GRIDS - GRaph-based, Integrated Development of Software: Integrating Different Perspectives of Software Engineering

Andreas Zamperoni

Leiden University
Dept. of Computer Science
P.O. Box 9512, 2300 RA Leiden, The Netherlands
email: zamper@wi.leidenuniv.nl
www: http://www.wi.leidenuniv.nl/~zamper/work.html

Abstract

Goal of the *GRIDS* project is to provide a formally based, multi-dimensional software engineering model - and tool - that integrates "partial" models of software processes, system architectures, and views onto the system into one consistent project framework, in order to enhance real-life, large-scale software development.

In this paper, we first introduce the static part of the so-called *Three-Dimensional Model of Software Engineering* (*3DM*), which captures and structures partial models, integrated project frameworks, and other relevant project information. We further describe the dynamic part of the *3DM*, which provides the necessary actions to generate, manipulate and maintain the entities of the static part.

Using the programmed graph rewriting system PROGRES gives us a powerful means to formally specify our conceptual model. We show how we apply PROGRES to formalize the *3DM*, and present the prototype of a tool, generated from the formal specification of the static and dynamic parts of the *3DM*.

Keywords: method engineering, meta-modeling, project frameworks, model integration, graph rewriting systems

1 Introduction: From One-dimensional to Multi-dimensional Software Engineering

Research in the field of software engineering is usually dealing with its complexity by investigating the different areas of concern separately. Conceptual models are developed for individual aspects, such as e.g., software process modeling [AM92, BFG93, TDK94], (object-oriented) specification techniques [Boo91, RBP+91, JCJO92], software architecture [FkNO92, PW92, GAO94], or requirements engineering [Poh92].

Focusing on isolated areas of concern can lead to sophisticated theoretical models, and thereby to more insights and deeper understanding about that certain area. But in real-life software development projects, the practical use of such "partial" models, resulting from focussed research, is often only very limited, because they often fail to capture the comprehensiveness of a software project, when different areas of concern interfere, and people with different roles and different tasks interact. As consequence of this shortcoming of the partial models, their acceptance among software engineering practitioners is often quite low [Pot93].

At *TNO Institute of Applied Geoscience*[1], the experiences with shortcomings of existing software engineering methods, dealing exclusively with single aspects of software engineering, triggered the **GRIDS**[2] project. Motivation of the project is to investigate how software development can be enhanced by modeling different areas of concern of software engineering within the same conceptual approach, and to integrate them into a comprehensive software engineering framework that can be adapted and used for a wide range of software projects.

Goal of *GRIDS* is to develop a **formally based, multi-dimensional model** that models and integrates different areas of concern of software development, and therefore can serve as basis to describe - and prescribe - the different tasks and products of *real* software projects consistently from different viewpoints.

To support this goal it is necessary to construct a **project-supporting tool** th that formal multi-dimensional model as basis, but that hides complexity and formality to its users, the software engineering practitioners.

In this paper we will introduce this multi-dimensional model (sec. 2), its underlying formalism (sec. 3), give an overview of its static (sec. 4) and dynamic part (sec. 5), and give shortly attention to the tool prototype (sec. 6).

2 3DM - a Multi-dimensional Software Engineering Model

The development of large-scale software is mastered by splitting it up into suitable units of decomposition - or composition - in order to divide the various development tasks into logical units of manageable size. Among all potential areas of concern that may and do trigger such a decomposition, three are generally fundamental for software development:

- The **software process**: Modeling the software process is essential to organize the temporal succession, iteration, and parallelism of development activities. In general, the software process is characterized by its *phases* (steps) and their *mutual dependencies* which describe the sequence and order of steps from an abstract, problem domain-oriented starting point to a technical, solution-oriented end point.
- The **views onto the system**: During the course of a project, a system and its requirements are described from different views, the most common being the static, the functional, and the behavioral view. Using different *views* in the specification documents of a project is indispensable to get a complete picture of the software to develop. Choosing the right views (first) has even become the focal point of many disputes concerning the best software engineering methodology [SO92].
- The **software architecture**: A key property for the quality of a software system is its software architecture. A good architecture makes the system more efficient, easier to understand, change, test and maintain, but has also influence on the development process itself, because the selection and organization of *system components* predetermines which parts of a system can be reused or adapted (from previous projects or vendor software), and which parts have to be developed from scratch. Therefore, it is important to deal with the architectural issues of components and their dependencies on a high level from the beginning, and rigorously. Software engineering methodologies like OMT [RBP+91] have recognized this and assigned it an own phase (system design) in their software process, but there are also research efforts dealing exclusively with software architecture (e.g., [PW92, GAO94]).

[1] Cf. sec. 8 for more about this research department.

[2] *GRIDS* is the acronym for **GR**aph-based **I**ntegrated **D**evelopment of **S**oftware.

The importance of all three of these areas of concern reflected by the numerous approaches, models, and specification techniques currently being available and worked on by research groups, and by the increasing number of specific tools offering support to commercial software developers.

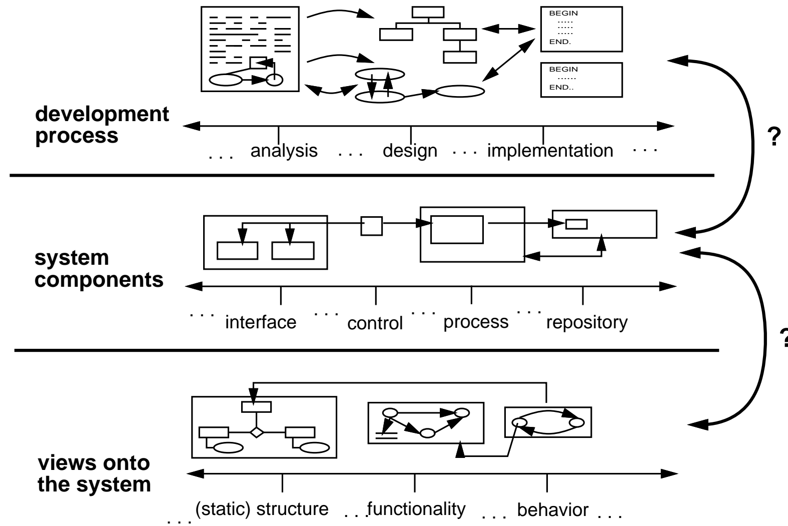

Figure 1: Traditional software engineering models deal with one particular area of concern at a time

However, those "partial" - one-dimensional - models help organizing real-life software development only in a limited way, because constraints relating different areas of concern are not modeled, and hence, can't be easily enforced. Examples of such constraints, crossing areas of concern, are e.g., "*Which views are necessary to be specified for a certain component, and which are not?*" or "*When (in which phase) has a certain component to be specified/introduced (i.e., from scratch, or later)?*" (cf. fig. 1).

To cope with these requirements to software engineering models, *GRIDS* concentrates on the **integration** of the different areas of concern into one comprehensive, multi-dimensional model. The three areas of concern introduced above are considered as especially important for system development, and used representatively as fundamental structuring mechanism for the construction of the multi-dimensional model. Consequently, modeling the isolated areas of concern will not be a purpose of its own anymore, but will become *modeling the dimensions* of the multi-dimensional model. As we presently limit to three "dimensions", the resulting model will be called **Three-dimensional Model of Software Engineering** (**3DM**) in the following.

