Techn. Report AIB 94-2,
RWTH Aachen, Jan. 1994

Development of Incremental Integration Tools Using Formal Specifications

Martin Lefering
Lehrstuhl für Informatik III, RWTH Aachen
Ahornstr. 55, 52074 Aachen
martin@i3.informatik.rwth-aachen.de

Abstract

The integration of documents is one of the key problems in software engineering today. This paper presents an approach for integrating documents that were created in different phases of the lifecycle by using integration tools. These tools execute a transformation algorithm that is expressed by a transformation rule set. They are based on a generic framework that can be reused for new prototypes without modifications.

Formal specifications of transformations that support various integration scenarios can be derived automatically out of specifications for the related documents and the definition of correspondences between the productions of these specifications. Three prototypes for integration tools according to this scenario have been developed.

Keywords

Software development environments, lifecycle integration, formal specification of tools, rule-based transformation, traceability, reusability.

1 Introduction

Achieving a tight *integration of tools* and documents is one of the major tasks in building software development environments today. Integration of tools can be supported by using a central database, similar behaviour, and a uniform user interface for all the tools in the environment [BFW92, BEM92]. But the *integration of the documents* that are created and manipulated by the tools still is a great problem. Ramesh and Edwards stress the need for consistent documents in the different lifecycle phases [RE93]. Traceability between documents of the same lifecycle phase (horizontal integration) and of different lifecycle phases (vertical integration) is necessary.

In the IPSEN research project at the Aachen University of Technology [Nag94], we achieve the integration of various documents by using so-called *integration tools*. All documents are modelled as graphs and stored in a special central database called GRAS (GRAph Storage) [KSW93]. The integration tools fulfil mainly two tasks: They execute a certain *transformation* from source documents into target documents and connect the transformed parts of the documents by so-called interdocument links. These links are used further on to *check the consistency* of the documents. After an inconsistency occurs, e.g. because one of the documents was edited, they detect this inconsistency and can mark inconsistent parts or even try to repair the situation by executing incremental repair transformations.

At the moment, we have *three different integrators* in our environment. The first one integrates three different documents in the requirements engineering phase (horizontal integration). These are a functional model that consists of data flow diagrams, an information model consisting of entity relationship diagrams, and a control model using state transition diagrams. The second integrator transforms such a requirements specification into a design document and checks the traceability of the information in these documents. The third example is an integrator that generates templates of Eiffel code (i.e. implementation documents) out of the module specifications that are part of a design document.

Though the integrators slightly vary in their tasks and supported scenarios (e.g. the first only checks consistency, the second also generates an initial design and the third even protects the generated code against manual modifications - you have to change the design instead), they have mainly the same functionality. Therefore, we developed a *generic framework* for those integration tools to extract the common parts. All the integrators are instances of this framework. The main parts of the framework as shown in Fig. 1 are a transformation table that embodies the transformation rule set, an integration document where the installed interdocument links are stored, and a control component that defines the scenario an starts the different actions. Control part, transformation table, and integration documents can be reused for every new instance of an integration tool, only the transformation rules have to be rewritten. The concepts of integration tools and framework are described in detail in [Lef93].

An interesting feature of the integrators is that the *transformation rule set* need not be unique. This means that several transformations are possible for a certain element of the source document. In this case, the user has to choose the appropriate transformation. Hence, in general, the integrators are semi-automatic tools. Only if the rule set is unique it is a fully automatic one.

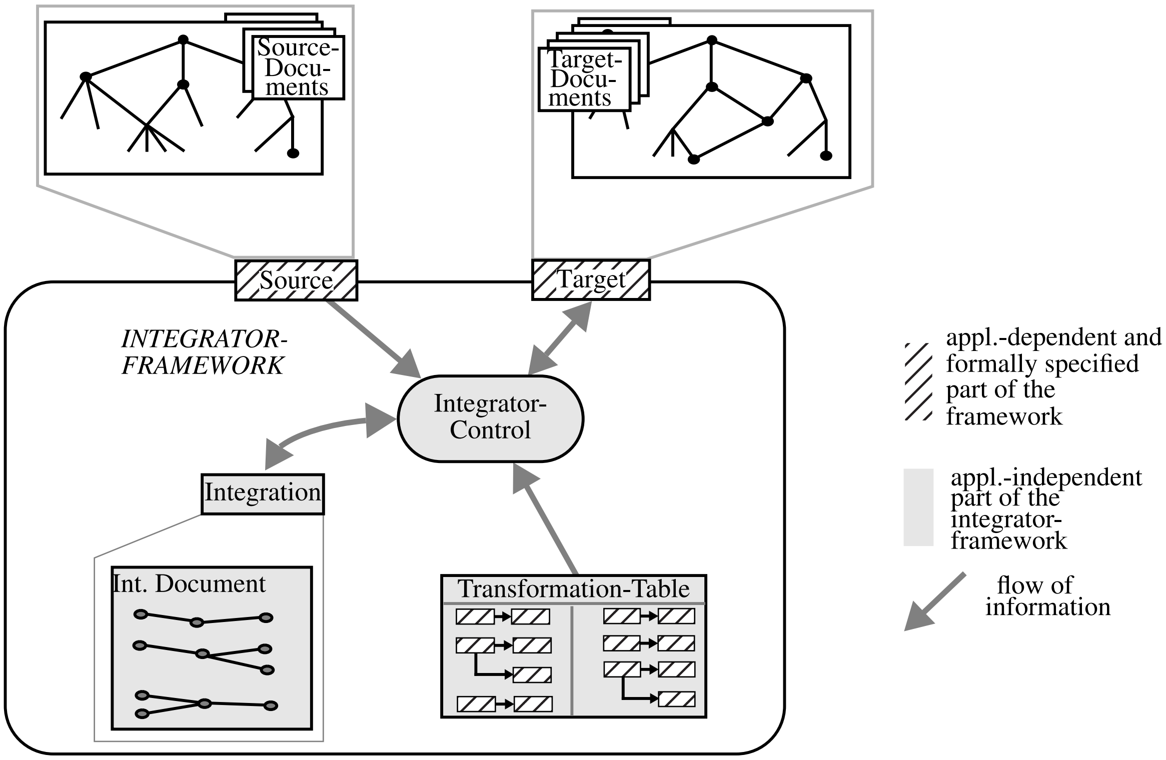


Fig. 1: *Main components of an integrator (overview)*

The user may also draw additional links completely manual, for instance if he wants to establish correspondences without changing the transformation rules.

As mentioned above, the transformation rules have to be formulated newly for every new integration tool. To avoid coding those rules from the scratch for every new example, we use a *specification* technique to describe the transformation on a very high, abstract level in a declarative way. The executable code can afterwards be derived from this specification.

Other approaches where transformer generators have been developed in the context of software development environments are TransformGen [GKS87] and Muir [Nor87]. In TransformGen a transformer between different revisions of documents is built automatically, while the user is editing changes in their EBNF. In Muir a transformer between PiS documents in different languages is generated from a formal description of the abstract syntax. Both approaches try to generate automatic batch transformers and are therefore restricted to a very small field of application areas, because the descriptions of the languages to be integrated (e.g. EBNFs) must not differ a lot.

