Graph-Based Tools for Distributed Cooperation in Dynamic Development Processes*

Markus Heller, Dirk Jäger

RWTH Aachen University, Department of Computer Science III,
Ahornstrasse 55, 52074 Aachen, GERMANY
{heller|jaeger}@cs.rwth-aachen.de

Abstract. The highly dynamic character of development processes makes it a challenging task to provide support for the management of such processes within an organization. The process management system AHEAD addresses the specific problems related to the management of development processes in engineering disciplines. The system stores all management data as graphs. The application logic is specified in a formal specification based on a programmed graph rewriting system. From this specification several management tools of the AHEAD system are generated. Recently, the AHEAD system has been extended to support distributed development processes. Two or more organizations use their own instances of AHEAD and these instances are coupled at run-time. The coupling logic is specified by graph transformations and the executable code for the coupling can be automatically generated from this specification. Furthermore, the precise notation of the coupling by a formal specification makes it easy to enhance or extend the coupling mechanism. This paper describes how graph transformations are used to realize the demanded functionality.

1 Introduction

The study of development processes in our group focuses on managing the development of a complex end-product in engineering disciplines like software or chemical engineering. Development processes tend to be highly creative and dynamic. For example, it may be difficult to predict all activities, their order, and their duration. Changes in the product specifications may induce variations of planning the development activities. Moreover, development processes tend to be highly unique. As a result of this, the management of activities, performers and resulting products is rather complex.

Today the development of an end-product is not always accomplished by one organization alone, e.g. a company or department of a company. For example, the know-how needed to perform some development activity may only be available in another organization. In a distributed development process the development activities are performed by employees of different organizations working together while development products are exchanged across organizational boundaries.

* The authors gratefully acknowledge financial support of Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center 476 "Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik".

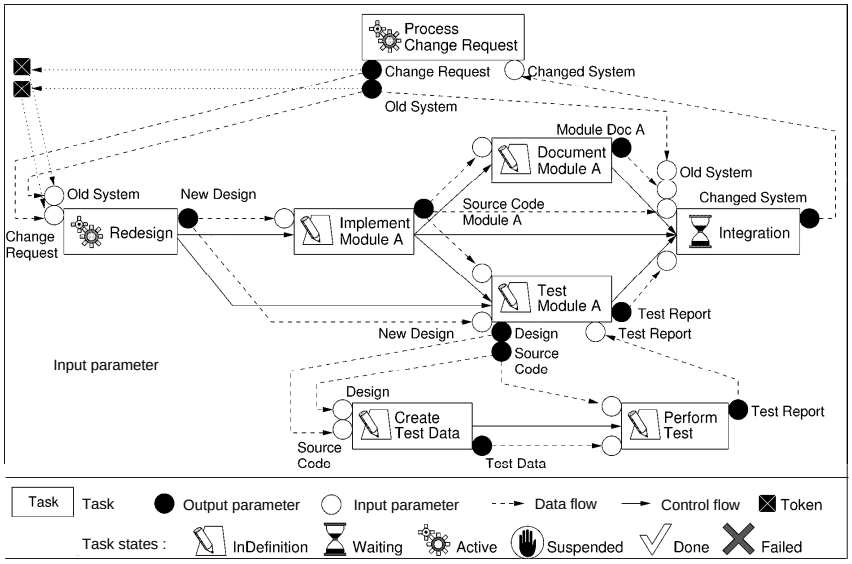

Fig. 1. Dynamic task net for a change request process in software engineering

The process management system AHEAD addresses these problems. AHEAD has been developed within a long-term running Collaborative Research Center IMPROVE (Information Technology Support for Collaborative and Distributed Design Processes in Chemical Engineering)[1]. In the supported scenario it is assumed that all participating organizations use the AHEAD system for their process management. The developed concepts and supporting tools are described in detail in [2, 3]. A demonstration scenario is described in [4].

2 The process management system AHEAD

2.1 Graph-based integrated management model

The management of a development process to develop a certain end product comprises the coordination of all development activities, the management of all related product data, e.g. technical documents and plans, and the management of the related resources.

The activities of a development process and the relationships between these activities can be modeled by a *dynamic task net*. Dynamic task nets are defined by the underlying model DYNAMITE[5]. They can be planned, executed, and analyzed in an integrated way.

Figure 1 shows an example of a dynamic task net. This task net resembles a specific change request process in software engineering. Such a process is executed in a software company during the development process for a new software system (and even after the release of the system to customers). A change request describes a change of the existing software system in order to fix a bug or to add some functionality to the system. The

first activity of this process can be a redesign of the current system, followed by the implementation, documentation, and test of all affected software modules. Finally, the changed modules have to be integrated into the software system.

