ELSEVIER

Information and Software Technology 42 (2000) 129–139

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

Generating tools from graph-based specifications

D. Jäger

Department of Computer Science III, Aachen University of Technology, D-52074 Aachen, Germany

Abstract

This paper describes an approach for generating graphical, structure-oriented software engineering tools from graph-based specifications. The approach is based on the formal meta modeling of visual languages using graph rewriting systems. Besides the syntactical and semantical rules of the language, these meta models include knowledge from the application domains. This enables the resulting tools to provide the user with high level operations for editing, analysis and execution of models. Tools are constructed by generating source code from the meta model of the visual language, which is written in the very high level programming language PROGRES. The source code is integrated into a framework which is responsible for the invocation of commands and the visualization of graphs. As a case study, a visual language for modeling development processes together with its formal meta model is introduced. The paper shows how a process management tool based on this meta model is generated and reports on our experiences with this approach.

Keywords: Structure-oriented software engineering tools; Graph-based specification; Generation of tools

1. Introduction

Construction of software engineering tools has been a busy area of research and development. Over the years, a wide range of technologies has been developed to make tool construction easier. In particular, base technologies have attracted much attention. This includes, e.g. object management systems for data integration [1], distributed infrastructures for communication integration [2], user interface toolkits for presentation integration [3], and broadcast message servers [4] for control integration.

Even though base technologies provide considerable leverage, tool builders still face difficult problems in tool construction. In particular, this holds for the construction of structure-oriented, intelligent tools operating on complex objects. In the 1980s, this problem was partially addressed by integrated programming environments and meta environments for generating tools from language descriptions [5]. In these environments, programs are internally represented as abstract syntax trees. Syntax-aided editors manipulate the internal representation; incremental error checking and code generation is handled, e.g. by attribute grammars.

Integrated programming environments have a restricted focus in that they mainly address programs (or other textual documents which at least have a well-defined syntax). However, many software documents, in particular those produced in earlier phases of the software life cycle, are based on graphical languages (e.g. CASE tools for the UML). Often, these tools are constructed in an ad hoc manner on top of user interface tool kits. Usually, these tool kits have no understanding of the semantics of the objects which are handled by the tool.

Above this level, generators for graphical editors allow the generation of tools from a description of a graphical language. Often, the language description merely defines the types of objects and links as well as their visualization on the screen [6,7]. The generated editor then offers basic commands for creating and deleting objects and links, for graph layout, etc.

There are also systems which are based on a formal specification of an underlying meta model [8]. By adapting the meta model, structural knowledge from the application domains can be incorporated into the generated tools. Though this approach supports the construction of domain specific tools and the automated checking of structural constraints, it is restricted in that it does not cover complex commands for analyzing and transforming graphs.

In this paper, we report on a framework which addresses these shortcomings. Our tool specifications are based on programmed graph rewriting. They are written in the specification language PROGRES [9,10] which combines concepts from database systems, procedural and rule-based programming. The graph structure is specified by a graph schema which defines types of nodes, edges, and attributes. Graph transformations are described by graph rewrite rules which essentially consist of a left-hand

E-mail address: jaeger@i3.informatik.rwth-aachen.de (D. Jäger).

0950-5849/00/$ - see front matter
PII: S0950-5849(99)00085-3

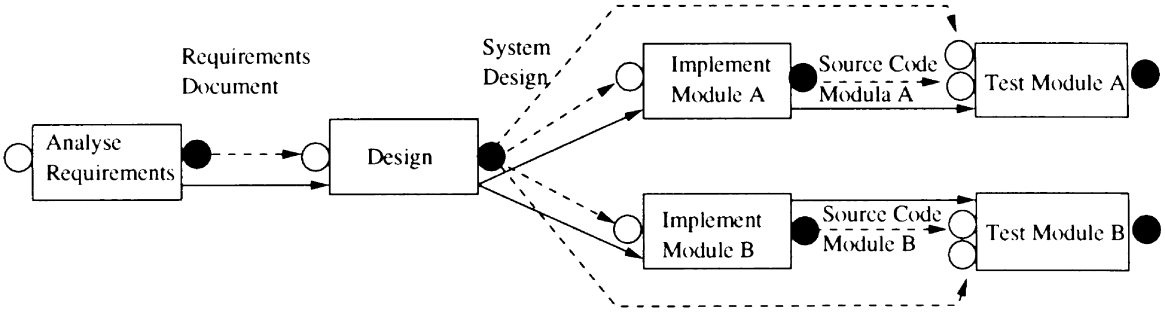

Fig. 1. Sample task net.

side—the graph pattern to be replaced—and a right-hand side—the replacing graph pattern. Programming with graph rewrite rules is supported by control structures such as sequence, branch, loop. From PROGRES specifications, we may generate graphical tools offering complex commands for replacing subgraphs rather than commands operating at the level of single objects and links.

The writers of PROGRES specifications are supported by an integrated development environment which offers structure-oriented tools for editing, analysis, and interpreting. After the specifications has been developed and tested in the PROGRES meta environment, a graphical tool may be generated from the specification. Since the PROGRES development environment has been described extensively elsewhere [11], our presentation will focus on the tools generated from the specification.

As a case study, we will use tools for managing software development processes. It is widely known that software processes are difficult to manage and sophisticated tools are required for planning, enactment, monitoring, or tracing (see Ref. [12] for a survey). Our own approach is based on dynamic task nets, which continuously evolve during enactment due to changing product structures, feedback, alternative task realizations, concurrent/simultaneous engineering, etc. Commands for building up task nets, for enacting the tasks, or for analyzing the current state of the project are specified by a programmed graph rewriting system of considerable size and complexity [13].

This paper is structured as follows: Section 2 gives an introduction to the application domains of graph-based tools. Process modeling is taken as an example to show how to get from the informal concept of a visual language (Section 2.1) by formalizing the concept (Section 2.2) to a structure-oriented tool. Section 3 describes the architecture of our tools and how they are generated from a formal specification, Section 4 reports on our experiences. Section 5 is the conclusion.