The phases of the process, the components of the architecture, and the perspectives of the view dimension are the **constituents** of the three dimensions, i.e., they are the "building blocks" for the partial models that capture the three areas of concern involved.

Of course, software engineering comprises many more areas of concern, but main purpose of *GRIDS* is to investigate integration issues of areas of concern in general, and on a formal basis. Other areas of concern can be important or even crucial for certain systems and in certain situations, too (e.g., resource management, version management), but they do not contribute in *3DM* to the course of a project in a structuring way. Where important information concerning other "dimensions" of software development, such as e.g., commitment of people to tasks, access rights to documents, and other project-related information is also included in the *3DM*, it is done in a subordinate way, and doesn't contribute to the structure of *3DM* models (cf. sec. 4).

The analogy of a **three-dimensional network** is used to support the intuition of the *3DM*. Each *node* of the network represents a logical unit of development - called **software engineering fragment** - which contains information about a (small) portion of the actual software development. Each software engineering fragment is identified by a unique combination of constituents of each of the dimensions of the *3DM*. These combinations of constituents determine, coordinate-like, the exact position of the fragment within the context of the software development project. *Edges*, connecting the nodes of the network, model relationships and constraints between the fragments[3].

Figure 2 illustrates the analogy of the three-dimensional network containing related fragments. The constituents of the dimensions are simplified for the purpose of clarity. Fragment `X` represents the *behavioral view* of the *interface component* in the *analysis phase*, while `Y` represents the *static structure* of the *control component* during *design*. Between the two fragments, a relationship `Information_Source_For` models the fact that (the specification of) fragment `X` serves as information source when working on (the specification document of) fragment `Y`.

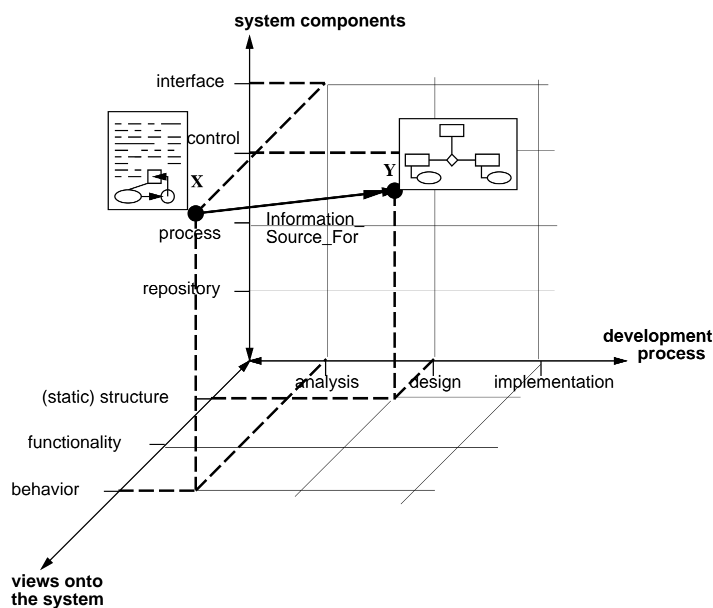

Figure 2: An example of software engineering fragments.

As motivated in the introduction, a major purpose of the model is pre- and describing the course of actual software projects. This implies a certain kind of construction process of the network that models a certain project, and furthermore the possibility of evolution of that network to adapt to changes that occur during the course of the project. Modeling these dynamics of the network are part of *3DM*, too.

The relationships between the nodes of the network constrain also how (whether, in which order, and under which circumstances, etc.) members of a software development team can access the software engineering fragments to perform the tasks to which they have been committed. This

[3]Even more intuitive is the analogy of the molecular grid structure of a crystal lattice from which the acronym for the project was derived. The molecules (= fragments) - arbitrary complex units containing information - are the basic logical units of the grid (= project framework), while electrons (= edges) bind the molecules, i.e., establish relationships between them. Each crystal (= project framework) has its own particular shape, but all are built up in a more or less regular - organized and clear - form, according to the higher principle of chemistry (= software engineering).

means that the work of the project team members can be modeled by prescribing, identifying, and following **paths through the 3DM network**.

We will present the different parts of the *3DM* - especially the static network (in sec. 4) and the network dynamics (in section 5) in more detail, concentrating on the way they can be formalized with graph rewriting systems. Therefore, we first introduce the formal system we use to specify *3DM*. An extensive descriptions of the conceptual modeling of *3DM* can be found in [Zam94].

3 Graph Rewriting Systems as Formal Specification Language for 3DM

Using a *network* as analogy for the static structure of the *3DM* suggests using *graphs* to formalize the model. In the past, special types of graphs have been successfully applied to systematically capture the syntax and semantics of various, complex problem domains (e.g., [Eng92, Wes92, Rek94]).

We use *attributed, node and edge labeled, directed graphs*, and in particular, the graph rewriting system **PROGRES**[4] [Sch91], because PROGRES offers a rich and powerful graph specification and manipulation language for the formalization of the conceptual *3DM* model. Furthermore, the developers of PROGRES provide a range of development and prototyping tools (cf. sec. 6).

In its static part, PROGRES is based on **graphs** which consist of labeled, attributed nodes which are connected by binary, directed, and labeled edges. These graphs can be specified by ER-like graph schemes. The static structure of *3DM* (*3DM* graphs) can be straightforwardly specified as PROGRES **graph class** (cf. fig. 3) PROGRES node classes can have (node-)subclasses (as e.g., `CONSTITUENT` and its subclasses `VIEW`, `PHASE`, and `COMPONENT`). Subclasses inherit the features and constraints of their ancestors (multiple inheritance is possible), avoiding redundant definition of node properties. An overview of the static structure of the *3DM* will be presented in section 4.

PROGRES supports a sophisticated, rule-oriented and diagrammatic specification of atomic **graph rewriting rules** with complex preconditions, and imperative programming of composite graph transformation processes by means of deterministic and non-deterministic control structures. Describing the full range of graph rewriting concepts of PROGRES goes beyond the scope of this paper, but examples of PROGRES graph transformation rules will be given and explained when describing the dynamics of *3DM* in section 5.

Additionally, PROGRES includes a number of **tools**, including a syntax-directed editor for visual and textual specification of graph schemes and graph transformations, and facilities to **execute specifications**, visualizing the resulting graphs and their evolutions. Furthermore, **generation of interactive prototypes** from PROGRES specifications is supported. This allows the construction of a tool prototype on top of the formal *3DM* specification (cf. sec. 6).

4 The static structure of 3DM

Figure 3 shows the three parts of the static structure of *3DM*. Node classes are drawn as rectangles, edge types as (labeled) solid arrows between two node classes. The dotted arrows depict inheritance between node classes. For clarity reasons, only a part of the static structure and no node attributes are displayed.