Our approach can also be compared to several so called “CASE Frameworks”, which have the aim to support integration of existing software development tools on different levels. The purpose, however, is different: In IPSEN we try to simplify the realization of new tools. The profit is a tight integration on tool and data level.

Other approaches are either restricted to the control integration level like FIELD [Rei90], HP’s SoftBench [Ger89], and Forest [GI90] or concentrate on the data integration level like PCTE [Bou88, WJ93] or IBM’s AD/CYCLE [IBM89]. The first use a broadcast message server to make single tools communicate with each other using a certain protocol. Shortcomings of this approach are that the tools have to be modified to understand that protocol and that no integration on the data level is supported, i.e. no tool understands the other tools' documents and, consequently, cannot refer to them. On the other hand, it is not sufficient for the tools to operate on the same repository (i.e. data integration), but they also have to work in an integrated manner.

Platforms, where the different aspects of integration are combined, are DEC's COHESION [Wel91] or Atherton's Software Backplane [Asl91]. But these approaches simplify the integration only on the coarse-grained (i.e. document) level but are not appropriate to support a tight integration on the fine-grained (i.e. increment) level.

This paper introduces a way to describe a mapping between different documents in a formal specification language and to generate a specification for the transformation from it automatically. The resulting specification can be translated into executable code. For this purpose, a formal specification language called PROGRES (PROgrammable Graph REwriting System) has been developed [Sch91], including a compiler that generates Modula-2 code for a specification. In the following, we describe how a mapping between two languages can be specified and how various transformations can be generated out of it. Here, we concentrate on the transformation of data flow diagrams into module specifications as a part of the second integrator mentioned above.

A more detailed description of the requirements definition language can be found in [Jan92], the design language is described in [Bör93] and the transformation is explained [BJ92] and [Lef93].

2 Functionality of framework-based tools

In this section we describe the scenario of integration tools that are developed with the framework introduced in section 3.

The possible functionality of integrators based on the new framework is essentially the union of the functionalities of the former integration tools (hence, these tools can be rebuilt within the framework): they realize an exchangeable and adaptable transformation algorithm, draw links between related source and target increments for consistency checks and browsing (so-called interdocument links), are able to repair consistencies, allow to draw links manually, and support user interaction. The links are collected in a so-called *integration document*.

The integrators realized with the framework work according to the following *integration process*: When an integration is started, the tool searches for inconsistent links in the source documents and in the integration document. Inconsistent links occur when

- a new increment has been inserted in the source document,
- a target increment that had been connected with a source increment in a preceding integration process has been deleted,
- a rule that was applied to transform a certain increment is no longer valid.

If new target increments are inserted, this does not lead to an inconsistency. If an already integrated source increment is deleted, the corresponding link is removed and the related target increment has no connection to a source document and can be changed further on without leading to an inconsistency.

Sometimes the applicability of rules may depend on the former execution of other rules. I.e. such rules expect the existence of certain interdocument links. These dependencies among the rules induce dependencies among the links, which are drawn by the rules, and have to be considered by the integrator (cf. section 3).

As an example consider the following rule for the transformation of entity attributes from the integrator between Requirements Engineering (RE) and Programming in the Large (PiL):

```
IF   exists an attribute type A as a component of an
          entity type B (RE pattern)
     AND entity type B was transformed into a data type module (INT pattern)
THEN transform A into a local data type in the realization part of that
     module.
```

A rule transforms a *dominant source increment* (here: attribute type A), if some *context increments* (here: entity type B) exist. Since this is the search pattern in the source documents, it is called *source pattern* (here: RE pattern). Yet the execution of the example rule does not only depend on the existence of B, but also on its transformation. It is executed only, if B was transformed into a data type module. This is expressed in an *integration pattern* (INT pattern). There are other transformations for entity types, too.

After having collected the inconsistent links the integrator can try to repair them. It can be driven in two modes. In *semi-automatic mode* the tool tries to execute applicable rules to transform the according source increment and make the links complete. This works as long as the rule set is unique, i.e. in every situation only one rule is applicable. If more than one rule is applicable in a certain situation, the integrator asks the user to select a rule. After this user interaction it continues until the next conflict occurs. Thus, in semi-automatic mode the user is asked only in conflict situations. If the rule set is unique, this is even a fully automatic mode. Semi-automatic mode is normally used, when documents are integrated for the first time.

In *guided manual mode* the integrator stops at every inconsistent link to ask the user how to continue. This mode is needed in integrations of already related documents that became inconsistent. The user can take several actions to continue the integration process:

- execute the same rule that has been applied before, if it is still applicable,
- execute another applicable rule,
- mark the source increment as not to be transformed again, or
- draw a manual link to an arbitrary target increment.

By giving the user the responsibility to repair inconsistent links the integrator avoids re-establishing former situations that would have to be changed again. If e.g. in the example above an attribute type A had been transformed to a local data type A in a preceding integration and the designer has deleted this type afterwards, the integrator would recreate data type A again in semi-automatic mode. Thus the modifications made by the designer in the meantime would be ignored. In manual mode the designer has the above listed choices. E.g. he could mark attribute type A as “not to be transformed again”. Then the integrator regards the interdocument link of attribute type A as consistent, though the half-link to the target document is missing.

The integrator uses the dependencies between the rules to determine the order in which it transforms the links (more precise: the dominant source increments of the links). In each step only inconsistent links that are not depending on other inconsistent links can be transformed. Those links are called *safe inconsistent links*. When they are repaired and made consistent other links will become safe inconsistent and can be repaired in the next step.

This integration process offers high flexibility, because all the working steps that are sensible in a certain situation are allowed. It also offers a good guidance, because it prevents the user from doing working steps that possibly have to be redone later, like transforming dependent links.

3 A generic framework for integrators

To unify the architecture of the integration tools we are developing a new *application-independent framework*, which is the same for the integration tools in all examined applications. The single tools are instances of that generic environment.

The development of the framework has two different aspects: finding a common architecture for integrators (with the identification of application-independent parts that can be reused without or with minor modifications in every instance) and identifying application-dependent parts.

The architecture of an integration tool is shown in Fig. 2. There is the part which embodies the transformation algorithm and which is a set of transformation rules, called *transformation table* (TrafoTable).