2. Application domains of graph-based tools

Our approach has been applied in several domains during recent years. Among these is the reengineering of information systems [14] within the Varlet project. It is aimed at the development of tool support for an explorative and evolutionary reengineering process and the automatic

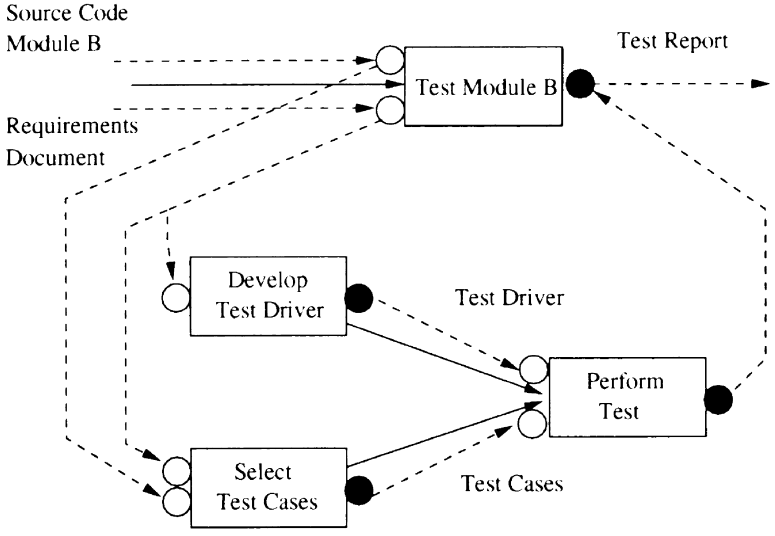

Fig. 2. Task decomposition.

generation of a middleware for object-oriented client/server information systems.

The redesign of legacy applications written in COBOL is another research area where our approach has been applied [15]. The generated tool facilitates the recovery of design information and the subsequent restructuring of the existing source code.

Related to this area is the development of the distribution tool DiTo which supports the distribution of existing monolithic applications [16].

Our research group is concerned with building tools for managing software development processes. Typical of development processes, as they occur, e.g. in software development, is that they are highly dynamic. It is impossible to plan them completely in advance, because neither the structure of the product nor the tasks to be performed are known in detail when the process starts. As a consequence, we cannot simply write down the process in an arbitrary formalism and execute it afterwards. We inevitably have to mix planning and execution of development processes.

The process manager is faced with the error prone task of keeping a constantly evolving process model consistent. To support the process manager, our tools incorporate structural knowledge. They offer complex commands for manipulating process models which automatically preserve the consistency of the model. This requires the use of a formal meta model specifying the object types and relationship types of a process model as well as consistency and manipulation rules.

2.1. Process modeling using dynamic task nets

Dynamic task nets are a visual language for modeling development processes [17]. They are the *external view* on process models, i.e. they are what the users of our tools see on the screen.

Fig. 1 shows a sample task net. A task is represented by a box. Tasks have *input parameters*, denoted as empty circles, and *output parameters*, denoted as filled circles. The parameters are the documents the task is working on. A task reads its input documents upon activation, works with these documents and finally produces some output documents. Tasks are connected by control flow relationships, denoted by arrows, which describe the order of execution of tasks. Data flow relationships, denoted by dashed arrows, can be viewed as a refinement of control flow relationships. While a control flow only states the existence of a sequential dependency between tasks, a data flow exactly describes which output document of one task is consumed as an input document by another task. Within our management tool, a process manager is therefore provided with two levels of abstraction.

Another important feature of dynamic task nets is the distinction between *complex tasks* and *atomic tasks*. We are used to thinking about development projects in a top down manner. For example, developing a software system is a complex task which requires analysis to be performed, a requirements document to be written, a design to specified, etc. Again, these subtasks can be broken down further, leading to several levels of task decomposition until we reach a level on which tasks are simple enough to be handled by a single person in a limited amount of time. We call these atomic tasks.

The decomposition of tasks is shown in Fig. 2. The complex task `Test Module B` is realized by a net that consists of three subtasks. One subtask is the selection of appropriate test cases, another one is the development of a test driver. As soon as both of those tasks are completed, the test can commence. The subtasks work on two kinds of documents. On the one hand, there are intermediate documents, which are only of interest within this level of decomposition, e.g. the test driver for module B. On the other hand, the subtasks consume the input documents which are input to the parent task. This is expressed by data flows between input parameters of the parent task and input parameters of subtasks. Analogously, subtasks eventually produce the documents which are the required output parameters of the parent task. This is denoted by a data flow between the output parameters of the subtask and the output parameters of the parent task.

For dynamic task nets a number of structural consistency rules are specified: there may be no cyclic control flow relationships, because this would mean a deadlock, data flows within one level of the decomposition hierarchy must originate from output parameters and end in input parameters, etc. A tool for modeling development processes using dynamic task nets must know about all of these rules to reject net manipulations which would lead to a faulty net structure.

To make process models executable, not only the structure but also the behavior of tasks has to be defined. This is achieved by assigning a state to each task. The state transitions are specified through a state transition diagram which is depicted in Fig. 3. State transitions are either triggered by certain events within the task net or are requested by the user.

2.2. Formal modeling

The brief description of dynamic task nets given in

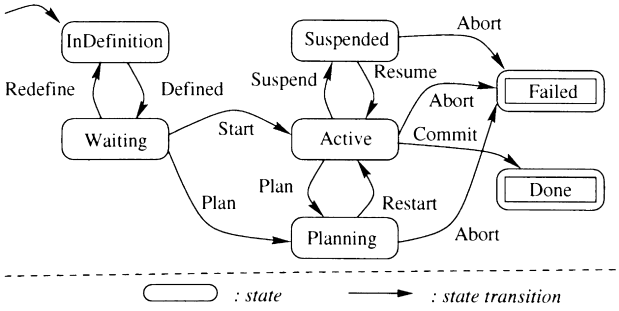

Fig. 3. State transition diagram of tasks.