[4]**PRO**grammed **G**raph **RE**writing **S**ystems

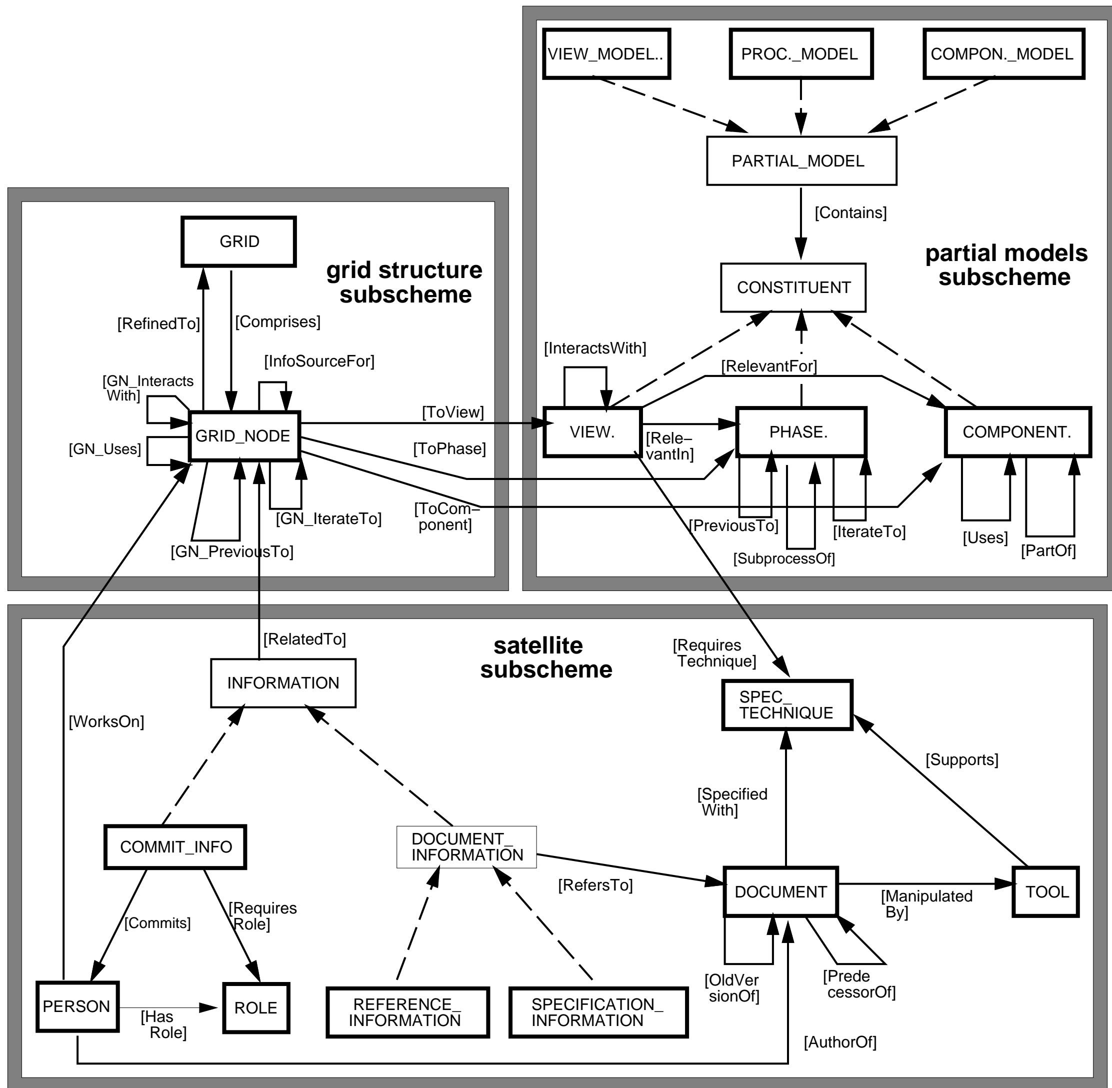

Figure 3: (A part of) the *3DM* graph scheme

The **partial models subscheme** contains the node classes and edge types that are needed to construct the partial models for the three constituting areas of concern. As explained in section 2, these partial models concern software processes, (coarse-grain) system architectures, and the collections of views onto the system.

The graph scheme for these partial models is kept generic and simple, providing subclasses of a a generic **constituent node class** (`CONSTITUENT`), plus a collection of edge types to relate the nodes within each partial model. The `...MODEL` node classes are anchors for the collections of constituents of one partial model, using the inherited `Contains` relationship (defined between the superclasses `PARTIAL_MODEL` and `CONSTITUENT`). Table 1 shows the elements of the subscheme of partial models.

Although this subscheme supplies only simple set of elements to construct partial models, it is possible to construct - with some adaption and abstraction - a large variety of instances of the models for the three areas of concern. Figure 4 shows a (simplified) model for an evolutionary

Dimension:	Constituent nodes (node classes):	Relationships (edge types):
phases	PHASE, PROCESS_MODEL	PreviousTo, IterateTo, SubprocessOf, Contains
components	COMPONENT, COMPONENT_MODEL	PartOf, Uses, Contains
views	VIEW, VIEW_MODEL	InteractsWith, Contains

Table 1: Node classes and edge types of the partial models subscheme

software process, modeled as partial *3DM* model.

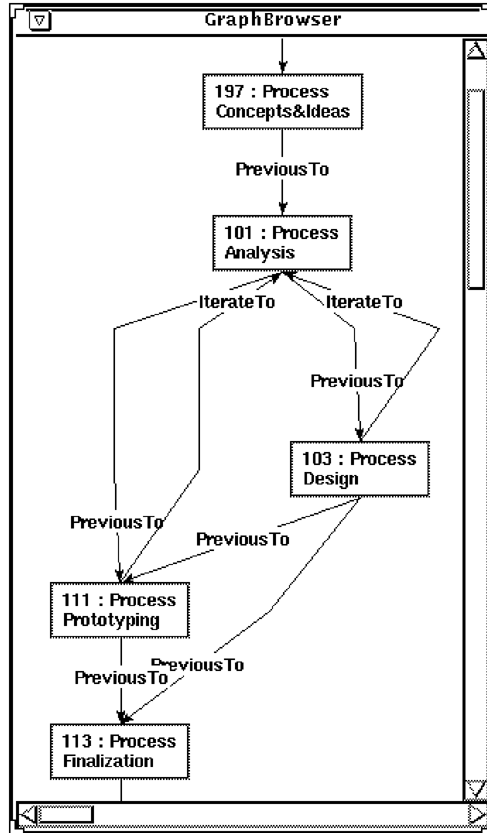

Figure 4: An detail of a partial *3DM* model of an evolutionary software process

.

The partial models are *project-independent*, i.e., they can be reused or adapted for different projects. It is conceivable to keep and maintain a *library of partial models*, from which suitable instances can be selected (and eventually adapted) for individual software projects.

The selected or newly defined partial models are then used to generate an *integrated, project-specific framework* for the actual software project (cf. sec. 5).

The **grid structure subscheme** contains the node classes and edge types which are used to structure and build the integrated project frameworks (cf. fig. 3). GRID) is the anchor for a set of software engineering fragments (GRID_NODE) and related to them by Comprises edges. Grid nodes are identified by a combination of constituents from each of the three dimensions to which

they are linked by the three edge types ToView, ToPhase, and ToComponent. The grid nodes are also mutually related by the same kinds of relationships that relate the constituents of the partial models (e.g., GN_PreviousTo[5]).

As logical unit, each software engineering fragment can be *decomposed* into an own sub-framework (cf. (edge type RefinedTo). A sub-framework may be built from different partial models than its "parent" framework. This decomposition mechanism allows to "zoom in" on particular details - fragments - of the project, and to work on them according to different partial models than used for the rest of the project.