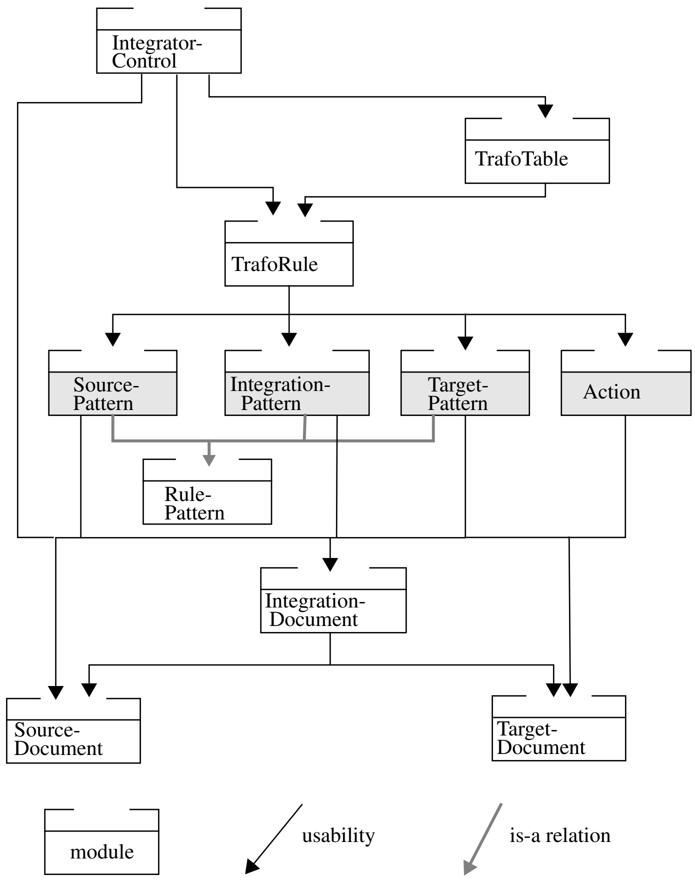


Fig. 2: *Coarse architecture of an integrator.*

Rules are modelled in the following way: The *action* part of a rule is an operation on the *target document* (insert/delete a certain increment) and/or an operation on the *integration document* (insert/delete a certain link). The patterns of the rules consist of different parts: There is a *source pattern* that describes which part of the source document should be transformed. Moreover, the activation of a rule can depend on the integration document as well, e.g. if one wants to describe that a certain source increment should be transformed only if another source increment has been transformed to a certain target increment (cf. section 5). To check this, the integrator has to search for the according link in the integration document. Hence, we need an *integration pattern*. It is also allowed to formulate *target patterns*, if the execution of a rule depends on a certain situation in one of the target documents. But target patterns are not needed in the RE-PiL-integrator. All patterns are subclasses of an abstract class *rule pattern*.

The code for the rules can be generated from a formal specification written in PROGRES. The concept of specification is described in detail in section 5.

The *integration document* stores the *links* between the integrated documents. They are divided in consistent and inconsistent links. This information is needed for the consistency checks that have to be started by the user explicitly. Since the rule set of the integration does not need to be unique, the user has to be asked sometimes to solve a transformation conflict. The *user decision* is stored in the interdocument links in order to have the information available when an inconsistency occurs.

Another information to be stored in the integration document are the *dependencies between links*. Those dependencies occur if the execution of a rule depends on an integration document pattern, i.e. a link that was drawn by another rule execution. A destruction or modification of such a link necessitates a new examination of the dependent link. Therefore, those dependencies are stored in the integration document as dependency links between the interdocument links. Hence, a *dependency graph* of the inconsistent links is built in the integration document.

An overview of the *internal structure of the integration document* is shown in Fig. 3. The links mark the transformation of a certain source increment, called *dominant increment*. Therefore every link has a half-link to the related dominant increment in one of the source documents. Another attribute is a *state* tag that stores the user decision for this link. It indicates whether the link is

- inconsistent,
- consistent by rule execution (in this case the applied rule is stored as well),
- consistent by manual completion, or
- marked as consistent by user definition without connection to target increments (i.e. the dominant source increment is marked as not to be transformed).

Additionally, the inconsistent links (more precisely: inconsistent and not yet transformed links) contain the information, which rules are potentially applicable. In the beginning of the integration process all the inconsistent links are collected. If such a link is detected, the integrator checks immediately which rules might be applicable according to their source document pattern and stores them in the inconsistent link. The execution of these *potential rules* might also depend on the existence of other links (INT pattern) which possibly have not been transformed so far. These INT patterns of the potential rules determine the dependency edges to other consistent or inconsistent links. The dependency graph is acyclic, since no cyclic dependencies among the rules are allowed. To mark the actual match in the source documents, which might be much more than just the dominant increment, pointers to the so-called *context increments* are stored with the potential rules.

An inconsistent link can be repaired by one of the options described in section 2, if it does not depend on other inconsistent links (like link 1, 2, 3, and 4 in Fig. 3). To do this, the set of potential rules has to be updated, because some of them may not be applicable any more due to transformation decisions made in the meantime. For instance link 3 and link 5 in Fig. 3 might have both two potential rules. Rule A in link 5 requires rule A in link 3 to be applied, rule B in link 5 requires rule B in link 3 to be applied. If e.g. rule A of link 3 is applied now by user decision, thus making link 3 consistent, link 5 becomes a *safe inconsistent link*, i.e. a link that is independent of other inconsistent links, and can be repaired. But since rule A of link 3 was applied, rule B of link 5 is no longer applicable. Hence rule A is the only potential rule left in link 5. In semi-automatic mode it would be applied automatically by the tool.