The dynamic task net for this example is built of *tasks* representing activities, e.g. *Redesign*, *Implement Module A*. Each task has an *execution state* as 'InDefinition', 'Active', 'Suspended', or 'Finished'. Tasks can be connected with each other by directed edges representing *control flow* relationships defining the temporal execution order of tasks. Tasks are characterized by an *interface* of all available output and input products. To limit the possible types of input or output products, *input* and *output parameters* are introduced. An output parameter of a task is linked to a corresponding input parameter of another task by an edge denoting a *data flow* relationship. *Tokens* representing products can be passed between tasks along these data flows. Tasks can be either *atomic* or *complex*. Complex tasks can be further refined by task nets, e.g. the task *Test Module A* is refined by a task net with tasks *Create Test Data* and *Perform Test*. Thus dynamic task nets are hierarchical.

The product model COMA [6, 7] defines the representation of *products* (or *documents*) and the relationships between documents. Documents can be versioned. *Configurations* containing a set of product versions can be defined. The management of human and non-human resources is defined in the resource model RESMOD[8]. Plan resources, e.g. project teams and roles, and actual resources, e.g. organizational team units and team members, as well as a mapping between them can be represented.

The management models of AHEAD rely on graphs as the fundamental data structure. For example, a dynamic task net can naturally be represented as a graph of task nodes connected by edges denoting control or data flow relationships between different tasks. Operations on task nets (e.g. the insertion or deletion of tasks) are realized by graph transformations.

An example of such a graph structure is shown in figure 2. Different instances of the node classes `TASK` and `PARAMETER` as well as different instances of edge types for relationships between nodes are shown. Node classes are connected via `has`-edges with instances of the node types `Input` or `Output`.

The graph schema for dynamic task nets contains the common superclass of all model elements `ITEM` as root of the class hierarchy from which the node classes `ENTITY` and `RELATION` are derived. While `ENTITY` is used to represent entities like `TASK` and `PARAMETER`, the node class `RELATION` is introduced to model relationships between entities as, for example `has`.

In a similar way, the human and non-human resources and the product model are modeled using graphs and all operations on these models are realized as graph transformations. All management data for an instance of a development process are stored within a single graph containing all information about development tasks, product data, and resource data.

The models for dynamic task nets, resources, and products form the base model layer of the integrated management model of AHEAD. These base models are then integrated with each other pair-wise in a higher layer. The full integration of all three models is realized in a third layer above. A detailed discussion of this integrated management model can be found in [5].

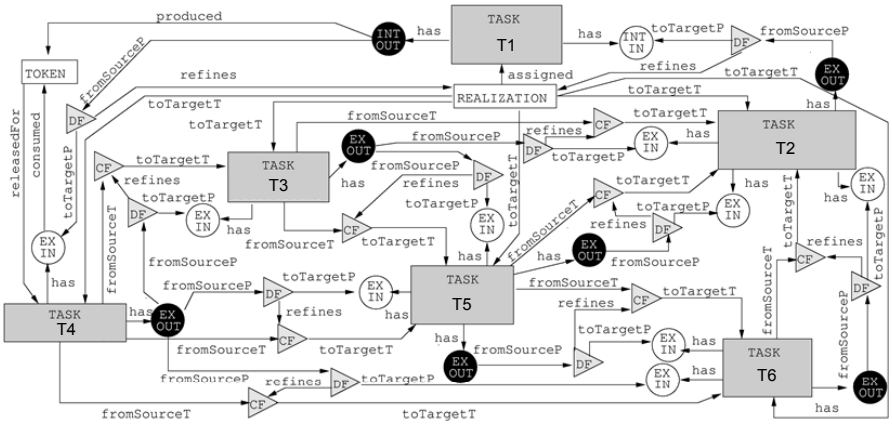

Fig. 2. Internal representation of dynamic task nets

The base models and the integration of these models are all combined in one single specification written in the language PROGRES[9]. PROGRES uses directed, attributed graphs with labeled nodes and edges as its fundamental data model. A PROGRES specification defines a graph schema and graph transformations. In a *graph schema* different node classes, node types and edge types can be defined. A node type is an instance of a node class. Attributes can be defined for node classes or types (not for edge types) to store additional information. *Graph transformations* are written in the form of *graph rewrite rules*. Each graph rewrite rule consists of a left-hand side, describing a matching graph pattern, and a right-hand side, with a subgraph for the replacement of the pattern on the left-hand side for each matching part of the graph. Further details can be found in [9].

2.2 System Architecture of the AHEAD system

Figure 3 gives an overview of the AHEAD system. Users of the AHEAD system are usually either *process managers* or *developers*. Process managers use the *management tool* to create and manipulate instances of dynamic task nets which represent development processes. The states of all activities of the process can be controlled, e.g. by starting or suspending activities. The managers may also define project teams (from human or non-human resources), manipulate the status of product data and assign team members to tasks. Developers use the *developer frontend* to get an agenda containing all tasks that are assigned to them. For every task, a work context can be opened showing detailed information about this task together with all associated input or output documents. There, developers can activate specific development tools working on the documents, too.