The **satellite subscheme** (cf. fig. 3) captures all information that does not contribute to the "topological" structuring of the project-specific graphs. Node classes of the satellite subscheme provide information about project- and team-specific resources and constraints (e.g., PERSON, DOCUMENT, TOOL, SPEC(ification)_TECHNIQUE) which is usually related to one or more software engineering fragments. Using different subclasses of the node class INFORMATION (e.g., REFERENCE_INFO about reference documents, manuals, etc., or COMMIT_INFO about commitment of persons (in a certain role) to fragments), this project- and developer-related information can be explicitly linked to specific grid nodes.

5 The Dynamic Part of 3DM

The dynamic part of *3DM* defines all actions with which *3DM* graphs can be generated, manipulated, navigated through, etc.. As mentioned in section 3, PROGRES distinguishes a number of concepts to realize graph manipulation operations. Low level rewriting operation are called *productions* in PROGRES, and can be used as building blocks for more complex manipulation operations, called *transactions*. Before we describe the dynamic part of *3DM* in more detail, we give two short examples of manipulation operations, in order to illustrate how the dynamic part of *3DM* - manipulation operations on the static structure, rules and constraints - can be formalized with PROGRES.

The PROGRES production Commit_Person (cf. fig. 5) specifies how to commit a person to a software engineering fragment, i.e., how to relate the node representing that person to a commitment information node related to the grid node representing the fragment. The production has three *formal parameters*: the commitment information node (commit_info), the person node involved (person), and the amount of resources requested from that person for that commitment[6]. The left side of the production (the first of the two dotted rectangles) and the condition clause constrain the action of committing a person to a software engineering fragment in a number of ways, by defining a (unique) pattern that has to be matched in the current host graph. Numbers in the graphical part of the production are used to identify the different nodes:

1. The input parameter nodes commit_info and person have to be present in the host graph, and may not be already related by a Commits edge (crossed-out arrow).
2. There may not exist another node of type COMMIT_INFO that is related to the same grid node (node '3) and that commits the given person *in the same role*. This is ensured by the so-called *restriction* "valid..." which points to node '4. It states that the attribute values Requires_role of the two commitment information nodes '4 and commit_info ('1) have to be the same. The *negative node symbol* (crossed-out node) demands that "there is *no* node

[5] GN_... stands for **G**rid **N**ode ...

[6] For the purpose of this example, "resource" is only an abstract integer, but in *3DM*, it will be possible define values as man-hours, or full-time equivalents as "resource".

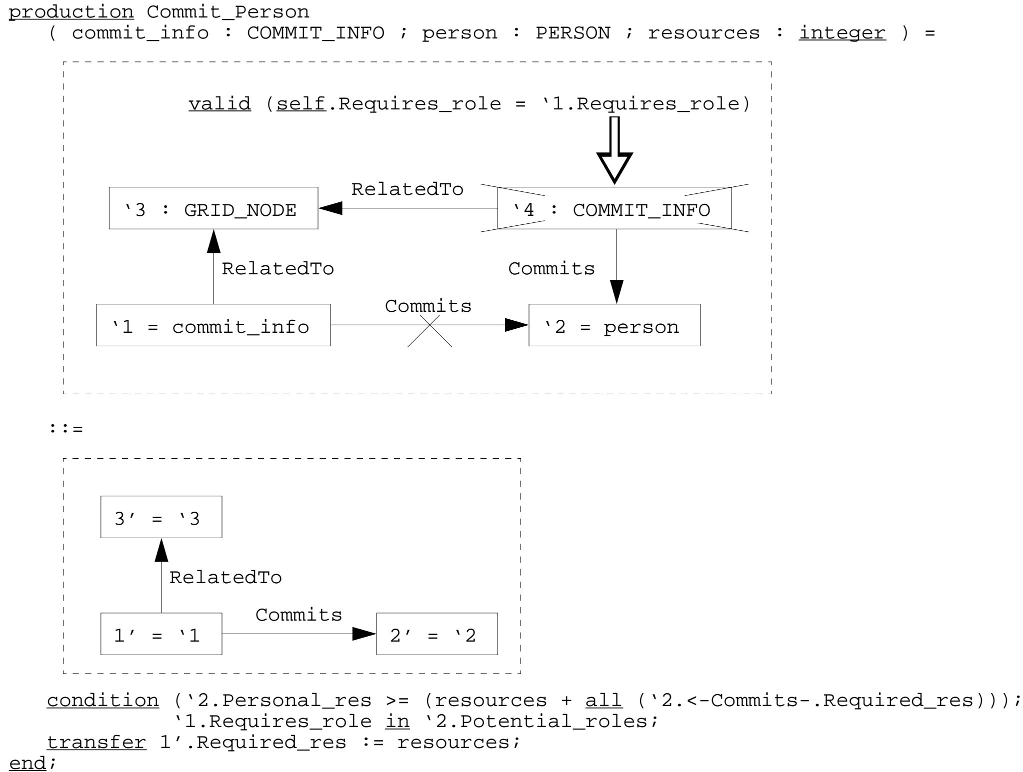

Figure 5: The PROGRES production `Commit_Person`

where the condition is valid". Therefore, the semantics of that construct is that it is *not* allowed to commit the same person several times to one software engineering fragment in the *same* role.

3. The role required by the commitment (`‘1.Requires_role`) has to be one of the (potential) roles of the requested person (`...`<u>`in`</u>` ‘2.Potential_roles`).
4. The sum of the resources employed by the requested person plus the resources requested for the new commitment (`resources + `<u>`all`</u>` ...`) has to be less or equal to the personal resources of the person (`‘2.Personal_res`). The sum of resources committed to other software engineering fragments is calculated by making use of the *path expression* (`‘2.←Commits-`) from the person node *backwards* to other commitment information nodes.

If all constraints are fulfilled, then the sub-graph matching the left side of the production is replaced by the sub-graph defined on the right side. In this case, only the requested `-Commits→` edge is added, and the requested amount of resources stored in the appropriate node attribute (<u>transfer</u> `1’.Requires_res ...`).

To realize more complex graph manipulations, PROGRES provides transactions which include imperative control structures (e.g., loops, if-then-else), and allow to define atomic sequences of graph transformations. Figure 6 shows an example of a transaction involving the production `Commit_Person`, presented above.

`DEFINE_COMMIT_INFO` creates a commitment information node and attaches it to the desired grid node (input parameter `grid_node`). First, a node of type `COMMIT_INFO` is generated using

```
transaction DEFINE_COMMIT_INFO
   ( grid_node : GRID_NODE ; person : PERSON [0:1] ; req_role : Role ;
     resources : integer )                                            =
   use local_com_info : COMMIT_INFO
   do
        Define_CommitInfo
        ( grid_node, description, req_role, resources, out local_com_info )
     & choose
        when (def ( person ))
        then
           Commit_Person ( local_com_info, def elem ( person ), resources )
        else
           skip
        end
   end
end;
```

Figure 6: The PROGRES transaction DEFINE_COMMIT_INFO

the production Define_CommitInfo (not shown). Define_CommitInfo searches for the software engineering fragment (grid_node) in the host graph to which the new commitment information node has to be attached. If the requested node is not found, the production fails, i.e., no re-writing will take place, and consequently the whole transaction fails. Otherwise, a new node of type COMMIT_INFO is generated, and local development information is transferred to the respective attributes of the commitment information node.