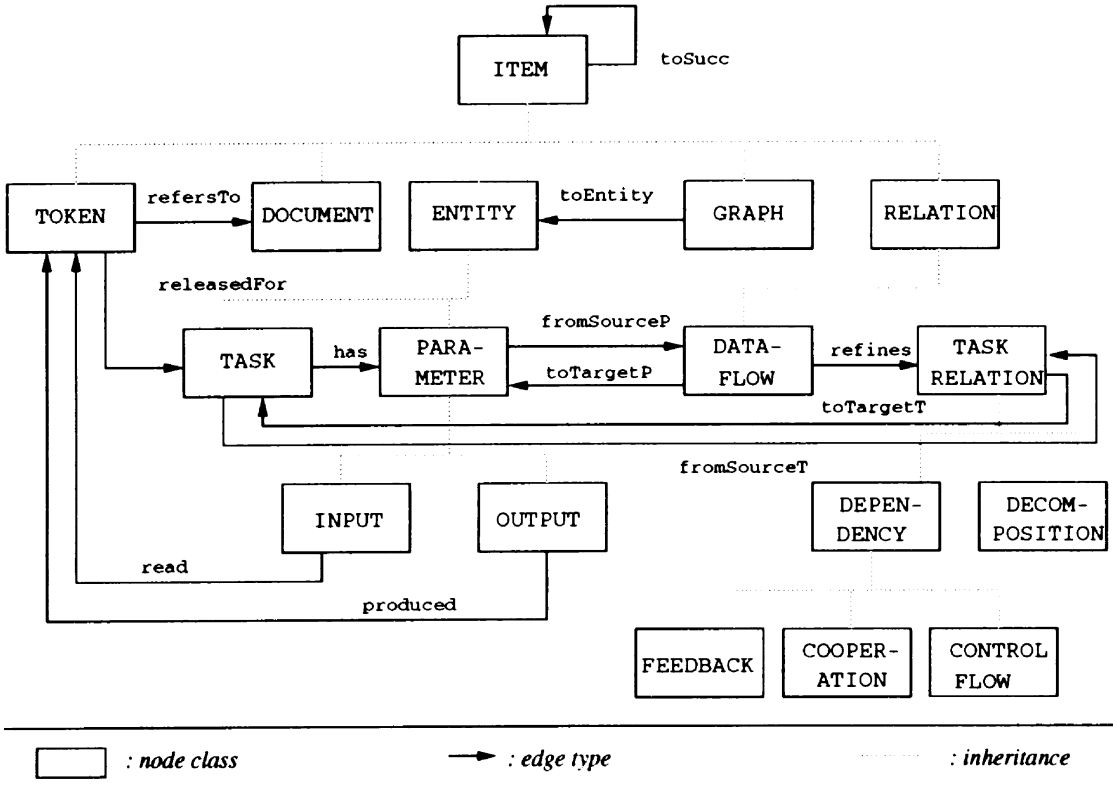

Fig. 4. Graph schema for dynamic task nets.

Section 2.1 is rather informal. The intention was to give an impression of the user's view on dynamic task nets. In the following section, I will present the underlying formal specification, the DYNAMITE (DYNAMIc Task nEts) meta model.

The dynamic task nets presented above are the external view on the process models. The *internal representation* of a model is a graph which is stored inside a special purpose database, called GRAS (GRAph Storage) [18]. Graphs are a very general and widely applicable data model. Every system can be modeled as a graph, because systems by definition consist of entities, which are mapped to nodes, and relationships between entities, which are mapped to edges between nodes. The properties of entities and relationships are expressed by attributes attached to nodes and edges.

Having chosen graphs as the basic data structure, we need a formalism for the manipulation of graphs. As pointed out above, we want to provide the manager of a development process with tools that incorporate process knowledge and automatically ensure consistency of the process model with regard to an arbitrary set of predefined rules. Graph rewriting systems are such a formalism. A graph rewriting system consists of a graph schema and a set of productions. The graph schema defines the node types and edge types of a graph. Graph transformations are described by graph rewrite rules, also called *productions* which essentially consist of a left-hand side—the graph pattern to be replaced—and a right-hand side—the replacing graph pattern.

For constructing the meta models on which our tools are based we use the high level programming language PROGRES, which stands for PRO-grammed Graph REwriting Systems. It offers a variety of features for manipulating graphs, traversing paths within a graph, matching graph patterns, etc. and supports the graphical specification of graph patterns.

Fig. 4 shows a graphical representation of the graph schema of the DYNAMITE meta model. A graph schema in PROGRES is similar to a database schema in (object-oriented) database systems. It declares node classes which are organized into a class hierarchy. PROGRES uses a two level type system where nodes are instances of node types and node types are instances of node classes. Node types are therefore first order objects which can be passed as parameters to productions.

The graph schema of Fig. 4 describes the abstract structure of a dynamic task net. Some important properties are already specified here: control flow relationships can only be established between tasks, data flow relationships can only be introduced between parameters, etc. On the other hand, there are some properties that cannot be expressed by the schema, e.g. that there may not be cyclic control flow relationships. In PROGRES, they are specified by introducing additional constraints. Fig. 5 shows the constraint which forbids control flow cycles.