So all management data (task net, resource, and product data) are represented as graphs. Management tools and developer frontends have access to the management data, which are stored in the graph-based database GRAS[10]. The underlying base models are combined with each other in a PROGRES specification.

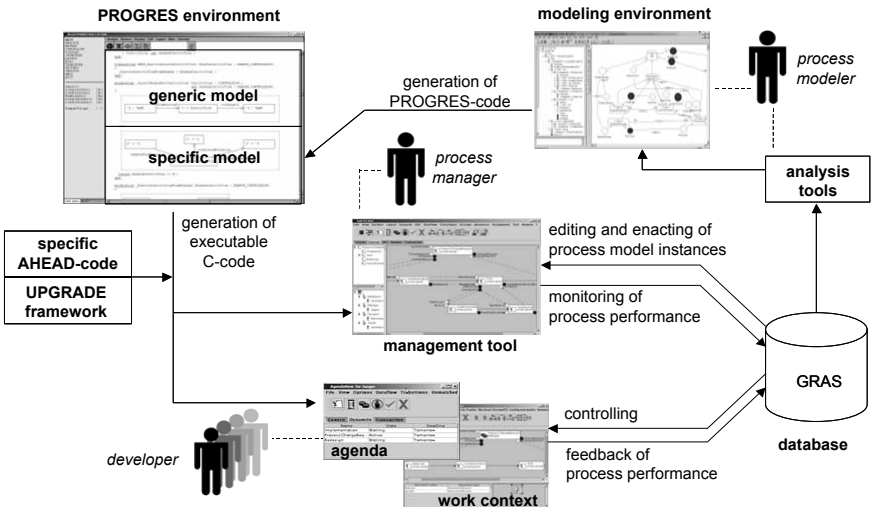

Fig. 3. Overview of the AHEAD system

The PROGRES *environment* provides a graphical editor for specifications and an interpreter to execute the graph code, generated from specifications as executable C-code. With the help of the framework UPGRADE[11] the specific AHEAD-code can be embedded in graphical user interfaces of the AHEAD system, e.g. the management tool or the developer frontend. With our graph-based tool machinery (GRAS, PROGRES, UPGRADE) tools can be generated by rapid prototyping, starting with a formal specification. Changes of the functionality of the generated prototype are done by changing the related parts of the graph schema and graph transformations in the specification, followed by an almost mechanical procedure to generate the new version of the prototype.

The AHEAD system can be adapted for different applications. Therefore, the specification is split into two parts. The first part of the specification, the *generic model*, contains all concepts which are independent of a special domain. This generic part is coded only once. The second part of the specification, the *specific model*, comprises all concepts (e.g. special types of tasks or special relationships between tasks) valid only in a specific application domain. For every new application domain a corresponding specific model can be defined. A *process modeler* does not need to use PROGRES for the definition of the specific part of the model directly, but he can rely on a *modeling environment* which allows to model the specific parts using the UML (Unified Modeling Language)[12]. The modeling environment is based on the commercial tool *Rational Rose*. From the model in UML a PROGRES specification can be generated. Additional analysis tools help to evaluate the management data and aid process modelers in defining domain-specific models [13].

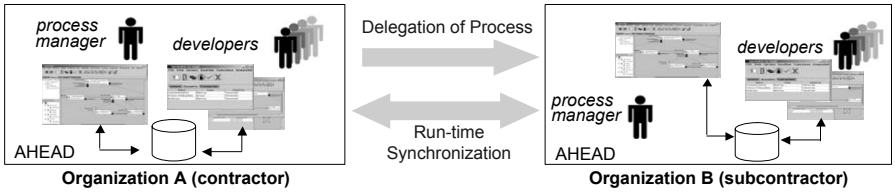

Fig. 4. Scenario of a distributed development process

3 Concept and tool support of distributed development processes

Often two organizations cooperate in *delegation relationship* where one organization (*contractor*) delegates a set of activities to another organization (*subcontractor*). The delegated activities have to be carried out by the subcontractor and some resulting products have to be returned to the contractor.

If each participating organization uses a process management system (each with its own database) to manage its development process, the question arises, how these management systems can be coupled to support a distributed development process. Up to now, it is assumed that both organizations use the AHEAD system with separate databases. Both AHEAD systems use the same graph-based PROGRES specification. This scenario is shown in figure 4 where organization A acts as a contractor and delegates a part of the overall process to organization B. Both organizations have to establish a run-time synchronization of their process management systems in order to inform each other about changes of those process parts which are executed on either side.

3.1 Requirements for an efficient support of distribution

In the AHEAD-project several important *requirements* for an efficient support of distributed development processes have been identified[3]:

1. The *contractor* must be able to *delegate* one or more related activities to be carried out by the *subcontractor*. The results of the delegated activities have to be returned to the contractor.
2. The contractor must be able to monitor the execution of the delegated activities (based on *milestones*).
3. The subcontractor must be able to refine the delegated activities.
4. Both sides must be able to work independently with their own management systems.
5. Both parties must be able to hide process details from each other.
6. Both parties understand every delegation of activities as a *contract*. The violation of this contract by the contractor or the subcontractor may have legal consequences.
7. The execution semantics of the delegated activities have to be preserved by the management systems used by both sides.