The sequence of operations within a transaction is separated by "&"s. The assignment of a person to a commitment information node is not obligatory (...PERSON [0:1]), the responsible person might be assigned later. If a person has been specified (**when def (person)**), then Commit_Person relates person and commitment information in the way described above. Otherwise this step is skipped.

Analogue to the static part of *3DM*, its dynamic part is classified into several categories, too. A coarse overview of how the *3DM* is applied to *structure and guide a software project* from a technical point of view reveals these categories.

A project-specific framework is set up by actions grouped together into the category of **preparation actions**. A first step is to *specify the partial models* for the three dimensions of *3DM*, as well as all *project-specific information and constraints* known at the starting point of the project (and captured by the "satellite part" of *3DM*). To define partial models structures and constrains the project concerning the three areas of concern. Partial models and project parameters are usually described in early project documents, as e.g., the software project management plan (SPMP) or the software quality assurance plan (SQAP). These base documents can be used as source/reference when modeling the partial models for processes, architectures, and views that can be obtained:

- by adopting "standard" (text book) models
- by designing (new) partial models which suit the individual development situation
- by reusing suitable partial models from previous software projects

in *3DM*- usually by a software quality assurance officer or the technical project leader.

Figure 7 shows an example of a preparation action, defined as PROGRES production. With Define_ProcessIterateTo, two phases of the process dimension can be related by an -IterateTo→ edge. This allows iterating (back) from one phase (phase) to another phase (predecessor) in the software process defined by this model. A number of constraints is checked before allowing the addition of the new edge:

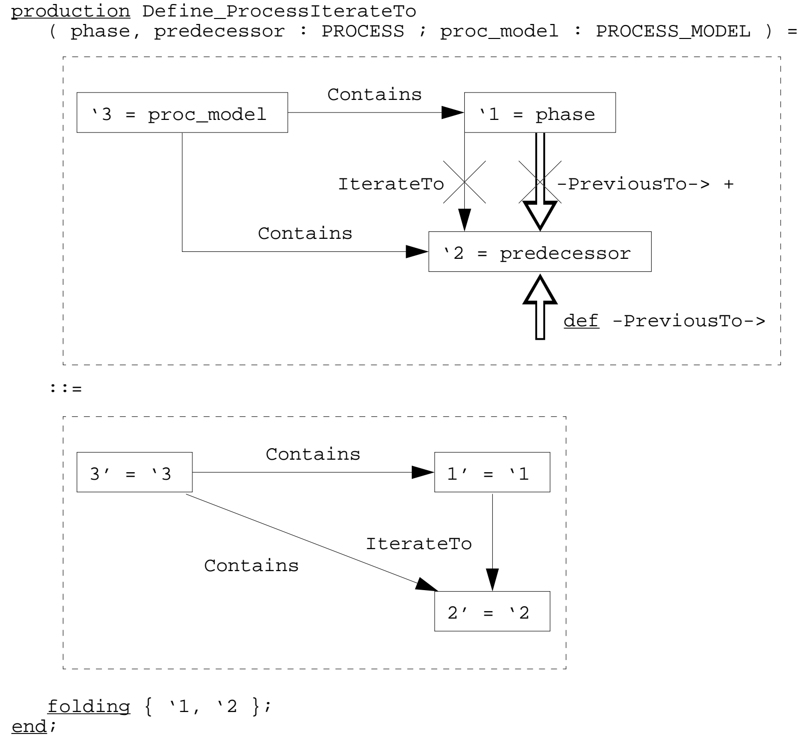

Figure 7: The PROGRES production Define_ProcessIterateTo

1. Both phases have to be part of the same partial model (-Contains→ edges from the same process model anchor proc_model).
2. The two phases are not already connected by an -IterateTo→ edge (crossed-out -IterateTo→ edge symbol).
3. The two phases are not connected by a non-trivial path of -PreviousTo→ edges. If such a path existed, adding an -IterateTo→ edge would allow to iterate back to a phase that is also (transitive) *successor* of the source node of the -IterateTo→ edge. The sequence of at least one -PreviousTo→ edges is specified by the path symbol (the thick arrow) labeled -PreviousTo→ +, its non-existence by crossing it out.
4. At least one -PreviousTo→ edge has to start from the phase to which iteration should be allowed (predecessor), in order to prevent iteration to a "dead end". This is specified by the restriction <u>def</u> -PreviousTo→.
5. The <u>folding</u> clause allows the two phases to be identical, i.e., to match *one* node in the *3DM* host graph, allowing to direct iteration to the same fragment.

After defining at least three partial models (one for each dimension), a *draft project framework is automatically generated*. Figure 8 shows the result of the construction process of a project framework from partial models. In order to illustrate this process clearly, the whole example is kept simple and abstract. The basic principle of framework generation is to create a software engineering fragment (i.e., a grid node) for every possible triplet of combinations involving one constituent per dimension. Following strictly that principle would generate a perfectly regular three-dimensional grid (cf. fig. 8, small drawing on the right).

