint of this operation is violated (the in-context of a terminated task must not be modified). Next, a transaction is started, making use of the Coobra repository services (in PROGRES, the compiler inserts this step automatically due to the transactional semantics of programmed transformations). The story pattern following the start of the transaction inserts the control flow and triggers the required re-validations. Subsequently, it is checked whether inconsistencies have been introduced in enforcing mode. In this case, the transaction is rolled back; otherwise, it is committed.

In the case of incremental validation, it has to be decided for each change which constraints on which graph elements have to be re-evaluated. The story pattern for

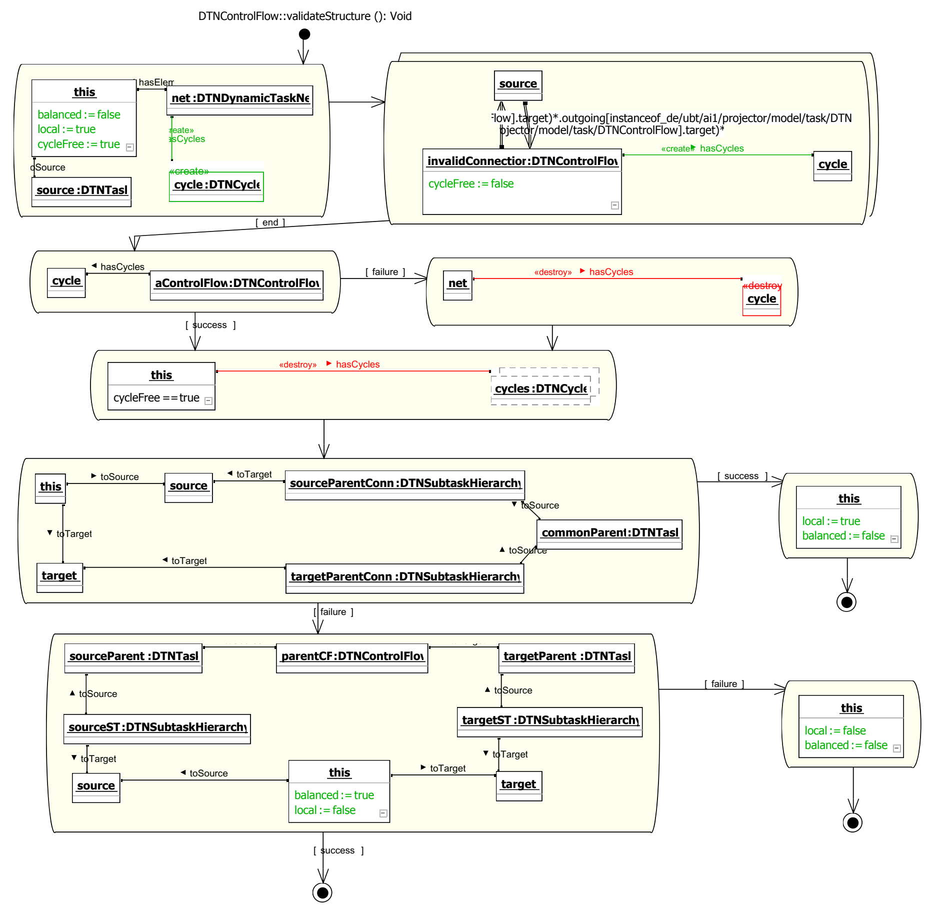


**Fig. 9.** Story diagram for validating a control flow

creating a control flow triggers validation of all constraints for the new control flow (1). In addition, it has to be checked whether any previously unbalanced control flows are balanced by the new control flow (2). Likewise, feedback flows which did not have a path of opposite control flows have to be re-validated (3). This control logic makes the model for incremental validation both larger (model size) and more difficult to program (modeling effort) than the model for global validation.