These requirements are chosen in order to balance the interests of organizations taking part in a distributed development process. For example, contractors are interested in

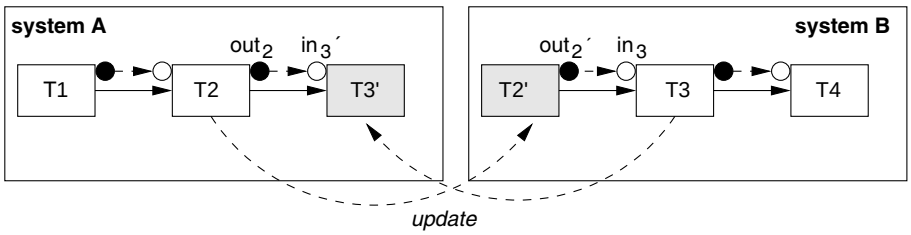

Fig. 5. Strategy for the distribution of a task net

gaining extensive control of the progress of delegated activities, while subcontractors aim to shield their internal process structure. Both interests tend to conflict with each other. A detailed discussion of these requirements is given in [2].

One extreme case in the bandwidth of possible concepts for a proper support of distributed development processes is the so called *white-box* approach where the contractor has full access to all data maintained by the subcontractor. This is typically the case when the subcontractor is forced to work on the contractor's database. In the other extreme, the subcontractor uses an independent instance of a database to which the contractor has no access (*black-box* approach). The proposed concept is right-hand in the middle of these extremes and, therefore, called a *grey-box* approach.

3.2 Strategy for the coupling of process management systems

Figure 5 shows a part of a development process which is distributed over two different process management systems A and B. The process consists of four tasks T1 - T4 which are linked by control flows. As T2 and T3 reside in different systems, the control flow between these tasks has to be realized in a different way. It is assumed that a command to start T3 was activated in system B. Due to the control flow between T2 and T3, the state of T2 has to be tested before T3 can start (for example, if the control flow is sequential, T2 has to be finished before activation of T3).

A possible strategy is to store duplicates of all tasks of system A, which have to be accessed from system B. For instance, in figure 5 a new task T2' (shown as a grey box) is created in system B as a *local copy* of the *master copy* T2 of A. From the perspective of system A, the task T2 is called a *monitored* task, as a local copy of it is stored in another system. A *private* task is neither monitored nor a (local) copy of a monitored task. In the example, T1 and T4 are private tasks. All changes to the state of the monitored task T2 are passed by *change messages* to system B, and system B updates its local copy of T2. Thus, both tasks T2 and T2' are kept consistent with respect to their state. Of course, the same strategy can be applied to duplicate copies of tasks of system B and maintaining them in system A, for instance, a duplicate task T3' is created for task T3 in figure 5. The definition of private, monitored tasks and local copies is always seen from the perspective of one of the two systems, which is called a *local* system. The other system then is a *coupled* system.

It can be decided within the system boundaries of system B, if a certain command, where a task of system A is involved, will be applicable or not, because all relevant in-

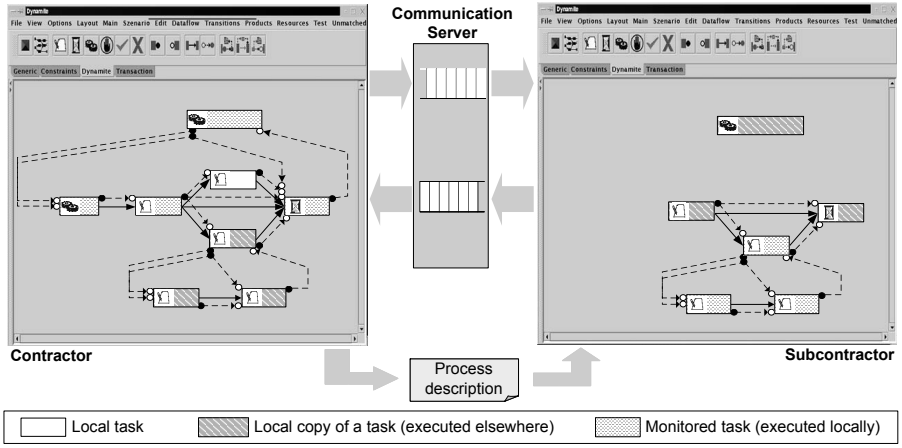

Fig. 6. Coupling of two instances of the AHEAD system

formation is stored within system B. Both systems do not need to be coupled when such a command is triggered. It is sufficient to ensure that every change message, sent by one of the systems, is delivered to the other system. If one of the two systems becomes unavailable, all change messages are temporarily stored in queues. Thus, both systems can work *autonomously*. With this idea of duplicate tasks, the relations between tasks in different systems can be handled in exactly the same way as relations between tasks residing in the same system. Concerning the execution of commands it now becomes *transparent* whether a task is private or not.