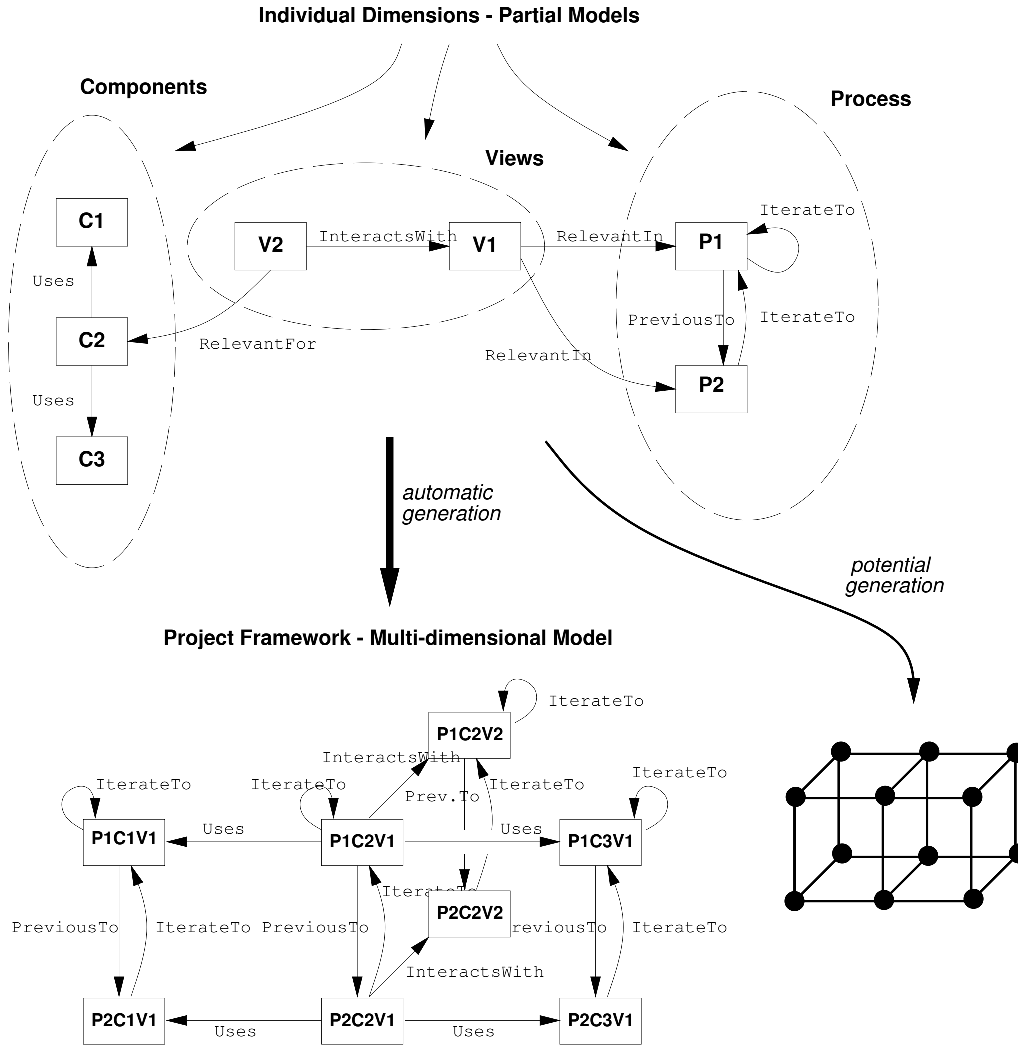

Figure 8: Relating partial models and generation of an draft project framework

But this straightforward approach would yield (too) many, probably superfluous, fragments[7]. Therefore, the generation of software engineering fragments is steered manually by an *intermediate step* in which views are explicitly related to *certain* components and/or to certain phases by the person who is responsible for the technical set-up of the project. `-RelevantFor→` and `-RelevantIn→` edges are defined to indicate the relevance of a view for either a certain (set of) component(s) and/or a certain (set of) phase(s) (cf. fig. 8, upper part).

Then, only those software engineering fragments are generated for which at least one of the two constituents of the arbitrary combination of a component and a phase is related to a view. This limits the number of software engineering fragments generated at project start, and thereby avoids necessary deletion of insignificant fragments *during* the course of the project. Figure 8 (lower part) shows the resulting *draft project framework*, generated from the partial models defined in the upper part. After generating the fragments, also the *relationships* between constituents of the

[7]Partial models of only 5 phases, 3 views, and 5 components would result in a project framework with 75 software engineering fragments.

partial models are taken over from the partial models to the draft project framework. Relationships between two constituents of the partial models are "copied" to all pairs of grid nodes which incorporate these two constituents in their structuring information.

To complete the set-up for the actual project, the project-specific information, constraints, resources, etc., that have been stored in the "satellite" part of the *3DM* structure, is explicitly linked to the appropriate software engineering fragments.

Once the project-specific graph has been generated, and development has started, **manipulation actions** enable manipulation of the project graph to reflect and to adapt to the actual development situation, while preserving its consistency.

Most common is to add, change, or delete "satellite nodes" (e.g., specification documents, persons, roles). But also the grid structure of the project can be changed, itself. The introduction of new components, phases, or views, that have not been part of the partial modeling and therefore also not reflected by the draft project framework initially generated, makes it necessary to alter the structure of the project specific graph in a controlled way. Figure 9 shows the PROGRES specification of a graph manipulation that *merges* two grid nodes.

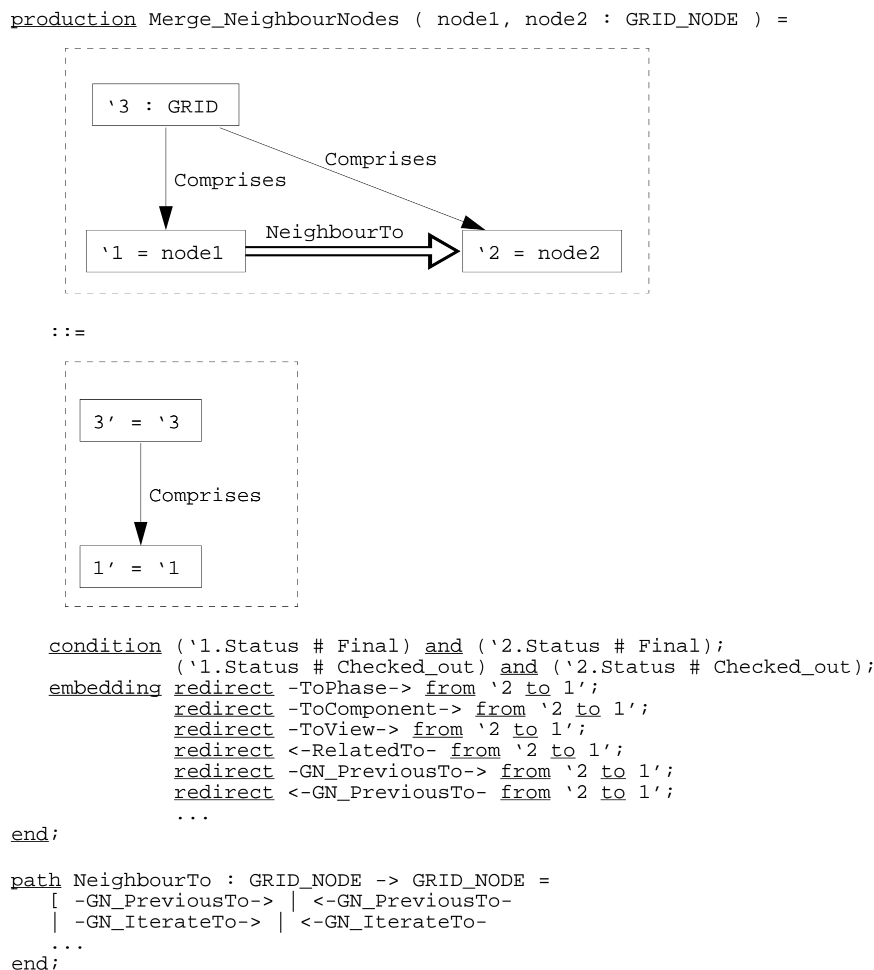

Figure 9: The PROGRES production Merge_NeighbourNodes

The two grid nodes node1 and node2 can be merged if they are direct neighbors, i.e., if they are

connected by one of the edges listed in the path `NeighbourTo`. This constraint is imposed on the merging process to prevent merging arbitrary, completely unrelated nodes of the project framework. Allowing only to merge neighbored nodes preserves the clearly laid-out structure of the grid, and ensures that only related fragments (and with them, the development tasks) are combined into one logical unit. Furthermore, the revision status of the two nodes to merge may not be `Final` (in that case, fragments and information attached should not be changed anymore) or `Checked_out` (currently edited by a team member).

If the conditions for merging are fulfilled, all outgoing and incoming edges (including the development information linked to by `-RelatedTo`→ edges) of grid node `node2` are *redirected* to the remaining grid node `node1` (`embedding...` clause), while grid node `node2` is deleted (it doesn't appear on the right side of the production).

As long as concerning individual software engineering fragments, evolution of the project graph and manipulation of the "satellite" information is due to individual developers. But merging, deleting, or adding fragments, i.e., changing the structure of the project graph, are tasks that have to be performed by project members with a higher responsibility (e.g., by the technical project manager, by a software quality assurance officer, or by a method engineer).