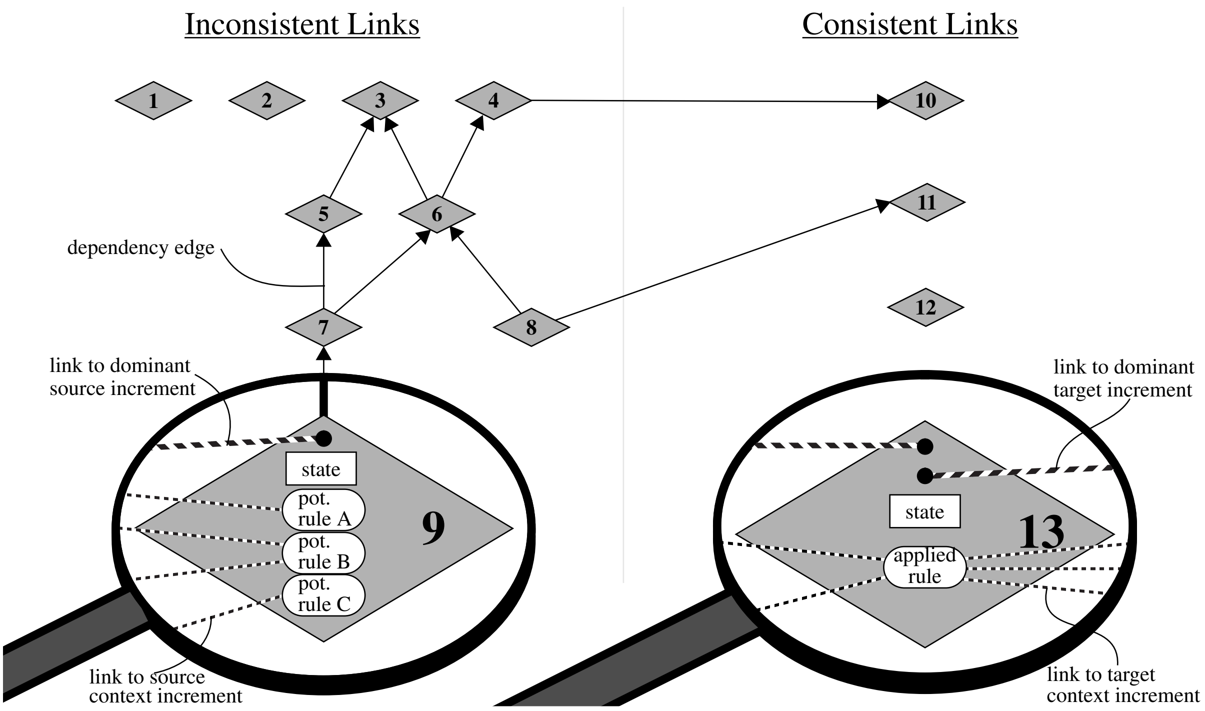


Fig. 3: Int*ernal structure of the integration document.*

After the user or the tool have made a decision how to repair a link, that link becomes a *consistent link*. The applied potential rule is stored as the *applied rule* in the consistent link together with its pointers to context increments in the source document pattern. They help to check the consistency of the link later on. Additionally to the half-link to the dominant source increment, a half-link to the *dominant target increment* is stored as well as pointers to target context increments together with the applied rule.

The dependencies among the consistent links can be forgotten. They are only needed to prevent the tool from repairing inconsistent links that depend on other inconsistent links, because otherwise links might have to be repaired several times during one integration process. Consider the example mentioned above, where link 5 in Fig. 3 depends on the decision made in link 3.

If link 5 had been repaired first e.g. by applying rule1, and if link 3 was transformed afterwards by applying rule B, then link 5 had to be repaired again, because the applied rule A is no longer applicable and link 5 is inconsistent again. To prevent this, only the transformation of safe inconsistent links is allowed.

The *integrator control* defines the integration process. It starts and controls the different operations of the tool dependent of the working mode the user has chosen (semi-automatic or manual mode, cf. section 2).

The only components of the architecture in Fig. 2 that are not application-independent and have to be rebuilt for every new tool are the rule patterns and actions (painted gray in Fig. 2). Hence, their efficient derivation is important. In the future they will be generated out of a formal rule specification in our prototype.

With this framework integration tools can be built by just coding (at the moment) or specifying (in the future) the transformation rules needed and by instantiating an integrator for the right source and target document types.

4 Integrator-views on documents

Specifying a transformation between two languages in PROGRES faces a problem: Since PROGRES does not have a module concept, it is not possible to hide internals of the documents in encapsulated parts (e.g. modules) within the specification. Thus, the complete internal structure of the documents (i.e. all the details of the database schema) has to be open to the integrator. This is not acceptable since it means ignoring data abstraction completely. It is not necessary as well, since the integrator only needs certain parts of the documents for the transformation.

To deal with this problem, we developed a special abstract interface for the documents called their *integrator-view.* The integrator-views are simplified *graph schemas* (i.e. database schemas for GRAS) that include the subset of the information that is needed for the transformation.

First of all, these graph schemas define an *is-a hierarchy of node classes/types*. Every node in a graph has a unique type and this type belongs to a certain class (hence, a class is a metatype, a type of a type). In our example, the class hierarchies are rather simple. In the source document we differentiate between DFDs and bubbles which represent process declarations and their applications (in other DFDs). In the target document we can similarly distinguish declarations (function modules, i.e. modules that do not encapsulate data) and their applications (imports of these modules in another module). All the nodes of the integrator-views have a string attribute “Id” which contains the identifier of the element. Additionally it is defined which types of edges are allowed between which types of nodes.

Furthermore the *graph structure* is defined. In our example, the documents consist of a set of declarations which contain other declarations or applications of other declarations. A data flow diagram (DFD) e.g. is a process declaration and contains bubbles (process applications) and data flows (applications of type descriptions which are not included in our example, but are of course part of the complete language). This contains-relation is expressed by an edge of type “cont” in the integrator-views. The relation between an application and its declaration is expressed by an edge of type “AtD” (ApplicationToDeclaration).

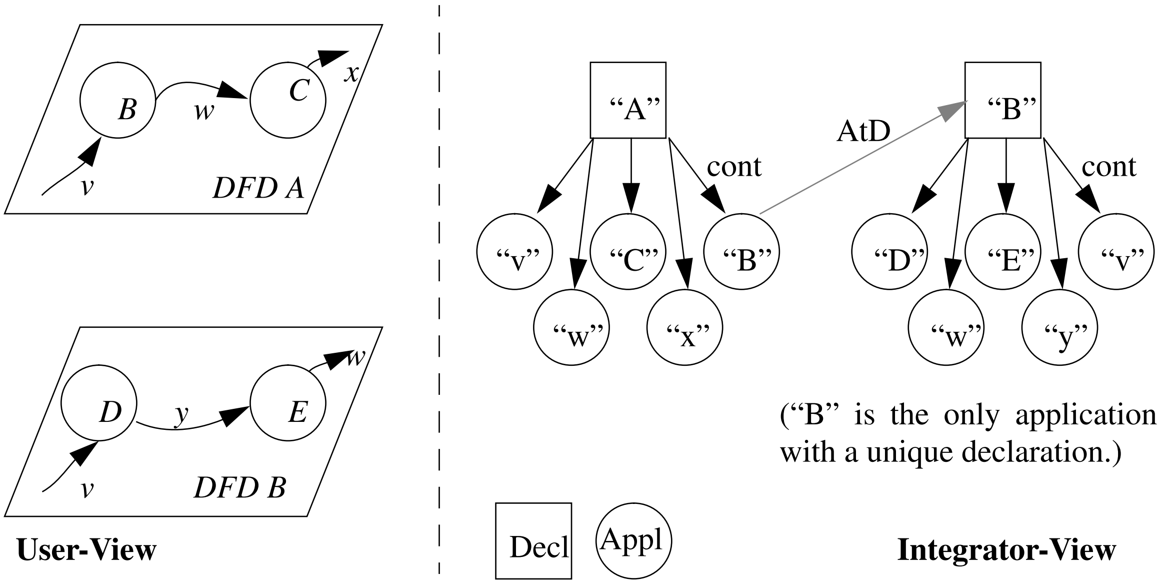


Fig. 4: Requirements specification and its integrator-view

An example of a (very) simple requirements specification consisting of two DFDs and its integrator-view-representation is shown in Fig. 4. This figure simplifies a bit since it does not show the different node types. For instance all the circles are applications, but e.g. “C” and “B” belong to the node type “ProcAppl” (i.e. process application), while e.g. “w” and “x” are instances of the node type “DFAppl” (i.e. data flow application).