If a task is to be stored in two systems as described, it is necessary to store some context information for that task as well. The *context* of a task T contains all tasks that are directly connected to T, e.g. the parent task of T, all tasks connected to T by a control flow, all tasks connected to T by a data flows and the relevant output and input parameters, and all tasks in a refining task net of T. For example, in the lower part of figure 5 the context of task T2 consists of T1 and T3, together with all relevant parameters of these two tasks, and the control flow between T2 and T3.

3.3 Delegation of processes

The developed concept and tool support of distributed development processes in the AHEAD system is based on the delegation of process fragments[3]. Figure 6 provides an overview of the coupling of two instances of AHEAD:

1. *Preparation step*: The contractor selects a part of the task net for delegation within his own instance of the AHEAD system and exports this subprocess into a file. This file is sent to the subcontractor. The subcontractor imports the file into his own instance of AHEAD. A task net is set up there which corresponds to the delegated subprocess in the AHEAD system on the contractor side. The subcontractor's task net contains all delegated tasks together with their context elements. Copies thereof

remain in the system of the contractor and they are updated every time the corresponding tasks within the system of the subcontractor change their state.

2. *Run-time coupling*: The management systems of contractor and subcontractor are connected via a *communication server* at run-time to exchange change messages. With these messages the two management systems inform each other about changes of the state of the tasks maintained on either side. The contractor is informed about changes at delegated tasks, which are executed by the subcontractor, and vice versa. Each task can only be manipulated by exactly *one* of the two coupled AHEAD systems. For example, the subcontractor is responsible for the delegated tasks. He may also refine the delegated tasks with private task nets, which are hidden from the contractor. The contractor cannot manipulate the delegated tasks and can only monitor changes performed by the subcontractor. Conversely, the responsibility for the tasks of the context tasks remains with the contractor. These tasks can only be changed in their state by the contractor and these changes can only be monitored on the subcontractor side. If one of the two management systems is disconnected for a while and can no longer process incoming events, messages are stored in queues maintained by the communication server between the two AHEAD systems. After the system is connected again, queued events are processed.
3. *Manipulation of the delegated sub-process*: When the two systems are coupled at run-time, the delegated sub-process can be altered: for example, new tasks may be added to the delegated task net, input and output parameters may be attached to tasks, control flow and data flow relationships between tasks can be created, between local tasks as well as non-local tasks. Private tasks can be made visible for the partner by including them into the corresponding context. As contractor and subcontractor have to agree about these structural changes of the delegated process fragment, a *change protocol* is enforced by the AHEAD system. All structural changes have to be carried out by the contractor and are propagated to the subcontractor. The subcontractor can reject any of the propagated changes.

The AHEAD system offers a set of *remote link commands* for each of these phases. For example, there are commands for the export and import of process description files and for the refinement of delegated tasks. Contractor and subcontractor use special commands to register with the communication server. The largest set of commands deals with the structural manipulation of the delegated subprocess.

The described *delegation model* satisfies all of the requirements for a proper support of distributed development processes mentioned above. The idea to duplicate tasks in both management systems allows for the monitoring of duplicated tasks, thereby covering, together with the developed delegation model, requirements 1 to 4. The duplication of tasks for use in other systems can be forbidden (requirement 5). Management systems can work independent of each other, when messages are queued, e.g. in a *communication server* (also requirement 4). Requirement 6 is met by the change protocol for the manipulation of the delegated parts of a task net. Finally, both organizations use AHEAD for their process management and thus the execution semantics for tasks is the same in both systems (requirement 7).

4 Realization based on graph concepts

The graph-based realization of the proposed delegation concept is explained in this section. A couple of technical problems had to be solved: First, it had to be made clear how a delegated process fragment (a portion of a task net) is transferred between two coupled systems. Second, a mechanism to synchronize two coupled AHEAD systems had to be developed. Third, the remote link commands had to be specified.

4.1 Export and Import of delegated task nets

After a part of a task net has been selected for delegation in the AHEAD system of a contractor, it has to be transferred into the AHEAD system of the subcontractor. For this purpose an asynchronous method has been chosen: The delegated subnet and its context are exported to a file and then passed to the subcontractor, where it is being imported into his system. The XML-based language GXL (Graph eXchange Language)[14] has been selected for the implementation. A synchronous transfer, for example using a network connection between both systems, cannot be used here, because both systems have to work independently.

All information about delegation relationships is maintained within the graph structures of AHEAD. The existing AHEAD specification had to be extended to designate whether elements of the task net are “private”, “remote” or “monitored”, as well as the unique id’s of remote elements. These id’s refer to elements stored in a coupled system, and the local system receives them from the coupled system. They are used within changes messages to denote the corresponding remote elements.