The last two categories of actions will only be mentioned briefly. They are also part of the formally specified dynamics of *3DM*.

As stated in section 2, the activities of an individual team member are captured by the sequence of software engineering fragments he works on. **Navigation actions** relate the project team members to software engineering fragments, i.e., person nodes representing them to the respective grid nodes (by `-WorksOn`→ edges - cf. fig. 3). Navigation actions determine the activities of the software developers by constraining the selection of fragments depending on the structure and status of the actual project graph and its nodes.

Finally, **information retrieval actions** are used to query information about the project graph, without changing it. These actions serve two purposes:

1. as part of navigation and manipulation actions, they are used to determine the validity of preconditions and the context of graph-changing transaction,
2. to obtain project information to be used outside the scope of *GRIDS* for project management purposes.

Examples for actions of this category are the *retrieval of the set of grid nodes related to a certain person node*, in order to determine the software engineering fragments that team member currently works on, or the *generation of an overview of grid node statuses* to determine the progress of the project.

6 Tool Support

As mentioned in the introduction, large-scale software development requires tool support. This holds also for a project framework based on a formal specification. The users, i.e., the members of the development team, want to specify and use integrated project frameworks, based on the *3DM*, in an easy way, exploiting the benefits of correct and consistent specification without having to deal with the details of the underlying formalism and the formal model of the *3DM*.

PROGRES provides two possibilities to execute a given specification. The first one is integrated in the PROGRES environment itself, and is based on **direct interpretation** of the specification.

The main purpose of this first possibility to execute specifications is for debugging and testing the specification under development The lack of performance and a sophisticated user interface prevent this option to be used as tool support for software projects.

The second possibility is to translate specifications into equivalent Modula-2 or C code, and to compile the generated source code together with the graph DBMS GRAS and the user interface toolkit TCL/TK. Apart from the sophisticated performance, the resulting **rapid prototype** offers more advantages and features that are essential for the project-supporting tool envisioned.

- While the formal specification of *3DM* is very large and complex, only a small subset of the specification, the so-called *top-level transactions*, is supposed to be accessible by its users, the software developers. When generating the rapid prototype, it is possible to make those top-level transactions available to the users, and encapsulate the rest of the specification items (comparable to public and private methods in object-oriented programming).

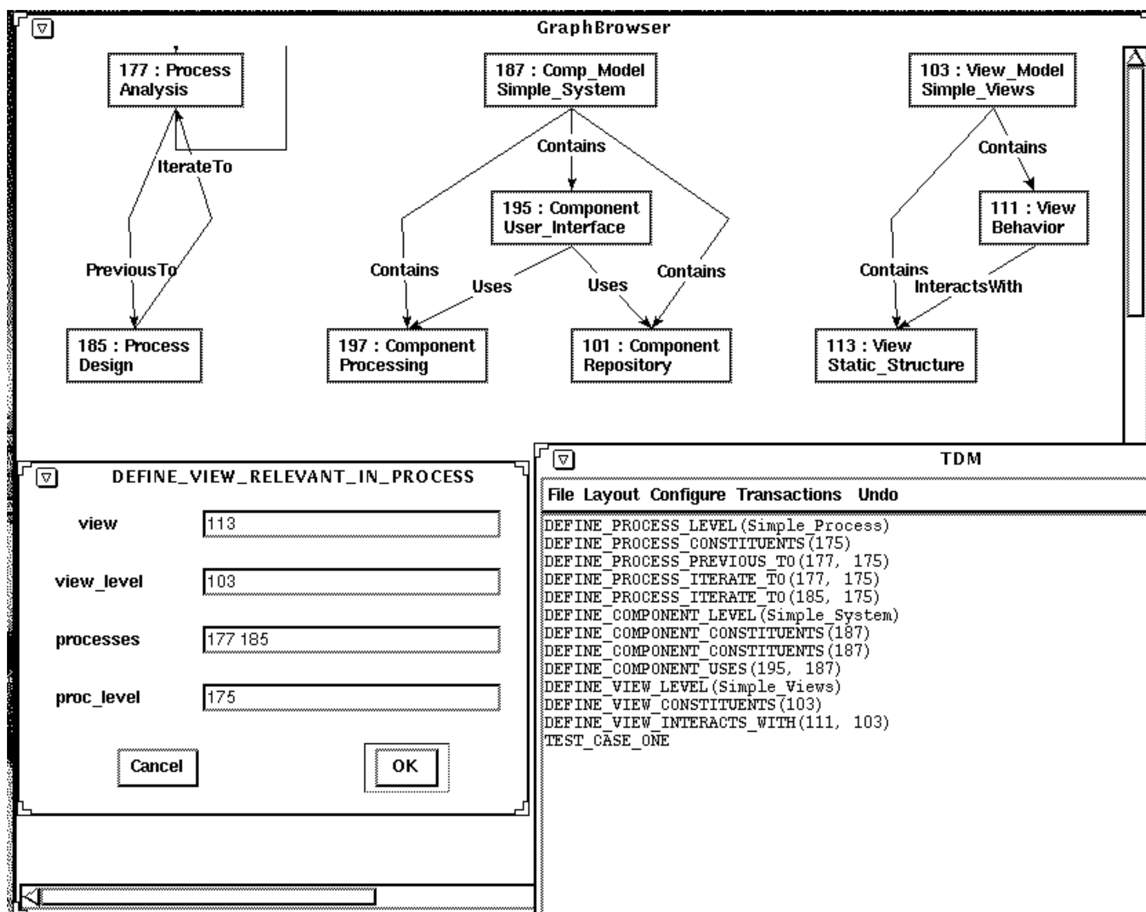

Figure 10: A snapshot of the *3DM* rapid prototype

- The main window of the user interface displays the actual host graph for the project, i.e., the current instance of the *3DM* graph class. The user can, at any time, work with graphically displayed project frameworks, taking advantage of the visualization of the graph as (three-dimensional) grid structure.

  Figure 10 shows a snapshot of the user interface of the rapid prototype. In the *Graph Browser* window, the current project graph is displayed. The sequence of *3DM* transactions that led

to that graph is displayed in the *TDM* window. Three partial models have been specified, and now the transaction `DEFINE_VIEW_RELEVANT_IN_PROCESS` has been selected from the pull-down menu *Transactions* in the *TDM* window. The pop-up window on the lower left is used to enter the necessary parameters for the transaction, in this case a set of node identifiers that can be either typed in or selected by clicking on the appropriate nodes in the *Graph Browser* window.

- To control visibility of and access to the different parts of the project framework, it is convenient and possible to define *configurations* on the project graph. In that case, not the whole graph is displayed, but only the instances of a certain, predefined selection of node classes, e.g., only the partial models, or only the project grid structure without satellite nodes.

Other features are currently added to the generation mechanism of the prototype. These include for example different layout algorithms for displaying the host graph, and a context-sensitive selection of alternatives offered to the user. On the one hand, after clicking on a (set of) node(s), only those transactions will be offered which fit to the selected nodes as input parameters. On the other hand, after selecting a transaction, it will be possible to have all those nodes highlighted that fit as parameters of the selected transaction.

7 Related Work

A number of approaches try to enhance development of large, heterogeneous software systems by integrating different aspects of software engineering.