Similarly, a corresponding integrator-view of a design document consists of a set of module specifications as declarations. They include procedure declarations (the exported and the local procedures of this module) and applications of other modules (i.e. imports from other modules). The edge types are the same as in the requirements integrator-view. An (again very simple) example of a design document containing two module descriptions together with their integrator-view is shown in Fig. 5. Again, the node types are not shown. E.g. the upper declarations “A” and “B” are of type “Module”, while the contained declarations (i.e. the target nodes of the cont-edges) are of type ExpProcedure. Here, all the applications are of the same node type ImportAppl.

The integrator-views described in this section are only *partial views* on the documents. They contain only the information that is needed for the transformation and not all the information that is needed to build up a document. For instance in the requirements integrator-view it is only described that a data flow is contained in a DFD. The view does not contain the information which bubbles are connected by the data flow, because this information is not needed in the design. (For instance in Fig. 4, the data flow “w” means that the procedures “B” and “C” that are transformed out of processes “B” and “C” are called in the body of procedure “A” with the same actual parameter of type “w”. But the bodies of the procedures are not part of the design.)

In the following, specifying a transformation between requirements and design means specifying a transformation between the according integrator-views. There is an *automatic transfor-*

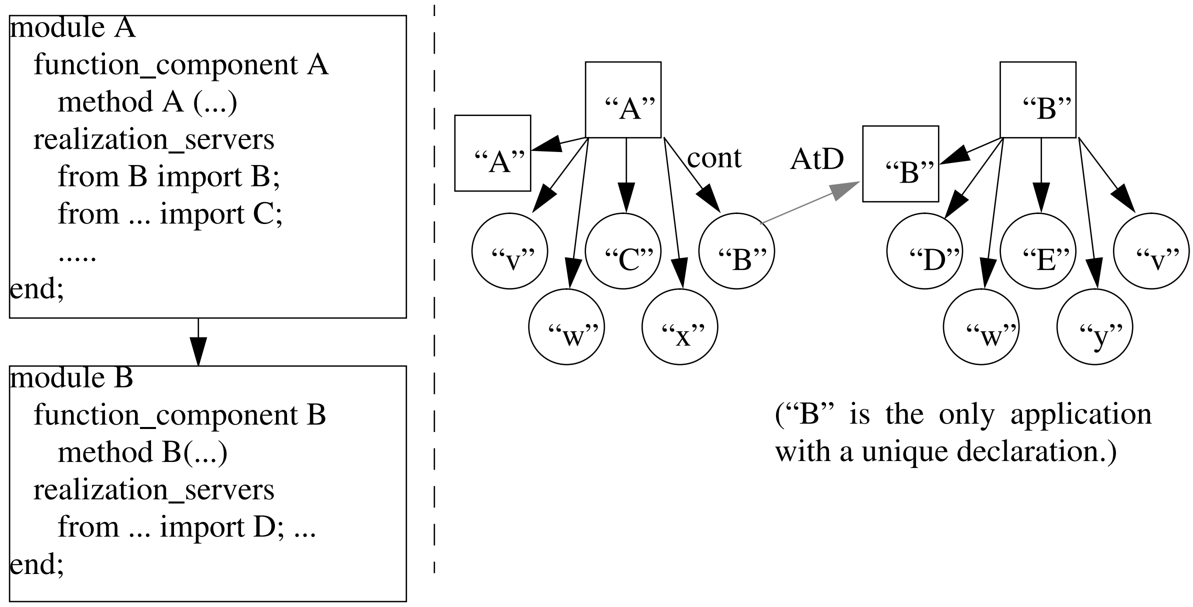


Fig. 5: Design description and its integrator-view

mation that transforms such an integrator-view into the internal document schema or vice versa. But that is not the topic of this paper.

5 Specification of the transformation

As mentioned above, our documents are modelled as graphs. For the specification of operations on those documents we use graph rewriting systems expressed in the language PROGRES. (For our purpose of document transformations we do not need the full power of PROGRES, but only a small subset of the language.) A PROGRES specification mainly consists of a set of productions (i.e. rules) which describe the replacement of a graph pattern (search pattern) by another one.

5.1 The approach

The approach for specifying a document transformation that is introduced in this paper is based on the ideas of *pair grammars* by T.W. Pratt [Pra71]. To describe a transformation between different representations of programs, he used graph grammars for either representation and defined correspondences between the nonterminals in the rules. But Pratts approach is not sufficient for our problem, since it is restricted to context-free graph grammars and one-to-one correspondences between the nodes. Neither is the case here.

But we can use the basic idea of defining graph grammars as PROGRES specifications for the involved documents and *define correspondences* between the rules of the specifications and the nodes in the corresponding rules. Both are many-to-many relations. From the document specifications and the correspondences, different transformation rule sets can be generated automatically. The main advantage of this approach is that only *document-local operations* need to be specified. Then, by defining interdocument correspondences, the relations between the documents are added. The concrete integrator does not need to be specified, it can be generated.

Hence, there is a *separation of local and global parts* of the specification of the transformation and there is a separation of the specification of the transformation (i.e. the relation between the documents) and the integrator (i.e. the operations of the tool).

The specification that has to be written for the documents must allow to create every document state that can occur (i.e. that can be edited with an editor), but it need not consist of real structure-oriented editor operations. Therefore, it is possible to restrict to expanding rules. That means a right side of a production is always a superset of the left side. The rules may only add nodes and edges, they may not delete any of them.

The principle of generating transformation rules from the document specifications is fairly easy: Let

A -> A'

be a production in the specification of the first document (e.g. requirements definition), where A and A' are graph patterns and the rule means "search pattern A in the graph and replace the first match by A' ". If pattern A appears more than once in the graph, one will be chosen non-deterministically. Similarly be

B -> B'

a corresponding production of the other document (here: design document). Then we have four possibilities to generate transformation rules out of them:

- A' c B -> A' c' B' is a transformation rule that can be applied when the first rule has been executed already (i.e. A' exists) and the search pattern of the second can be matched in the design document. Then, the second rule will be applied as well and B is replaced by B'. c and c' represent the interdocument links, which are generated from the correspondences between the nodes in the rules. Hence, c consists of the correspondences between A and B (these still exist, since A' is a superset of A). And c' consists of c and additionally the correspondences of A' and B', i.e. the correspondences of the new inserted nodes. This example is the transformation that is generated for the transformation of requirements into design, where the requirements document exists already and a corresponding design document has to be created during the transformation. This is also the transformation of the integrator between design and Eiffel.
- A c B' -> A' c' B' is the inverse transformation. In our example this might be useful in a re-engineering task, where (a part of) the requirements document has to be restored out of the design document, or similarly a design document has to be restored out of an implementation.
- A' c B' -> A' c' B' is a pure consistency checker. Here, both documents have been created already and the integrator searches for corresponding parts (A' and B') and installs the interdocument links between them.
- A c B -> A' c' B' is a scenario where both documents are build up simultaneously. This is useful only if the rules are explicitly started by a user, or if there is a unique way of building up a document either in A or in B (which is usually not the case).