4.2 Run-time synchronization of a delegated distributed task net

AHEAD stores all graph related data in the graph database GRAS. Unfortunately, GRAS has not been designed to access graphs in another instance of a GRAS system. Therefore, a JAVA-based *coupling module* has been realized to handle the technical details of the coupling, such as the connection of two instances of the AHEAD system and the execution of the remote link commands.

As GRAS is an *active database*, it generates a *database event* of a certain type when the graph is altered and propagates this event to all interested *event listeners*. For example, when a graph node of type `t` is inserted into the graph, a database event of type `NodeCreated(t)` is sent to all interested listeners. Similarly, changing the value of an attribute of a graph node leads to the generation of a corresponding modification event.

This kind of event handling is used to realize the delegation of tasks between two management systems. If a task T1, stored in process management system (PMS) A, is delegated to system B for execution, then a new copy T1’ of T1 is created in PMS B. Only for monitoring purposes the original task T1 remains in PMS A. When the state of T1’ in PMS B changes, a change message is sent back to PMS A to trigger appropriate steps to update the local copy T1 to the new state. The same applies for all task net elements, e.g. control flows, parameters, and data flows. Every instance of the process management system can at the same time act as a *producer* of events regarding all

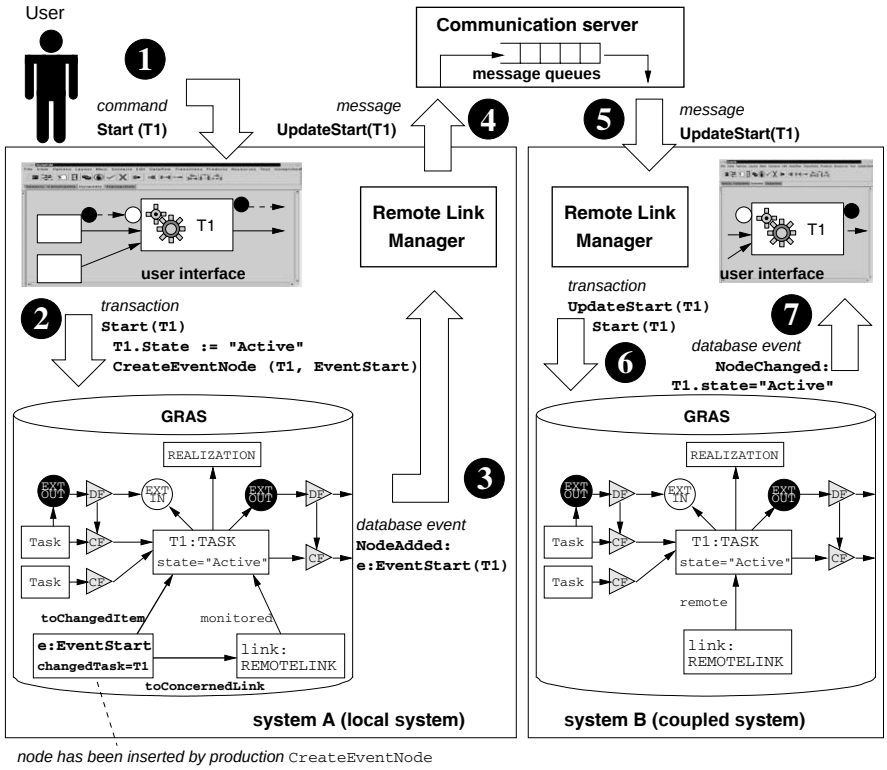

Fig. 7. Delegation of process fragments between two instances of the AHEAD system

elements which are monitored elsewhere and as a *consumer* of events regarding all elements which are executed elsewhere and only monitored locally.

This idea is now illustrated in the following example (see figure 7):

1. A user triggers a command “Start(T1)” at the graphical user interface of the (local) PMS A. Then a corresponding PROGRES transaction is executed in PMS A.
2. In this transaction it is tested whether the command is applicable in the current state of T1 or not. If so, the state of task T1 is changed to `Active`. If T1 is a task which is monitored elsewhere, a node of type `EventStart` is inserted into the graph of PMS A. This new event node is connected with T1 by an edge and T1 contains all data needed in a coupled system to update the maintained copy of T1 there (in the example, the identification of T1 is stored in `changedTask` as the only necessary information).
3. The insertion of the new event node raises a new database event `NodeAdded` generated by GRAS. This database event is propagated to all registered listeners of this event type.
4. The *Remote Link Manager* of PMS, listening to all events of such type, receives this event. The Remote Link Manager sends (via a *communication server* between the

```
node type EventAddConFlowNeutralDelegated : RELATION_EVENT
  derived
     taskID    = getNetworkID ( self.-toChangedItem-> );
     parentID  = getNetworkID ( self.-toChangedItem-> : TASK.=toParent=> );
     taskName  = self.-toChangedItem-> : TASK.Name );
     taskState = self.-toChangedItem-> : TASK.State );
     [...]
end;
```

Fig. 8. Event node type `EventAddConFlowNeutralDelegated`

two Remote Link Managers) a message `UpdateStart` together with the stored data of the event node to the Remote Link Manager of the coupled system. As stated before, the communication server can store such messages in case that one of the two systems is temporarily unavailable. The stored messages are processed when the system is available again. In the current prototypical implementation it is assumed that no change message is lost during transport from one system to another. This standard problem in distributed systems can be addressed by using a more reliable transport protocol. The messages can be numbered and the receipt of a message with a certain number can be signaled to the sender.