Approaches like *MetaEdit* [SLTM91], *PPP* [GLW91], *ToolBuilder* [Ald91] or *Fusion* [CAB+94] aim at giving the developers more freedom in their choice of specification techniques by integrating techniques from different *software engineering methodologies*, in order to offer alternatives when the specification techniques of one methodology fail to cope with

the requirements of the actual software project. *Meta-models* are used to uniformly and formally describe the different techniques, and tools are offered to support the (meta-)developers to tailor their individual methodology. But this is only a first step of integration, as it only deals with one area of concern, i.e., the views onto the system.

Method base [SIWyS93] goes one step beyond, offering a formal meta-model (based on Object-Z) that integrates specification techniques (views dimension) *and* specification process (process dimension).

ViewPoints [NF92] finally integrates five different areas of concern: "style" (specification techniques), "work plan" (description of activities), "problem domain", "specification" (document), and "work record" (development history and state). The resulting concept of "*loosely coupled, locally managed objects, encapsulating representation knowledge, development knowledge and specification knowledge of a particular problem domain*" is very similar to the one of *3DM*'s software engineering fragments. The notion of "actions" to generate and manipulate the framework is known, too. While also offering a tool prototype (*TheViewer*), ViewPoints is not based on a formal specification (yet).

8 Conclusion

In recent years, *TNO Institute of Applied Geoscience* has developed a number of very successful software systems to manage exploration & production (E&P) data. Experiences have shown that on the one hand it is very difficult (and doesn't lead to good software systems) to organize software development by defining rigid standard models prior to project start, and then trying to stick

to them, especially when dealing with innovative, experimental software systems [ZG94]. On the other hand, a lack of thorough conceptual modeling, not only of the system, but also of the process, inevitably leads to a lack of necessary clearness and consistency during the course of the project. This usually results in a qualitatively inferior system at the price of an exceeding project schedule.

The *GRIDS* project takes these two extremes into account, and offers a formally based, multi-dimensional software engineering model, the *3DM*, that integrates different areas of concern of software development. While the important areas of concern - process, components and views - can be modelled individually and independently, they are integrated into consistent frameworks for the actual projects. While tailorable, reusable (in its partial models), and adaptable (to changes in the actual development situation), *3DM* leads to a clearer structuring of the development efforts, its logical units and their dependencies, and therefore also to a better understanding of software development in general.

References

[Ald91] Albert Alderson. Meta-CASE Technology. In Albert Endres and Herbert Weber, editors, *Software Development Environments and CASE Technology, European Symposium*, LNCS 509, pages 81–91. Springer, 1991.

[AM92] Vincenzo Ambriola and Carlo Montagnero. Oikos at the Age of Three. In *Proceedings of the Second European Workshop on Software Process Technology (EWSPT '92), Trondheim, Norway*, LNCS 635. Springer, September 1992.

[BFG93] Sergio Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process Modeling-in-the-large with SLANG. In *Proceedings of the Second International Conference on the Software Process*, Berlin, Germany, February 1993.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[CAB+94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes, editors. *Object-Oriented Development: The FUSION Method.* Prentice-Hall, Inc., 1994.

[Eng92] Gregor Engels. Graphs as Central Data Structures in a Database Design Environment (Abstract). In P. Klint, Th. Reps, and G. Snelting, editors, *Programming Environments*, March 1992. Dagstuhl-Seminar Report 34.

[FkNO92] Christer Fernström, Kjell-Håkan Närfelt, and Lennart Ohlsson. Software Factory Principles, Architectures, and Experiments. *IEEE Software*, pages 36–44, March 1992.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In David Wile, editor, *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94)*, Software Engineering Notes, pages 175–188. ACM Press, 1994.

[GLW91] Jon Atle Gulla, Odd Ivar Lindland, and Geir Willumsen. PPP: An Integrated CASE Environment. In *Proc. of the 4th Int. Conference on Advanced Information Systems Engineering (CAISE '91)*, LNCS 498, pages 194–221. Springer, 1991.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[NF92] Bashar Nuseibeh and Anthony Finkelstein. ViewPoints: A Vehicle for Method and Tool Integration. In Gene Forte, Nazim Madhavij, and Hausi Müller, editors, *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering (CASE '92)*, pages 50–60. IEEE Computer Society Press, 1992.

[Poh92] Klaus Pohl. The Three Dimensions of Requirements Engineering. Technical Report 92–11, RWTH-AAchen, Informatik V, 1992. (NATURE Report Series).

[Pot93] Colin Potts. Software Engineering Research Revisited. *IEEE Software*, pages 19–28, September 1993.

[PW92] D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *ACM Sigsoft, Software Engineering Notes*, 17(4):40–52, October 1992.

[RBP+91] James Rumbaugh, Michael Blaha, William Premberlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[Rek94] J. Rekers. On the Use of Graph Grammars for Defining the Syntax of Graphical Languages. In *Proceedings of the Colloquium on Graph Transformation*, Palma de Mallorca, Spain, 1994. (Available by *anonymous ftp* from `ftp.wi.leidenuniv.nl` in `/pub/cs-techreports` as `tr94-11.ps.gz`).

[Sch91] Andy Schürr. PROGRES: A VHL-Language Based on Graph Grammmars. In *Proc. of the 4th Int. Workshop on Graph-Grammars and their Application to Computer Science*, LNCS 532, pages 641–659. Springer, 1991.

[SIWyS93] Motoshi Saeki, Kazuhisa Iguchi, Kuo Wen-yin, and Masanori Shinohara. A Meta-Model for Representing Software Specification and Design Methods. *Information System Development Process*, pages 149–166, 1993.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. MetaEdit - A Flexible Graphical Environment for Methodology Modeling. In *Proc. of the 4th Int. Conference on Advanced Information Systems Engineering (CAISE '91)*, LNCS 498, pages 168–193. Springer, 1991.

[SO92] X. Song and L. Osterweil. Towards, Objective, Systematic Design-Method Comparisons. *IEEE Computer*, May 1992.

[TDK94] Joachim Tankoano, Jean-Claude Derniame, and Ali B. Kaba. Software Process Design Based on Products and the Object Oriented Paradigm. In *Proc. of the 3rd European Workshop on Software Process Technology (EWSPT '94)*, LNCS 772, pages 177–185. Springer, 1994.

[Wes92] Bernhard Westfechel. A Graph-Based Approach to the Construction of Tools for the Life Cycle Integration between Software Documents. In *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering*, pages 2–13. IEEE Computer Society Press, 1992.

[Zam94] Andreas Zamperoni. Integration of the Different Elements of Object-Oriented Software Engineering into a Conceptual Framework: The 3D-model. Technical Report 94-18, Leiden University, Dept. of Comp. Science, 1994. (also available by *anonymous ftp* from `ftp.wi.leidenuniv.nl` in `/pub/cs/techreports` as `tr94-18.ps.gz`).

[ZG94] Andreas Zamperoni and Bart Gerritsen. Integrating the Developers' and the Managerial Perspective of an Incremental Development Life Cycle. In *Proc. of the 12th Annual Pacific Northwest Software Quality Conference, Portland, USA*, pages 227–242, October 1994. Also as Technical Report 94-22, Leiden University, Dept. of Comp. Science, The Netherlands (Available by *anonymous ftp* from `ftp.wi.leidenuniv.nl` in `/pub/cs-techreports` as `tr94-22.ps.gz`).