5.2 Transformation requirements -> design

In this section we want to discuss the transformation of the DFD hierarchy into a corresponding design as part of the integrator between requirements engineering and design. The rules for requirements and design documents will be presented and it is shown how the transformation rules are generated out of the definition of correspondences.

The main idea of the transformation rules is as follows: There are *three possibilities* to transform a process declaration (i.e. DFD). Thus, the rule set is not unique and the integrator is a semi-automatic tool. The user has to select one of the possibilities. The three options for the transformation are:

- a process declaration can become a *functional module* that exports a procedure of the same name,
- a process declaration can become an *exported procedure* of an already existing functional module (that was for instance transformed out of another process declaration),
- a process declaration can become an *local procedure* of an already existing functional module (that was for instance transformed out of another process declaration).

The transformation of a process application depends on the transformation of the surrounding DFD, i.e. the DFD that contains this bubble, and also depends on its own process declaration, i.e. its refining DFD. Therefore, there are two transformations for applications:

- If the depending declarations have been transformed to procedures in different modules, the application becomes an import of its own declaration in the module that contains the procedure created out of its surrounding DFDs.
- If both declarations have become procedures in the same module, the application has no corresponding design increment.

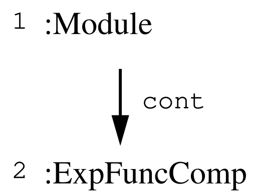
In the following PROGRES examples, the left and right sides of the rules are represented in the same figure. Solid boxes and edges between them form the left side of the rule (i.e. the search pattern) and the dashed boxes and the edges from and to them are the additional elements of the right side, i.e. those nodes and edges that are inserted by the rule.

```
production InsertProcDecl ( NewId : string )
 =

              1 :ProcDecl

   transfer 1.Id := NewId;
 end;
```

Fig. 6: Rule that inserts a process declaration in the requirements

When defining corresponding document specifications for the declarations, we have *one production* that creates a process declaration in the requirements and *three corresponding productions* that insert a module with procedure, an exported procedure, or a local procedure, respectively. The production for the requirements is shown in Fig. 6. There are no solid boxes,

```
production InsertModule ( NewId : string )
 =
```




```
   transfer 1.Id := NewId;
            2.Id := NewId;
 end;
```

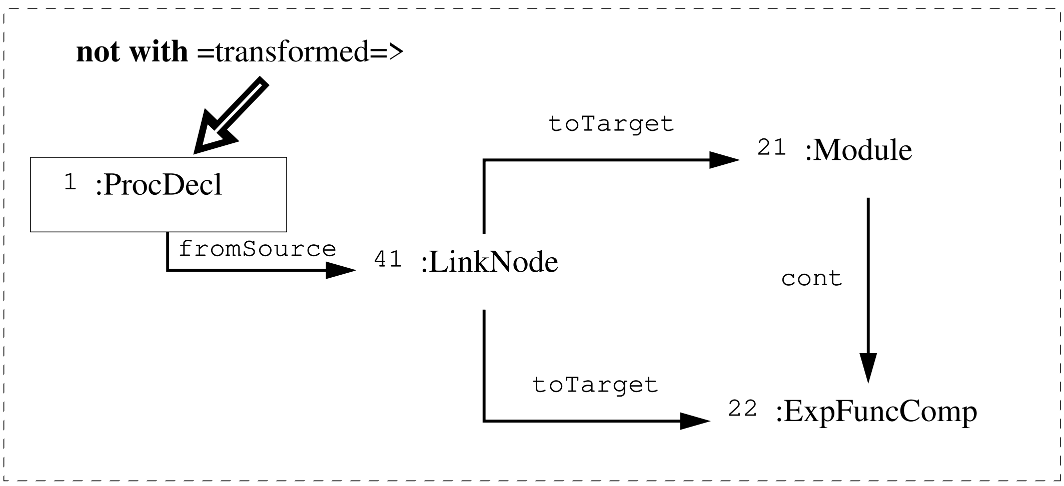
Fig. 7: Rule that inserts module and procedure in design

because the left side of the rule is empty. The node type of the inserted node is ProcDecl. The identifier of the new inserted node is supplied as parameter to the production. The transfer statement is used to assign attribute values. Here, the parameter is assigned to the Id attribute of the new node.

The first corresponding production in the design is the creation of a module with an exported procedure of the same name (called export function component in our design language). This rule is shown in Fig. 7. Again, the left side is empty.

The following correspondences are defined between the nodes of this rules (the syntax is `<doc>.<nn>`, where `<doc>` is RE for requirements engineering or D for design and `<nn>` is the node number in the rule): RE.1 <-> D.1 and RE.1 <-> D.2. From these rules and the correspondences, the transformation rule in Fig. 8 is generated. The double-lined arrow pointing at ProcDecl is a *restriction*. It demands that there is no *path* called “transformed” starting at this

```
production TrafoProcDeclToModule
 =
```



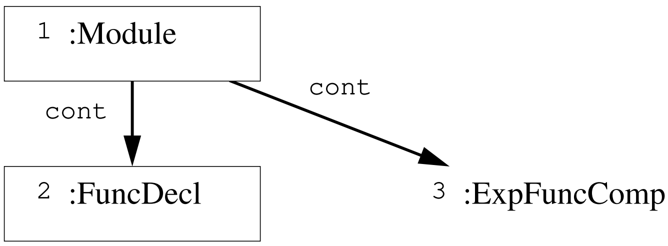

```
 transfer 21.Id := 1.Id;
          22.Id := 1.Id;
 end;
```

Fig. 8: Transformation rule process declaration -> module

node. The path “transformed” is nothing but an abbreviation for a connection of this node to a node in the design document consisting of an edge of type fromSource, a node of type LinkNode and an edge of type toTarget. If the restriction is fulfilled (i.e. the path does not exist, the element is not transformed yet), the rule can be executed, and a module with a procedure and the LinkNode and its edges are inserted. Every rule can insert exactly one LinkNode, and nodes with corresponding nodes in the other document are connected to this LinkNode either by fromSource or by toTarget edges. The LinkNode and its edges represent the interdocument links mentioned above. They are integration information that does not belong to any of the related documents. Therefore, this information is stored in an extra integration document. (The node numbers have been shifted in the figures to associate the nodes more easily to the different documents: nodes no. 1 - 20 are in the requirements document, no. 21 - 40 in the design document, and node no. 41 is always in the integration document.)