5. In PMS B the Remote Link Manager receives the message and executes an appropriate graph transaction `UpdateStart(T1)`. The parameters for this transaction are taken from the received message.
6. The execution of the transaction `UpdateStart(T1)` first determines the ID of the copy of task T1 in system B. Second, the state of this task is changed to `Active`.

The data exchange mechanism between the systems using the Remote Link Manager and the communication server is realized in JAVA, while the underlying logic of this coupling mechanism is specified in PROGRES.

4.3 Implementation of the *remote link commands* in PROGRES

As mentioned earlier, every event node type stores all needed data to update the state of the local copy of the task in a coupled system. PROGRES offers *derived attributes* for these node types. These attributes are not set to a specific value. Rather, it is specified which graph data have to be evaluated to determine the correct value of the attribute. By this mechanism it is possible to calculate the data of the event node types automatically in the instance the attribute is queried.

Figure 8 shows the corresponding specification of an event node type `EventAddConFlowNeutralDelegated`. This event is raised, when a control flow between two tasks is inserted while one of the two connected tasks, which has not been part of the delegated task net before, becomes a context task. As this task has not been instantiated in the coupled system before, a new context task has to be created in the coupled system, before the new control flow can be inserted between the two tasks. The corresponding event node type aggregates all data needed to perform the two described steps. To obtain the `taskID`, the evaluation starts at the event node (`self`) and if possible, navigates over an edge of type `toChangedItem` to the changed model entity. Next, the function

`getNetworkID` is called with this entity as an argument to obtain a network-wide ID that can be used in a coupled system to access this entity in the local system. The name of the changed task is retrieved in a similar way. Here the changed entity, which is expected to have type `TASK`, is reached during the graph navigation and its attribute `Name` is retrieved.

In the local system the insertion of a control flow between two tasks is done within a specific graph transformation. If both connected tasks are located in the local system, only the control flow is inserted. But if one of the connected tasks is located in a coupled system, it is decided during a case analysis, which type of event node is inserted into the local system.

The insertion of such event nodes is detected by the Remote Link Manager of the local system and a corresponding change message is sent to a coupled system. The Remote Link Manager of the coupled system then executes an update transaction. Every update transaction corresponds to exactly one event type, e. g. `EventAddConFlow-NeutralDelegated`, and specifies all necessary steps on the side of the coupled system to reach a synchronized state with the local system concerning this event.

This mechanism serves for the run-time synchronization of two AHEAD systems. In general, this mechanism can be used for the coupling of two graph rewriting systems operating on different (graph) databases. By this mechanism it is possible to execute remote transactions in one of the two rewriting systems triggered by the other system. For instance, within a graph transaction gt, executed in rewriting system A, the execution of a second graph transaction gt' in rewriting system B has to be triggered.

To achieve this, a new node of a special event node type et is created in system A. This event node can carry in its attributes some information of interest to system B in order to execute the remote transaction gt'. The new event node is inserted into the graph of system A. This leads to the generation of an event node of type `NodeAdded(et)`. A coupling module listens to all events regarding such event nodes. If the insertion of an event node of a specific type is detected, a corresponding message is sent to the coupling module of the other system B. The coupling module there by executing a predefined graph transaction gt', which can manipulate the graph of system B in a deliberate way.

5 Related work

Related work is shortly discussed on a conceptual level (delegation of processes) and on a technical level (coupling of two graph rewriting systems).

Van der Aalst gives an overview about concepts for distributed cooperation in [15]. Although the focus is on workflow management, the classification can also be used for the management of dynamic development processes. *Subcontracting* is similar to the delegation of process fragments proposed in this paper, but only allows to assign single activities of a process to another organization for execution. Delegation allows to assign whole task nets to another organization.

In [16] Dowson describes the use of contracts in the system IStar. There, the structure of a software development process is hierarchical and the responsibilities for all process parts are described by contracts. However, these contracts do not define how

the process is structured (black-box approach). Instead, every subcontractor is free to refine the assigned activities individually.

The *Contract Net Protocol (CNP)*[17] also deals with relationships between a contractor and a subcontractor. This work focuses on the fully automated negotiation of buying and selling of goods between systems, e.g. on electronic markets. A communication protocol with standardized messages is followed between an initiating agent and one or more participating agents who compete for an offered contract. The main focus lies on the automatic calling for and receiving of bids and awarding the contract to the most suitable bidding agent. This negotiation phase is not addressed in the AHEAD system. Here, both organizations have to come to a mutual agreement about details of the delegation relationship outside of AHEAD. After both organizations have come to an agreement, the AHEAD system handles the technical realization of the delegation of process parts and the coupling of both process management systems.