For the case of incremental validation, Fig. 9 shows the story diagram for validating static structural constraints for a control flow. The story patterns at the bottom check whether the control flow is local or balanced; they are the same as for global validation. However, the check for cyclic control flows is more complicated than for global validation since it requires the maintenance of auxiliary data structures for efficient re-validation: Control flows which are part of a cycle are attached to a cycle object. When a control flow is inserted, the cycle check is invoked on the new control flow. A cycle object is created tentatively, and control flows on a cycle are attached to the new cycle object. If no cycle has been found, the cycle object is removed again. Otherwise, it persists in the graph, and the `cycleFree` attributes of all control flows on the cycle are set to `false`. When a control flow is deleted, the same check is invoked on all control flows of all cycles to which the deleted control flow belongs. If a control flow turns out to be cycle free, it is removed from all cycles it was attached to.

The Fujaba model for *global validation* is structured similarly, but it is smaller and simpler to program. After each change, a global validation is trigged for all constraints on all graph elements — independently of the type of the change and the elements to which the change has been applied. Here, the cycle check does not require an auxiliary data structure: Control flows located on a cycle are marked by setting their `cycleFree` attribute to `false`. However, this brute force method of validation runs into performance problems when applied to large graphs.

To conclude this subsection, let us briefly discuss the use of Fujaba‘s *state diagrams*, which are not available in GMF/EMF and PROGRES. In a previous version of the Fujaba model for dynamic task nets [20], we mapped the state diagram of Fig. 2 onto a Fujaba statechart. However, Fujaba supports behavioral rather than protocol state machines. In dynamic task nets, state machines are used to define the life cycle of tasks from creation to termination. Thus, state machines describe only in which state some operation may be invoked and which target state is reached after the operation has completed. In contrast, behavioral state machines as supported by Fujaba describe which events an object may receive in which state, which actions are performed in response to an event, and which operations are performed while the object resides in a certain state. The underlying programming model takes care of concurrency (an inherent feature of statecharts) and deals with sending/receiving of events, event queues, etc. This stands in contrast to ordinary sequential programming based on method invocations. Since the Fujaba state machines did not match our intents, we stopped using them and decided to implement the state machine ourselves using the state pattern as described in [16] in a similar way as in the GMF/EMF solution.

## 4 Evaluation

Below, the modeling approaches applied in the case study are evaluated with respect to the language features offered, expressiveness of the modeling language, model size, readability, modeling effort, and efficiency. In the latter category, we are interested only in the support for incremental validation — which is crucial for interactive tools with immediate constraint checking. Please note that the evaluation is performed with respect to the case study only. Thus, the findings can be applied only to applications of the same profile as the case study.

Table 1 attempts to collect information on the *model sizes* in terms of the number of model elements classified into different categories. The PROGRES column refers to the specification where derived attributes are defined textually rather than graphically. Furthermore, the Fujaba numbers refer to the models realizing incremental and global validation, respectively (i/g). When a category is not applicable, the table contains the entry “–”. Constraints refer to OCL constraints in GMF/EMF, and to evaluation rules for derived attributes in PROGRES. For GMF/EMF, we counted the Java methods which

**Table 1.** Model size (number of model elements)

| | GMF/EMF | PROGRES | Fujaba (i/g) |
|---|---|---|---|
| Classes | 15 | 12 | 18/16 |
| Attributes | 7 | 5 | 21/19 |
| Associations | 5 | 6 | 19/16 |
| Inheritance rel. | 10 | 11 | 17/15 |
| Constraints | 24 | 15 | –/– |
| Methods | 8 | 34 | 31/28 |
| Control structures | 13 | 35 | 36/30 |
| Graph transformation rules | – | 5 | 90/66 |

were required for implementing the model. For Fujaba, this category refers to story diagrams. In PROGRES, we counted both transactions and functions. In the last row, we counted graph transformation rules in PROGRES and story diagrams in Fujaba (even if they merely describe a graph test rather than a graph transformation). Elementary story patterns (containing one object only) were not included in the numbers.