Similarly, transformation rules are generated for the other transformations of process declarations. E.g. the second production in the design document that corresponds to the requirements production in Fig. 6 inserts an exported function component in an existing module and is shown in Fig. 9. Here, the node correspondences are RE.1 <-> D.1 and RE.1 <-> D.3, i.e. similar to the rule above, the process declaration corresponds to the module and the procedure that is inserted. (“FuncDecl” is a node class that contains the node types ExpFuncComp and LocFuncComp. In this rule it means, that the module should contain an exported or local function component, i.e. it should be a functional module.)

```
production InsertExpProcedure ( NewId : string )
  =
```



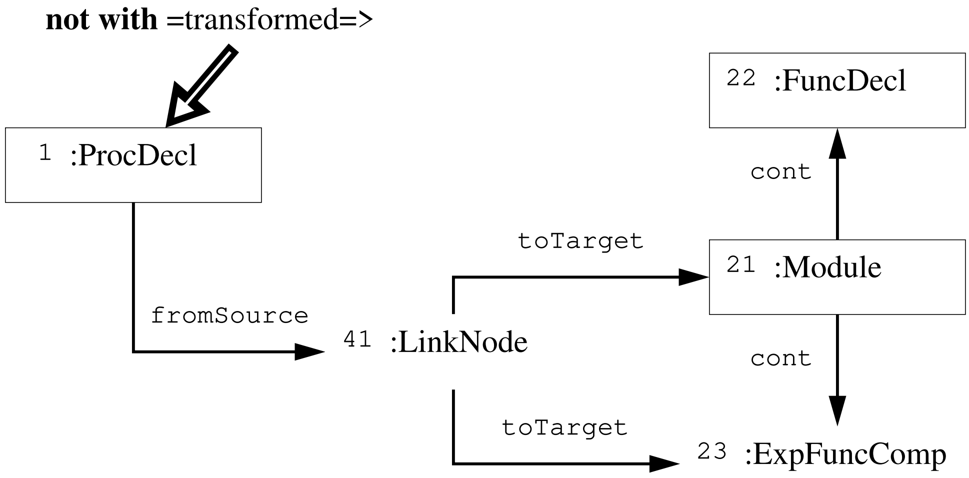

```
    transfer 3.Id := NewId;
  end;
```

Fig. 9: Rule that inserts a new procedure in a module

The transformation rule that is generated from this rule pair with its correspondences is shown in Fig. 10. It searches for a module in the design (that already contains a function declaration), inserts a new exported function component and connects it with an interdocument link. The third possibility of transforming a process declaration is identical to Fig. 10, except that node 23 is of type LocFuncComp. Therefore it is not discussed in greater detail here.

There are two possibilities of transforming a process application depending on the transformation of the declarations as mentioned above. If the declarations have been transformed into procedures of different modules, an import has to be created in the design. The operation in the requirements document is again the same for both possibilities and is shown in Fig. 11.

```
production TrafoProcDeclToExpProcedure
  =
```




```
    transfer 23.Id := 1.Id;
end;
```

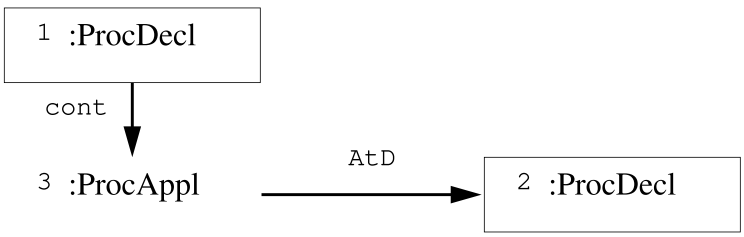
Fig. 10: Transformation rule process declaration -> exported procedure

The corresponding design rule inserts an import of a procedure (ImportAppl) from a certain module (ImpSource) in another module and is shown in Fig. 12.

The correspondences between the nodes of this rule pair are RE.1 <-> D.2, RE.2 <-> D.4, RE.3 <-> D.5 and RE.3 <-> D.6. According to the generation algorithm, the transformation rule in Fig. 13 is created.

If the declarations are transformed into procedures of the same module, there is no equivalent to the process application in design. Nevertheless, a design rule is formulated that does not change the design document, but it defines correspondences to nodes in the rule of Fig. 11. The transformation rule that is generated also does not change the design document, but it inserts some interdocument links between the process application and the procedure declaration.

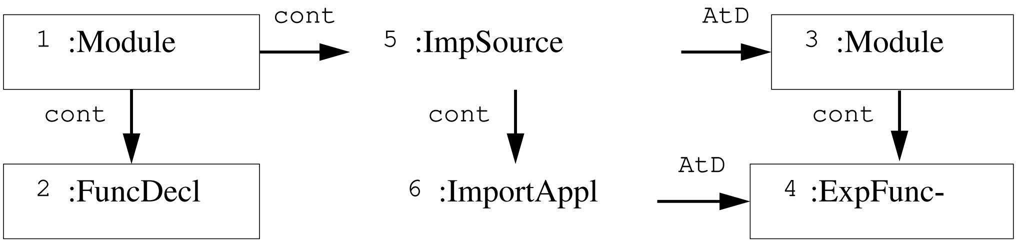
```
production InsertProcAppl( NewApplId : string )
  =
```




```
    condition 2.Id = NewApplId;
    transfer 3.Id := NewApplId;
  end;
```

Fig. 11: Rule that inserts process application in the requirements

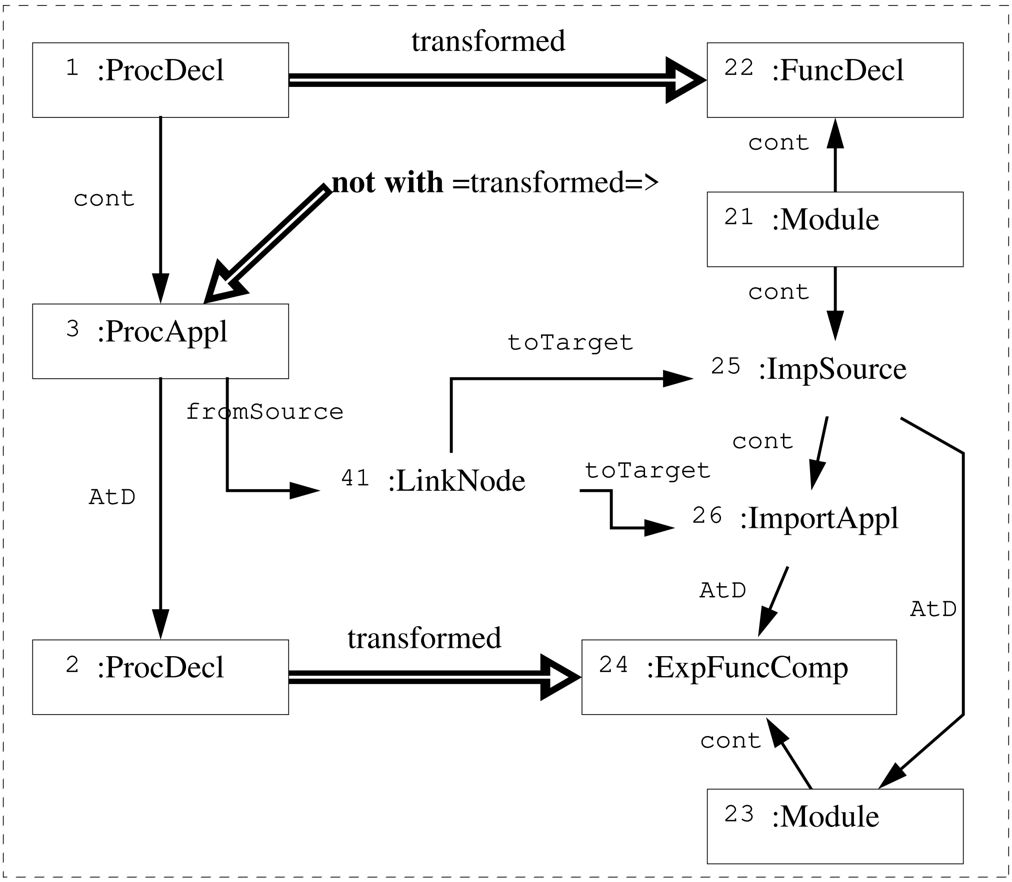
```
production InsertImport( ImportId : string )
  =
```




```
    condition 4.Id = ImportId;
              6.Id = 4.Id;
    transfer 5.Id := 3.Id;
             6.Id := ImportId;
  end;
```