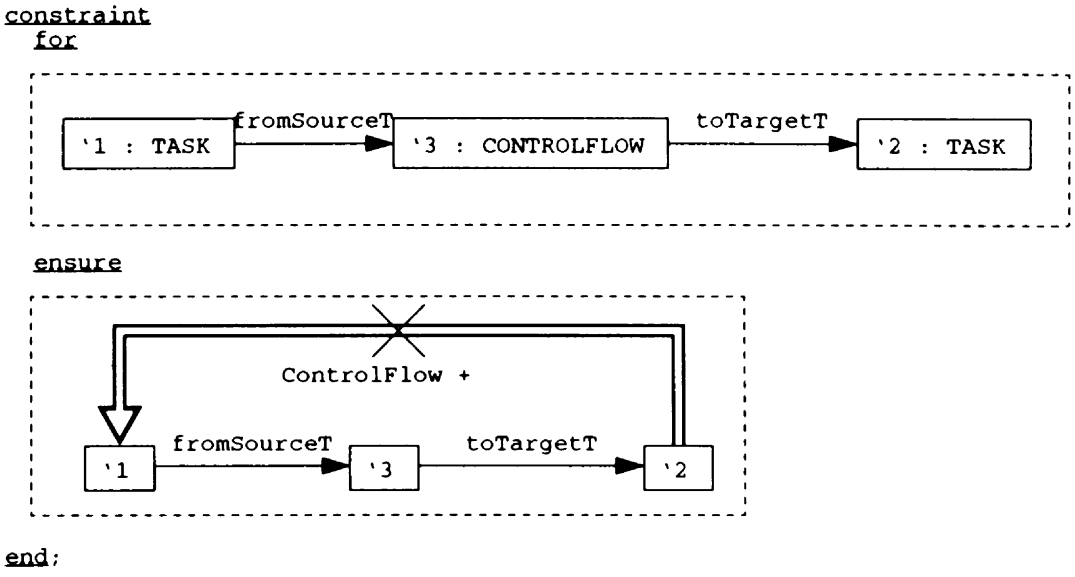

Fig. 5. Constraint for ruling out control flow cycles.

The schema together with constraints is sufficient for specifying the static properties of task nets, but it does not tell us how to build up a net. Remember that our goal is to build a tool for modeling development processes based on the meta model of dynamic task nets. A possible behavior of such a tool would be to allow arbitrary manipulating of task nets and check the consistency of the resulting net and report an error if the static properties are not fulfilled. We consider such a behavior undesirable. We think a user should be provided with a set of high level operations for manipulating task nets which preserve the structural constraints automatically and shield him from the complexity of the underlying formalism.

The basic steps for manipulating task nets are specified as productions. Fig. 6 shows the PROGRES code of a production for inserting a subtask. Several productions can be grouped into a *transaction*. PROGRES offers control structures to direct the flow of control within a transaction, e.g. sequence, branch or conditional execution of productions. Transactions have the ACID properties known from database systems. If one production of a transaction cannot be executed, because there is no match for the left-hand side within the graph, the whole transaction is rolled back.

Transactions are used for constructing the desired high-level operations from rather simple productions. Note that these transactions are the top-level abstraction of our specifications. There is no “main” function from which the control flow originates and which directs the order of execu-

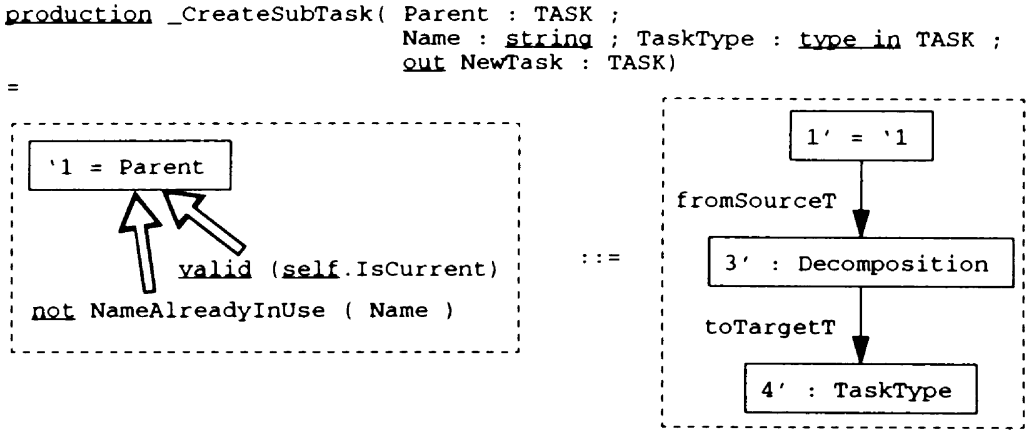

Fig. 6. Graph production for creating a subtask.

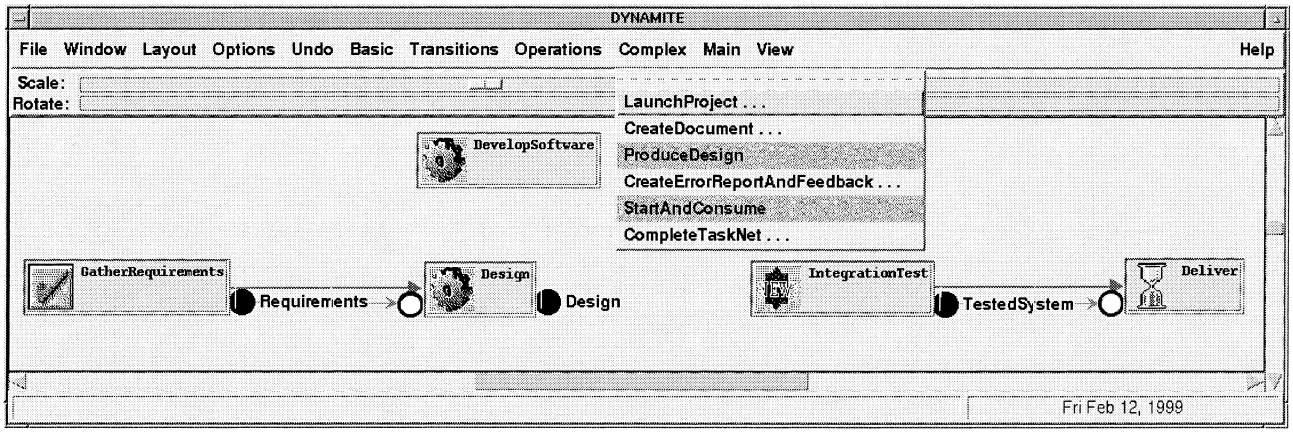

Fig. 7. Tool window showing an initial task net.

tion. Each transaction manipulates the graph in a formally defined manner, thereby preserving its consistency. Within the tools based on this specification, each transaction is mapped to a command the user can invoke through the user interface.

2.3. Using the generated tools

Being familiar now with the external view on dynamic task nets and their formal specification as a graph rewriting system, we are prepared to take a look at the management tool which is generated from the formal specification. The tool allows the creation, manipulation and execution of process models.