Taentzer et. al[18] use distributed graph transformations for the specification of static and dynamic aspects of distributed systems. There, a network structure consisting of local systems is modeled by a *network graph*. The internal state of each local system can also be described as a graph where nodes are data objects and edges between nodes express relationships between these data objects. *Local views* containing export and import *interfaces* are used to define the coupling of local systems within a network. The interfaces of a local system store a subgraph with all accessible nodes and edges. Coupling means connecting an export interface of one local system with an import interface of another. In contrast to this general specification approach, the coupling mechanism of this paper is developed with respect to the special case of coupled instances of the AHEAD system (based on PROGRES). The coupling of two AHEAD systems is the basis for the proposed concept of delegation of processes in the research area of process management. By a research prototype this delegation concept and the coupling of two AHEAD systems have been implemented.

6 Summary

In this paper a delegation-based concept supporting distributed development processes and its realization by the graph-based AHEAD system has been presented. In the scope of the IMPROVE project[1] this approach has been applied to a case study in chemical engineering, where the development of a chemical plant for polyamide is carried out in more than one organization.

The decision to use graph technology for the realization of the AHEAD system has proven to be a good basis for future extensions. The formal specification of the application logic of AHEAD gives a precise definition of the semantics of commands operating on the management data with AHEAD. The generation of executable code from this specification avoids the programming effort of implementing the commands in a traditional way. The application logic of AHEAD and the coupling logic are specified in PROGRES. As the coupling logic is based on the application logic of AHEAD, the same specification language can be used for both parts. Furthermore, changes in the logic are easy to implement by changing the specification and generating new executable code.

References

1. Nagl, M., Westfechtel, B., eds.: Integration von Entwicklungssystemen in Ingenieuranwendungen. Springer, Heidelberg (1998)
2. Becker, S., Jäger, D., Schleicher, A., Westfechtel, B.: A delegation-based model for distributed software process management. In Ambriola, V., ed.: Proc. 8th Europ. Workshop on Software Process Technology (EWSPT 2001). LNCS 2077, Springer (2001) 130–144
3. Jäger, D.: Unterstützung übergreifender Kooperation in komplexen Entwicklungsprozessen. Volume 34 of Aachener Beiträge zur Informatik. Wissenschaftsverlag Mainz, Aachen (2003)
4. Heller, M., Jäger, D.: Interorganizational management of development processes. In Nagl, M., Pfaltz, J., eds.: Proc. AGTIVE 2003. LNCS, Springer (2004) In this volume.
5. Westfechtel, B.: Models and Tools for Managing Development Processes. LNCS 1646. Springer, Heidelberg (1999)
6. Westfechtel, B.: A graph-based system for managing configurations of engineering design documents. Intern. Journal of Softw. Eng. and Knowledge Eng. **6** (1996) 549–583
7. Westfechtel, B.: Integrated product and process management for engineering design applications. Integrated Computer-Aided Engineering **3** (1996) 20–35
8. Krüppel-Berndt, S., Westfechtel, B.: RESMOD: A resource management model for development processes. In Engels, G., Rozenberg, G., eds.: TAGT '98 — 6th Intern. Workshop on Theory and Application of Graph Transformation. LNCS 1764, Springer (1998) 390–397
9. 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: Applications, Languages, and Tools. Volume 2. World Scientific, Singapore (1999) 487–550
10. Kiesel, N., Schürr, A., Westfechtel, B.: GRAS, a graph-oriented software engineering database system. Information Systems **20** (1995) 21–51
11. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: Building interactive tools for visual languages. In Callaos, N., Hernandez-Encinas, L., Yetim, F., eds.: Proc. of the 6th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2002). Volume I (Information Systems Development I). (2002) 17–22
12. 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
13. Schleicher, A.: Management of Development Processes – An Evolutionary Approach. Deutscher Universitäts-Verlag, Wiesbaden (2003)
14. Winter, A., Kullbach, B., Riediger, V.: An overview of the GXL graph exchange language. In Diehl, S., ed.: Software Visualization. LNCS 2269, Heidelberg, Springer (2002) 324–336
15. van der Aalst, W.M.P.: Process-oriented architectures for electronic commerce and interorganizational workflows. Information Systems **24** (1999) 639–671
16. Dowson, M.: Integrated project support with ISTAR. IEEE Software **4** (1987) 6–15
17. Smith, R.G.: The contract net protocol: High level communication and control in a distributed problem solver. IEEE Transactions in Computers **29** (1980) 1104–1113
18. Taentzer, G., Fischer, I., Koch, M., Volle, V.: Distributed graph transformation with application to visual design of distributed systems. In Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Parallelism, Concurrency and Distribution. Volume 3. World Scientific, Singapore (1999)