Fig. 12: Insert import in function module

```
production TrafoProcApplToImport
  =
```




```
    condition 24.Id = 3.Id;
    transfer 25.Id := 23.Id;
             26.Id := 3.Id;
  end;
```

Fig. 13: Transformation rule process application -> import statement

These links guarantee that the transformation of the process application is examined again, if the transformation of the declaration is modified (e.g. procedure declaration is deleted).

Hence, we have generated five transformation rules to transform process declarations and applications. The other elements of a requirements definition, i.e. data stores, entity types and relationship types are transformed in a similar way. Doing so, we can derive the complete transformation rule set automatically from the specifications of the related documents.

There are additional operations that are realized in the same way for all applications of integration tools and, hence, need not be specified formally. These are the operations to draw a link manually to an arbitrary target increment, to declare a link as consistent without a target increment, and to delete an arbitrary link.

6 Conclusion

We described an approach to automatically generate transformation rules between different documents in a formal specification language from specifications for the single documents. The so created transformation rules have to be integrated into a generic framework to install an integration tool that executes the transformations and checks the consistency of the so integrated documents.

The advantages of this approach to specify document transformations are the independence of a concrete integration scenario or tool functionality. Different scenarios and transformation directions can be supported in the same way. Furthermore, the specification has a logical structure, since document local and global parts are separated.

We already implemented the integrator between requirements engineering and design, and the integrators between design and implementation (Eiffel) and the integrator for horizontal integration within requirements engineering will be finished in the near future. The framework is implemented using C++ and Modula-2 on Sun SPARC-Stations.

The next step of this work will be to formalize our approach using a cathegory-based approach from graph grammar theory, to widen the supported scenarios, to validate our integrators, and possibly build further prototypes in different applications.

References

[Asl91] M. Aslett: “Constructing software engineering environments using the Software Backplane”, in F. Long: *Software Engineering Environments - Volume 3*, Ellis Horwood (1991), pp. 401-414.

[BEM92] A.W. Brown, A.N. Earl, J.A. McDermid: *Software Engineering Environments*, McGraw-Hill (1992).

[BFW92] A.W. Brown, P.H. Feiler, K.C. Wallnau: “Past and Future Models of CASE Integration”, Proc. of the CASE ‘92, Montreal (CDN), July 1992, pp. 36-45, IEEE CS Press (1992).

[BJ92] J. Börstler and Th. Janning: "Traceability Between Requirements Engineering and Design: A Transformational Approach", *Proc. of the COMPSAC '92*, Chicago, Sept. 1992, pp. 362-368.

[Bör93] J. Börstler: *Programming in the Large and Reusability* (in German), Ph.D. Thesis, Aachen University of Technology.

[Bou88] G. Boudier et. al.: "An Overview of PCTE and PCTE+", *Proc. of the 3rd Symp. on Practical Software Development Environments*, ACM SIGSOFT Software Engineering Notes, Vol. 13(5), Boston, November 1988, pp. 248-258.

[GI90] D. Garlan and E. Ilias: "Low-cost, Adaptable Tool Integration Policies for Integrated Environments", *Proc. of the 4th Symp. on Practical Software Development Environments*, ACM SIGSOFT Softw. Eng. Notes, Vol. 15(6), Dec. 1990.

[GKS87] D. Garlan, C. Krueger, B. Staudt: "A Structural Approach to the Maintenance of Structure-Oriented Environments", *Proc. of the 2nd Symp. on Practical Software Development Environments*, ACM SIGPLAN Notices, Vol. 22(1), January 1987, pp. 160-170.

[Ger89] C. Gerety: *A new generation of software development tools*, Tech. Report, SESD-89-25, Hewlett-Packard Software Engineering System Division, November 1989.

[IBM89] IBM: *AD/Cycle concepts*, Doc. No. GC26-4531-0, 1989.

[Jan92] Th. Janning: *Integration of Languages and Tools to Requirements Engineering and Programming in the Large* (in German), Ph.D. Thesis, Aachen University of Technology, DUV-Verlag (1992).

[KSW93] N. Kiesel, A. Schürr, B. Westfechtel: "GRAS, A Graph-Oriented Database System for (Software) Engineering Applications", *Proc. of the CASE '93*, pp. 272-286, Singapore, July 1993, IEEE CS Press (1993).

[Lef93] M. Lefering: "Tools to Support Life Cycle Integration", *Proc. of the 6th Software Engineering Environments Conference 1993 (SEE '93)*, Reading (UK), July 1993, pp. 2-16, IEEE CS Press (1993).

[Nag94] M. Nagl (Ed.): *Building Tightly Integrated Software Development Environments - The IPSEN Project*, to appear in LNCS, Springer (1994).

[Nor87] K. Normark: *Transformations and Abstract Presentations in a Language Development Environment*, Ph.D. Thesis, University of Aarhus (DK), February 1987.

[Pra71] T.W. Pratt: "Pair Grammars, Graph Languages, and String-to-Graph-Translations", *Journal of Computer and System Sciences*, vol. 5 (1971), pp. 560-595.

[RE92] B. Ramesh, M. Edwards: "Issues in the Development of a Requirements Traceability Model", *Proc. of the IEEE Int. Symposium on Requirements Engineering (RE '93)*, San Diego (CA), Jan. 1993, pp. 256-259, IEEE CS Press (1992).

[Rei90] St. Reiss: "Interacting with the FIELD Environment", in *Software Practice and Experience*, Vol. 20(S1), June 1990, pp. 89-115.

[Sch91] A. Schürr: PROGRES: A VHL-Language Based on Graph Grammars, in Ehrig/Kreowski/Rozenberg (Eds.): *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, pp. 641-659, Springer (1991).

[Wel91] T. Welsh: "Digital's COHESION environment", in F. Long: *Software Engineering Environments - Volume 3*, Ellis Horwood (1991), pp.363-378.

[WJ93] L. Wakeman, J. Jowett: *PCTE - The standard for open repositories*, Prentice Hall (1993).