Fig. 7 shows a screen shot of the process management tool. The tool displays an initial task net of a software development project. The net is still incomplete, because tasks for implementing and testing the modules of the system cannot be determined until the design has been completed. The net was created through executing the complex command LaunchProject. The menus Basic, Transitions, Operations, Complex, Main and View contain the commands the user may invoke. Each command corresponds to a transaction of the formal specification. The selection of a menu item by the user changes the underlying graph as specified by the transaction or, if for some reason the transaction fails, leaves the graph unchanged and reports an error message.

In Fig. 7 we can see two levels of the task hierarchy. The complex task DevelopSoftware is at the top of the window. At the bottom are the child tasks which realize this complex

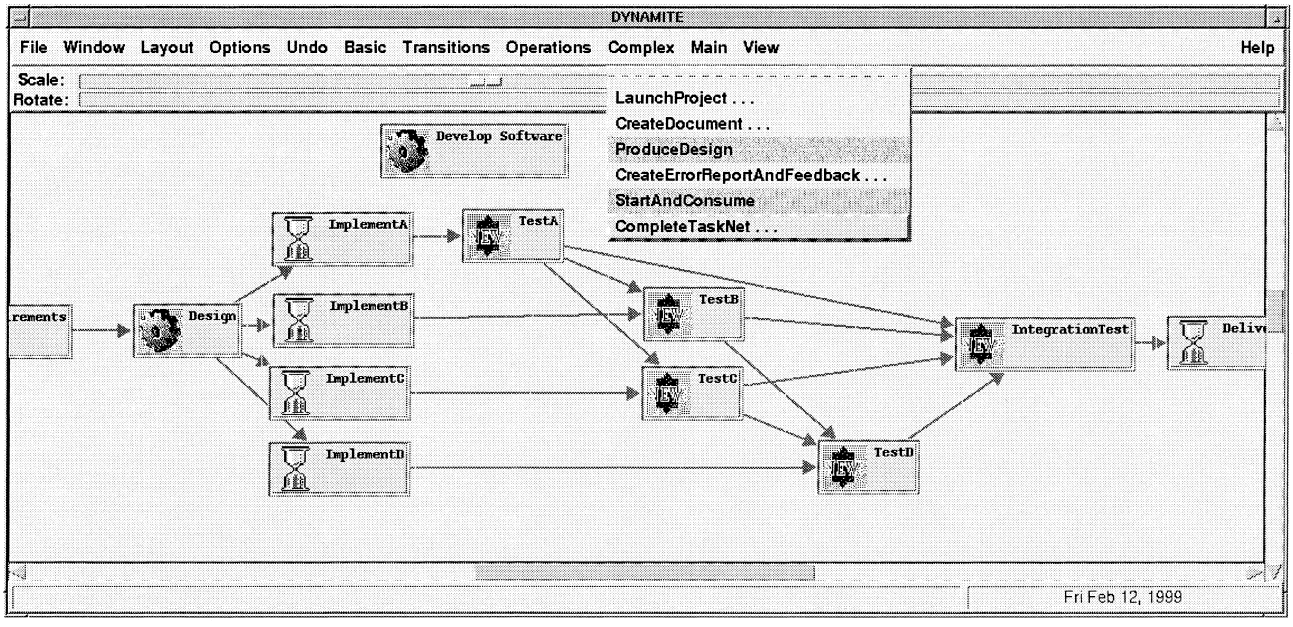

Fig. 8. Tool window after task net extension.

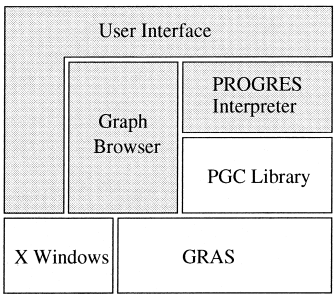

Fig. 9. Structure of PROGRES development environment.

task. The execution states of the tasks are visualized by their different icons. For example, the parent task DevelopSoftware and the task Design are in state Active while the task GatherRequirements is already in state Done.

Executing a command usually causes a modification of the task net. Fig. 8 shows the tool after the command CompleteTaskNet was executed. Assuming the design of the system is now available, this command inserts a corresponding subnet containing the implementation tasks and subsequent test tasks.[1] The interdependencies between the test tasks reflect the module hierarchy.

Due to the dynamics of the development processes we are modeling with our tool, such net changes occur frequently and some commands insert a considerable number of nodes into the net. For each newly inserted node, a position within the tool window has to be determined and existing nodes have to be repositioned to free the necessary space.

We think that the tool should support the user with an incremental layout algorithm that positions newly inserted node automatically while changing the positions of existing nodes as little as possible. Although we recognize this, our tool does not satisfy this requirement yet. Section 3.2.1 contains a more detailed discussion of the layout problem.

2.4. The need for model adaption

Code generation usually requires a framework in which the generated code is integrated. The effort to write a framework is considerably higher than implementing the whole application manually. It can only be justified if the framework can be reused frequently. Of course, the framework is reused in different domains and for different kinds of tools, but one might argue that for dynamic task nets it would suffice to implement the management tool based on the meta model manually.

This is only partly true, because the DYNAMITE meta model is merely a base model. It does not contain specific process knowledge. When a process evolves, the knowledge about the process increases. In most cases, knowledge can also be transferred from similar processes in the past. The incorporation of this knowledge requires an adaption of the base model.

Model adaption is achieved by extending the base model with new transactions, productions and node types which change the behavior as well as the structural constraints of a task net. Due to the need for model adaption we cannot simply implement our tool once. We must produce a new implementation for each specific process model we encounter. The approach can only work if a re-implementation requires minimum effort.

3. Tool implementation

A specification written in PROGRES is executable. The PROGRES programming environment is able to interpret PROGRES specifications step by step and construct a graph in the underlying graph database GRAS. This means, while executing the specification of the DYNAMITE meta model and constructing graphs according to it, the PROGRES environment already acts as a process modeling tool. For three reasons, this is not the kind of tool we would like to have.

Firstly, interpreting a PROGRES specification is rather slow. We would prefer to compile the specification into efficient code which can be executed independently of the development environment.