## 4.1 GMF/EMF

**Language features.** GMF/EMF supports the rapid generation of graphical editors from class diagrams and OCL constraints.

**Expressiveness.** Class diagrams and extended OCL constraints are powerful means for the static model. The dynamic model is not supported at all. In the case study, this restriction was not a severe problem, but it did require to write some Java code.

**Model size.** The model consists of two small class diagrams, and OCL constraints covering a few pages. In addition, we had to write 8 Java methods covering about 2 pages of source code. Thus, the size of the model is pretty small.

**Readability.** Class diagrams are widely accepted for the static model. OCL constraints tend to be hard to read and write as soon as complex structural conditions need to be expressed.

**Modeling effort.** The modeling effort is low as far as it concerns the static model being defined by class diagrams and OCL constraints.

**Incremental validation.** Basic support for incremental validation is provided, but the set of elements to be re-evaluated is not determined correctly in some cases.

## 4.2 PROGRES

**Language features.** PROGRES offers a wide variety of language concepts, but the language does not support state machines. In the case study, we made extensive use of derived attributes and incremental attribute evaluation for checking constraints. Graph transformation rules do not play a dominant role in the case study. Transactions are primarily used for wrapping graph transformations with consistency checks.

**Expressiveness.** As far as the static model is concerned, PROGRES and GMF/EMF are comparable with respect to expressiveness. PROGRES provides comprehensive and high-level support for specifying graph transformations, but the capabilities of PROGRES have not been exploited fully in the case study (only five graph transformation rules were required in the case study).

**Model size.** The model is larger than the GMF/EMF model, but moderate in size. The variant of the specification where we used textual notation for constraints comprises 13 pages (printed in 10 pt font). The increase of model size compared to GMF/EMF is primarily due to the fact that all operations (create/delete tasks and relationships, perform state transitions, etc.) have to be specified explicitly (while basic operations are generated automatically in GMF/EMF). This explains the number of methods (transactions and functions) plus graph transformation rules (about 40 altogether).

**Readability.** PROGRES specifications are rather difficult to read (and write) for two reasons: First, PROGRES does not use standard notation the user may be familiar with anyway. Second, the language is complex and offers lots of language constructs. Apart from that, the readability depends on the style in which the specification is written. In particular, we consider graphical notation for constraints easier to read than textual notation in most cases.

**Modeling effort.** For an experienced user of PROGRES, the modeling effort is moderate. It is possible to write specifications at a high level of abstraction without dealing with operational issues such as pattern matching, consistency maintenance, and rollback of failing transactions.

**Incremental validation.** PROGRES supports incremental evaluation of derived attributes and relationships.

### 4.3 Fujaba

**Language features.** In Fujaba, models are defined in terms of class diagrams, state diagrams, and story diagrams. Fujaba strongly supports graphical modeling and uses textual notation only to a limited extent (e.g., in path expressions).

**Expressiveness.** By and large, class diagrams, story diagrams, and state diagrams are powerful means for graphical modeling. However, the state diagrams provided by Fujaba were not adequate for our case study. Furthermore, constraint checking has to be performed in a procedural way. The Fujaba models are less declarative than their counterparts.

**Model size.** The Fujaba model is much larger than the PROGRES model. As in the case of PROGRES, all operations have to be modeled explicitly, while basic operations are generated in GMF/EMF. Constraint checking requires a lot of story patterns, even in the case of global validation.

**Readability.** Fujaba uses intuitive graphical notation. Therefore, Fujaba models are quite easy to read — as long as they remain small enough. In particular, story diagrams should be decomposed into methods of manageable size and complexity. Otherwise, the reader may easily lose orientation.

**Modeling effort.** Among the approaches investigated in the case study, Fujaba required the highest effort of modeling. This is due to the size of the model and the handling of algorithmic aspects (which in particular applies to the model for incremental validation).

**Incremental validation.** Since Fujaba does not support incremental evaluation of derived data, the respective algorithms have to be designed by the Fujaba user for each application anew.