Secondly, there is only a generic visualization of the model stored inside the database which does not conform to the external view a user should have on a dynamic task net.

Thirdly, there is no comfortable way of command invocation, i.e. triggering the execution of a transaction.

Fig. 9 shows a survey of the structure of the PROGRES development environment as far as the execution of specifications is concerned. Note that for the sake of simplicity some parts, e.g. for editing a specification, were omitted in this figure. Moreover, the internal structure of the user interface and the graph browser are not depicted. For accessing the graph inside GRAS, the PROGRES interpreter uses the functions of the PGC library (PROGRES Graph Code) which provide a level of abstraction above the procedural interface of the database itself.

Starting from the system shown in Fig. 9 our goal is to build a stand-alone version of a tool. Therefore we must replace three components (shown as gray boxes) of the system which are part of the PROGRES development environment, namely the interpreter, the graph browser and the user interface, by components that are on the one hand considerably more lightweight and on the other hand provide the user with a more convenient interface.

3.1. Code generation

The first component to be replaced is the PROGRES interpreter. Instead of interpreting a transaction's PROGRES source code each time it is executed, we gener-

[1] Understanding how the command knows about the module hierarchy, analyses it and inserts the corresponding tasks would require a more detailed discussion of dynamic task nets.

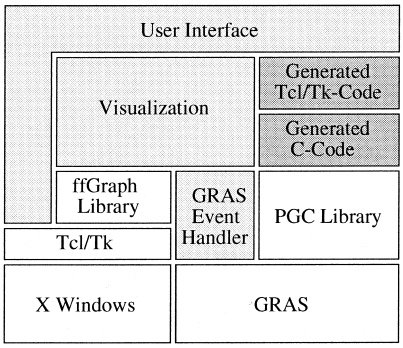

Fig. 10. Structure of generated tools.

ate source code in C or Modula-2 which calls the functions of the PGC library exactly the way the interpreter does when executing the specification. To understand the problems to be addressed by the generated code we have to consider how the PROGRES interpreter executes a specification. I will therefore discuss two very important features of execution: the matching of the left-hand side of a production and the handling of non-deterministic specifications.

The first step in applying a production to a host graph is to find a match for the subgraph of the left-hand side of the production within that host graph. This (*sub-*)*graph isomorphism problem* is known to be NP-complete in the size of the left-hand side of the applied production. Nevertheless, a heuristic solution of the problem is possible, which employs the additional knowledge we have on the subgraph to determine an efficient search plan [19]. The main idea is to rule out dead ends as soon as possible while minimizing costly queries to GRAS. For example, if one can decide whether a node is part of a match by examining its attributes, this is clearly preferable to examining the types of its adjacent nodes.

To find the optimal search plan for matching a production's left-hand side, the PROGRES interpreter determines for each of its elements (nodes and edges) the query actions which are necessary to match that particular element. Though dependencies between actions exist which restrict the order of their execution, there usually is a large number of possible search plans. The cost of a search plan is calculated using the estimated costs of its query actions and the plan with minimal costs is chosen.

A production's search plan has to be calculated only once, as long as the production is not changed. The source code for matching the left-hand side is therefore a hard-coded version of the search plan. To a great extent, it is the use of this heuristic solution for the graph isomorphism problem in PROGRES which renders the efficient execution of productions possible, allowing us to build tools which are based on graph rewriting systems.

Another important feature of PROGRES is its support of non-determinism. If there are multiple matches for a left-hand side of a production within a graph, one of these matches is selected and the production is applied. Transactions usually consist of a sequence of productions which are applied to the graph until one of them fails, because there is no match for its left-hand side. In this case, the transaction is rolled back to the point where the last non-deterministic choice was made, another match is selected and the execution of the transaction is resumed. This backtracking mechanism ensures that a transaction only fails if all possible combinations of matches have been tried without success.

When generating source code, we are therefore faced with the problem of implementing control flow backtracking in an imperative programming language. It is solved by employing a special programming style which uses nested loops and case expressions (see Ref. [20] for details). Data backtracking, i.e. resetting the graph inside GRAS to an earlier state, is supported by the extensive undo/redo mechanisms offered by GRAS [18].

3.2. Framework

Code generation provides us with a set of functions which encapsulate transactions on graphs stored in GRAS. These functions must be embedded into a framework. The framework has to offer an interface to the user through which he can activate a transaction and it has to visualize the graph in an appropriate way. Fig. 10 depicts the architecture of the resulting tools. The generated parts are shown in dark gray. Parts belonging to the framework are shown in light gray.

3.2.1. Visualization of graphs

For the visualization of graph structures inside a tool, several problems have to be solved. Firstly, we have to map the graph inside the database, which we call the *logical graph*, to the graph we want to display, which we call the *representation graph*. This is necessary, because the logical graph usually contains nodes and edges that are internal to the model and should not be presented to the user. Moreover, there might be different views on a model. For a dynamic task net, we have a control flow view and a data flow view. Not all the model elements should be visible in each view.

Secondly, the framework has to contain an appropriate set of *representation elements*, e.g. for displaying the nodes and edges of a graph as elements of a dynamic task net. Thirdly, we must be able to calculate a proper *layout* for this graph representation.

The user interface of our tools is based on the ffGraph library, a C++ class library for creating, manipulating and displaying directed graphs [21]. The representation graph is constructed from the node and edge classes of the ffGraph library. The ffGraph library uses the Tcl/Tk widget library [22] for Windows X11 systems. Tcl/Tk is a script language thus making the extension and modification of the tools easy. For example, the menus of the tools (see Fig. 7) are generated together with the C-code for the transactions and inserted into the main window when the tool is started. The

shape of nodes and edges can be customized to a certain extent, though it is difficult to program really sophisticated representation elements.

A central problem of graph visualization is the one of laying out a graph. Though is has been studied for a long time (cf. Ref. [23]), there is still no general solution. Only for some special cases, like planar graphs, solutions exist which are fast and produce nice results. One problem is that although humans can easily decide if they regard a certain layout as "nice" it is difficult to specify a complete and consistent set of rules for how to produce such a "nice" layout. Even if such a set of rules exists it is in general still NP-hard to calculate a layout according to it [24].

As I mentioned in Section 2.3, our models are highly dynamic and thus require an automated layout. Standard layout algorithms like Sugiyama [25] or Spring Embedder [26] are designed to layout arbitrary graphs and have no knowledge about the semantics of nodes and edges (assuming a semantics exists). On the other hand, the graphs within our tool are sentences of some visual language and thus do have a semantics which determines the desired layout to a great extent.

For example, in dynamic task nets we would like to order the tasks according to their control flow relationship from left to right. We can achieve this by using a *constraint-based* layout algorithm where we can define constraints restricting the relative placements of nodes. For ordering the tasks according to the control flow we may insert constraints of type LeftOf and RightOf, respectively, as invisible edges between subsequent task nodes into the net. A constraint-based layout is calculated in two steps. At first, the *constraint solver* computes for each node the possible range for its horizontal position. Afterwards, the actual *layout algorithm* places the nodes by selecting a position within that range according to some additional criteria, e.g. minimizing the net's horizontal extent.

The use of constraints offers a variety of possibilities for controlling the layout. Because they are edges of the graph, we can add and remove constraints at runtime, allowing us to change the behavior of the layout algorithm without touching its implementation. Moreover, the layout algorithm works incrementally. It repositions only those nodes which violate constraints, and even in that case it tries to keep them as close as possible to their original positions. Regarding the way the graphs inside our tools are changed, the advantage becomes obvious: a transaction usually manipulates a rather small area of the graph by inserting nodes and edges. These newly inserted parts have to be positioned while changing the rest of the graph as little as possible.

Layout is still an open problem for us. Currently the user of our tools can chose between the Sugiyama and Spring Embedder algorithm. This has the drawbacks described above. During the last years, we were constantly looking for a layout algorithm that satisfies our needs, but finally decided that there is no off-the-shelf solution to the problem. The described constraint-based layout algorithm is not available in the current version of our framework. But it is already implemented and used inside the PROGRES development environment and we are going to integrate it into the next version of the framework (see Section 4). We recognized that each visual language requires a specially adapted layout algorithm and we think that constraint-based layout, possibly combined with a standard layout algorithm, will be well suited for the task.

3.2.2. Activation of commands

The framework employs the Model–View–Controller design pattern for decoupling the activation of commands, the logical model and its visualization [27].

The user requests the execution of a command by selecting the corresponding menu item in the tools main window. If the command requires additional parameters, the user enters them in a dialog box. Often, one of the parameters is the identifier of the node to which a command shall be applied. In this case, the user may select the node and activate the command afterwards. The identifier of the selected node is taken as the required parameter.

After the parameters are checked for consistency and completeness, the generated code implementing the command is called. Through the functions of the PGC library, this code attempts to execute a transaction on the database. If the transaction fails, an error code is propagated upwards causing the user interface to display an error message. If the transaction is successful, it returns quietly.

Usually, each successful transaction causes a sequence of changes to the database, e.g. inserting or removing nodes and edges. If such changes occur, an event handler which was attached to the database when the tool started, is notified. The event handler collects the events caused by the transaction and passes them to the view window which updates the representation of the graph accordingly. It is also possible to attach multiple views to the database to provide the user with different visualizations of the model.

4. Experiences and lessons learned

At the beginning of the design, we had to decide whether to use commercial software within our framework or found it completely on free software (e.g. public domain or under GNU license). There are commercial toolkits that offer a variety of functions for handling and displaying huge graph structures. The use of such toolkits significantly reduces the development effort and increases the performance but usually the resulting tools may not be distributed as free software. Therefore, we decided to use the ffGraph library and Tcl/Tk which is available under GNU license.

Though we were able to distribute the whole system, including PROGRES, GRAS and the framework, under GNU license, the choice had some serious drawbacks.

It turned out that the framework becomes unacceptably

slow when graphs are larger than about 100 nodes. This is partly due to the fact that Tcl/Tk has only strings as data type. At the interface between the Tcl/Tk part and the C++ part of the ffGraph library, all parameters have to be converted into strings or from strings, respectively. Profiling our tools showed that under certain conditions, the program spends most of its time in string conversion routines.

Moreover, Tcl/Tk code is unstructured and difficult to maintain. A large amount of our software is developed by students as part of their diploma thesis. As the system evolved, the amount of time it took for new students to understand the existing code became very large.

Another problem is the appearance of our tools. Tcl/Tk is a toolkit for implementing user interfaces, not for drawing arbitrary graphics. Therefore, it is difficult to program sophisticated graphical representations for nodes and edges. People tend to judge software by its outward appearance. We often found that few people were attracted when we presented our tools in public, because at the first glance they might look like just another graph editor with an ugly user interface. But those who were attracted changed their opinion when they recognized the power of the generative approach behind our tools.

Due to these drawbacks we have decided to re-implement the framework in Java. We are going to replace the ffGraph library by the commercial toolkit ILOG JViews which provides classes for the efficient construction and visualization of graphs.

This promises several advantages: Firstly, the whole implementation becomes more homogeneous, avoiding the costly parameter conversions between Tcl/Tk and C-Code. Secondly, ILOG JViews is able to handle large graphs with more than 1000 nodes efficiently. Thirdly, Java offers a richer set of functions for programming sophisticated graphical representations of nodes and edges. The shapes of nodes and edges can be programmed exactly as the designer of the visual language intends them to be. Fourthly, the generated tools can be used on different hardware platforms, due to Java's portability. However, the portability of the whole system is still restricted to those platforms on which an implementation of GRAS exists, currently UNIX and Windows-NT.

The drawback when using a commercial toolkit is that our tools are no longer distributable under GNU license. We will therefore still maintain the old version of the framework for this purpose.

While the re-implementation of the framework has already started and should be finished by the end of this year, one of our long-term goals is to replace the database GRAS and store the graph either inside a standard object-oriented database or as a data structure inside the main memory. By doing so we would loose the special features GRAS offers for the storage and handling of graphs. On the other hand, the transition from GRAS to a more lightweight solution promises a considerable increase in speed. Further research is needed to find out whether standard object-oriented databases or a main memory implementation of the graph can satisfy all of our demands.