## 5 Conclusion

In our case study, we have compared GMF/EMF against Fujaba and PROGRES, which are both based on graph transformations, with respect to language features, expressiveness, model size, readability, modeling effort, and efficiency. Since the requirements imposed by our case study match fairly well the support offered by GMF/EMF, a process management tool was developed with the help of GMF/EMF rapidly with small modeling effort. PROGRES is able to compete with GMF/EMF and adds incremental attribute evaluation as a distinctive feature. Finally, the Fujaba model is larger and more procedural than its competitors.

The modeling support by GMF/EMF is confined to graphical editors with basic commands. Further extensions of the case study— e.g., process patterns or data flows— would go beyond the modeling support of GMF/EMF. Please note that building a full-fledged process management system considerably goes beyond building a simple graphical editor. A modeling language like Fujaba provides much more comprehensive modeling support in a single language, but requires further improvements, e.g. with respect to constraint checking and state diagrams. We hope that this case study provides some useful hints and suggestions for further improvements. Graph transformation rules do not play a dominant role in this case study, but they constitute an important building block of a language for model-driven development.

## References

1. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages, and Tools, vol. 2, pp. 487–550. World Scientific, Singapore (1999)
2. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A.: Tool integration at the meta-model level: the Fujaba approach. International Journal on Software Tools for Technology Transfer 6(3), 203–218 (2004)
3. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
4. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: [10], pp. 446–453
5. Bardohl, R., Ermel, C., Weinhold, I.: GenGED - A visual definition tool for visual modeling environments. In: [10], pp. 413–419
6. Minas, M., Köth, O.: Generating diagram editors with DiaGen. In: [9], pp. 433–440

7. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - visual automated transformations for formal verification and validation of UML models. In: 17th IEEE International Conference on Automated Software Engineering (ASE 2002), pp. 267–270. IEEE Press, Los Alamitos (2002)
8. Agrawal, A.: Graph rewriting and transformation (GReAT): A solution for the model integrated computing (MIC) bottleneck. In: 18th IEEE International Conference on Automated Software Engineering (ASE 2003), pp. 364–368. IEEE Press, Los Alamitos (2003)
9. Münch, M., Nagl, M. (eds.): AGTIVE 1999. LNCS, vol. 1779. Springer, Heidelberg (2000)
10. Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.): AGTIVE 2003. LNCS, vol. 3062. Springer, Heidelberg (2004)
11. Heimann, P., Joeris, G., Krapp, C.A., Westfechtel, B.: Graph-based software process management. Journal of Software Engineering and Knowledge Engineering 7(4), 431–455 (1997)
12. Krapp, C.A.: An Adaptable Environment for the Management of Development Processes. Aachener Beiträge zur Informatik, vol. 22. Augustinus Buchhandlung, Aachen, Germany (1998)
13. Eclipse Foundation: GMF - Graphical Modeling Framework (2006) (last visited, 21/03/2007), `http://www.eclipse.org/gmf`
14. Eclipse Foundation: The Eclipse Modeling Framework (EMF) Overview (2005) (last visited, 27/10/2006),
`http://dev.eclipse.org/viewcvs/indextools.cgi/checkout/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html`
15. Warmer, J., Kleppe, A.: The Object Constraint Language, 2nd edn. Addison Wesley, Boston (2003)
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading (1994)
17. Kiesel, N., Schürr, A., Westfechtel, B.: GRAS: a graph-oriented software engineering database system. Information Systems 20(1), 21–51 (1995)
18. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: A framework for building graph-based interactive tools. Electronic Notes in Theoretical Computer Science 72(2), 113–123 (2002)
19. Zündorf, A.: Rigorous object oriented software development. Technical report, University of Paderborn, Germany (2001)
20. Buchmann, T., Dotor, A.: Building graphical editors with GEF and Fujaba. In: FUJABA Days 2006, Paderborn, Germany, University of Paderborn, pp. 47–51 (2006)