5. Conclusion

I presented our approach of generating structure-oriented software engineering tools from graph-based specifications of visual languages and reported on our experiences. The tools incorporate knowledge from their application domains and provide complex commands for constructing and manipulating graphical models. These commands preserve the consistency of the models and go beyond the operations usually offered by graphical editors.

For the realization of this approach, problems from various domains had to be solved, including database systems, graph theory, software architecture, graph layout and user interface design. The purpose of this paper was to give a survey on the whole approach. The details of each area can be found in the cited publications.

The successful application to different research areas has demonstrated the power of the approach. But it has also shown the problems of the current implementation, which often result from early design decisions and thus are difficult to correct now. We are going to solve these problems within the re-implementation of our framework. Through an improved user interface, including sophisticated visualization as well as comfortable activation of commands, the usability of our tools will be significantly increased. In this way, we hope to attract new users who will apply our tools in an industrial context and provide us with valuable feedback from the application domains.

References

[1] L. Wakeman, J. Jowett, PCTE—The Standard for Open Repositories, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[2] T.J. Mowbray, W.A. Ruth, Inside CORBA, Addison-Wesley, Reading, MA, 1997.
[3] R.W. Scheifler, J. Gettys, A. Mento, D. Converse, X Window System: Core Library and Standards, Digital Press, 1996.
[4] S. Reiss, Interacting with the FIELD environment, Software Practice and Experience 20 (S1) (1990) 89–115.
[5] T. Reps, T. Teitelbaum, Synthesizer Generator: A System for Constructing Language-Based Editors, Springer, New York, 1988.
[6] R. Bardohl, GENGED—a generic graphical editor for visual languages based on algebraic graph grammars, Proceedings of the IEEE Symphosium on Visual Languages (VL'98), Los Alamitos CA, 1998, IEEE Computer Society Press, Silver Spring, MD, 1998.
[7] M. Minas, G. Viehstaedt, DiaGen: a generator for diagram editors, Proceedings of the IEEE Symposium on Visual Languages (VL'95), Los Alamitos, CA, 1995, IEEE Computer Society Press, Silver Spring, MD, 1995.
[8] MetaCase Consulting. MetaEdit+, Version 2.0: Method Workbench User's Guide. MetaCase Consulting, Micro Works Finland, September 1995.
[9] A. Schürr, A. Winter, A. Zündorf, Graph grammar engineering with PROGRES, in: W. Schäfer, P. Botella (Eds.), Proceedings of the European Software Engineering Conference (ESEC'95) Barcelona,

Spain, September 1995, Lecture Notes in Computer Science, 989, Springer, Berlin, 1995, pp. 219–234.
[10] A. Schürr, A. Zündorf, Specification of logical documents and tools, in: M. Nagl (Ed.), Building Tightly-Integrated Software Development Environments: The IPSEN Approach, Lecture Notes in Computer Science, 1170, Springer, Berlin, 1996, pp. 297–324.
[11] A. Schürr, A. Winter, A. Zündorf, The PROGRES approach: language and environment, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools, 2, World Scientific, Singapore, 1999, pp. 487–550.
[12] A. Finkelstein, J. Kramer, B. Nuseibeh, Software Process Modelling and Technology. Advanced Software Development Series, Research Studies Press (Wiley), Chichester, UK, 1994.
[13] P. Heimann, C.-A. Krapp, B. Westfechtel, G. Joeris, Graph-based software process management, International Journal of Software Engineering and Knowledge Engineering 7 (4) (1997) 431–455.
[14] J.H. Jahnke, J. Wadsack, Integration of analysis and redesign activities in information systems engineering, Proceedings of the Third European Conference on Software Maintenance and Reengineering (CSMR'99), Amsterdam, 1999, IEEE Press, New York, 1999.
[15] K. Cremer, A tool supporting the re-design of legacy applications, in: P. Nesi, F. Lehner (Eds.), Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, Silver Spring, MD, 1998, pp. 142–148.
[16] A. Radermacher, Tool support for the development of distributed applications, in: TAGT'98—Theory and Application of Graph Transformations, technical report tr-ri-98-201, Paderborn, University Paderborn, Germany, November 1998.
[17] P. Heimann, G. Joers, C.-A. Krapp, B. Westfechtel, DYNAMITE: dynamic task nets for software process management, Proceedings of the 18th International Conference on Software Engineering, Berlin, March 1996, IEEE Computer Society Press, Silver Spring, MD, 1996 pp. 331–341.
[18] N. Kiesel, A. Schürr, B. Westfechtel, GRAS: a graph-oriented software engineering database system, Information Systems 20 (1) (1995) 21–51.
[19] A. Zündorf, A heuristic for the subgraph isomorphism problem in executing PROGRES, Technical Report AIB 93-5, Department of Computer Science III, Aachen University of Technology, Aachen, Germany, 1993.
[20] A. Zündorf, Implementation of the imperative/rule based language PROGRES, Technical Report AIB 92-38, Department of Computer Science III, Aachen University of Technology, Aachen, Germany, 1992.
[21] C. Friedrich, The ffGraph library, Technical Report 9520, University of Passau, Germany, 1995.
[22] J.K. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley, Reading, MA, 1994.
[23] W.T. Tutte, How to draw a graph, Proceedings of the London Mathematical Society 13 (1963) 743–763.
[24] F.J. Brandenburg, Nice drawings of graphs are computationally hard, Lecture Notes in Computer Science 439 (1990) 1–15.
[25] K. Sugiyama, S. Tagawa, M. Toda, Methods for visual understanding of hierarchical system structures, IEEE Transactions on Systems, Man, and Cybernetics SMC-11 (1981) 109–125.
[26] P. Eades, heuristic for graph drawing, Congressus Numerantium 42 (1984) 149–160.
[27] P. Krasner, A cookbook for using the model–view–controller user interface paradigm in smalltalk-80, Journal of Object-oriented Programming 1 (3) (1988) 26–49.