A Uniform Declarative Framework for Automated Software Merging

Tom Mens and Kim Mens*

{ tommens | kimmens }@vub.ac.be

Programming Technology Lab

Vrije Universiteit Brussel

Pleinlaan 2, B-1050 Brussel, Belgium

Abstract. *We report on a prototype tool that automates the time-consuming and error-prone process of software merging. Our tool is significantly more flexible than existing merge techniques, as it can detect syntactic, structural as well as semantic conflicts. It is implemented as a general framework for software evolution that can be customised to many different domains. Because of this, it can be used to support evolution of any kind of software artifact, independent of the target language or the considered phase in the software life cycle.*

1 Introduction

In order for software engineering (SE) tools to be as widely applicable as possible, they should be built in a general way, independent of the chosen programming language, the available technology, or the considered phase in the software life cycle. The latter is especially important, as more and more emphasis is being put on the higher life-cycle phases (analysis, design and architecture). Hence, SE tools should be able to treat software artifacts in all phases transparently. Unfortunately, most current-day SE tools and environments lack such a uniform approach.

Another essential feature of SE tools should be their ability to deal with *software evolution*, since evolution is essential and inevitable in any SE activity. Existing tools address this need only in part, e.g., by resorting to a version control system [Conradi&Westfechtel1998]. However, besides versioning there are many other evolution issues that should be taken into account. In particular, the issue of *software merging*, needed when software artifacts are being modified in parallel by different software engineers, is not properly dealt with by existing approaches [Mens2000b]. Most current-day merge tools adopt the technique of *text-based merging*. This can be regarded as a domain-independent approach, since any software artifact is treated as a flat text file. However, this approach turns out to be inadequate for finding sophisticated inconsistencies -such as syntactic or semantic merge conflicts- during merging.

In research literature we encountered only a few more powerful merge approaches that have been proposed with the specific aim of being domain independent:

- Westfechtel proposes a 3-way merge technique that detects lexical conflicts when merging parse trees [Westfechtel1991]. He also deals with more complex conflicts that are due to changes in the bindings of identifiers to their declarations.
- Berzins takes an alternative approach by relying on a language-independent definition of semantic merging [Berzins1994]. To this extent, a generalisation of traditional denotational semantics is used to provide the additional structure necessary to formally define semantic merge conflicts. Because Berzins' approach works on the semantics of a model directly, it cannot be used to diagnose and locate conflicts between changes in the concrete syntactic representation of a program. This prohibits the approach to pinpoint the actual source of a semantic conflict in the software.
- Another domain-independent approach is proposed in [Mens1999]. It is more general than the work of Westfechtel and Berzins, in the sense that it can detect syntactic as well as semantic conflicts.[1] The technique is based on the mathematical formalism of *graph rewriting* [Mens2000a], and builds further on the *reuse contracts* approach for managing reuse and evolution in a

* Research funded by the Brussels' Capital Region and Getronics Belgium.

[1] However, the semantic conflicts that can be detected by Berzins are more sophisticated.

disciplined way [Steyaert&al.1996]. Because reuse contracts have been customised to many different domains, including class collaborations [Lucas1997a], UML models [Mens&al.1999, Mens&D'Hondt2000] and software architectures [Romero1999], they are a suitable candidate for a uniform merge approach. This paper reports on a general framework for software evolution we developed based on these ideas.

2 Dealing with Software Merging

This section discusses in detail how we intend to deal with the problem of software merging. Software merging typically occurs during large-scale collaborative software development, where separate lines of development are carried out in parallel by different software engineers, and have to be merged at regular intervals. Merging is a time-consuming and complicated process, because many interconnected elements are involved, and because merging depends on the structure and semantics of those elements. Therefore, sophisticated SE tools that provide automated support are essential. Unfortunately, most existing approaches to software merging either lack flexibility or expressive power, as indicated in [Mens2000b].

One of the more promising approaches is the use of *operation-based merging* [Feather1989, Lippe&vanOosterom1992], which is a specific flavour of *change-based merging*. With operation-based merging, not only the information in the original software artifact and its evolved versions is used (as is the case with *state-based merging*), but the evolution operations that were applied to obtain the evolved versions are taken into account as well. These evolution operations can be arbitrarily complex and capture the changes that are made in the SE environment. An operation-based merge approach facilitates conflict detection, and allows for better support when solving these conflicts.

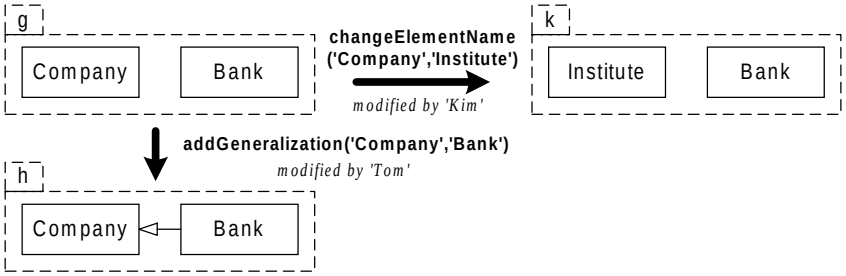


Figure 1: Operation-based merging

To illustrate the idea of operation-based merging, consider Figure 1. We start with two unrelated classes `Company` and `Bank`. Developer Tom decides to introduce a generalisation relationship from `Company` to `Bank` using the operation `addGeneralisation('Company','Bank')`. Independently, developer Kim decides to rename `Company` into `Institute` using the operation `changeElementName('Company','Institute')`. With state-based merging, the renaming cannot be detected, since there is no way to distinguish it from the introduction of a new class `Institute`. With operation-based merging, we can take the renaming operation into account to detect and resolve this problem.

Syntactic merge conflicts arise when the result of the merge is no longer well formed. For example, this would be the case when the horizontal operation in Figure 1 would be `classRemoval('Company')`. If we try to merge this modification with the introduction of a generalisation edge to `Company`, we obtain an *undefined source conflict*. (One cannot remove a class that is being depended on.) Operation-based merging allows us to detect syntactic conflicts in a straightforward and efficient way. Instead of comparing two parallel revisions, we simply compare the operations that have been applied to obtain these revisions, and look up the associated conflict (if there is one) in a *conflict table*. Obviously, the used operations and reported conflicts may vary for each domain.

Structural conflicts arise when one of the parallel operations is a *refactoring* or *restructuring* transformation[2] [Opdyke1992, Roberts&al.1997, Fowler1999], and the merge algorithm cannot decide in which way the merged result should be structured. As an example of such a conflict, consider the situation of Figure 2. Horizontally, the object-oriented class diagram `bankApplication.0` is restructured, by splitting up class `Bank` into two parts `Bank` and `Agency`. During this process, all adjacent

[2] These operations have the special property that they are behaviour-preserving, i.e., they do not change the semantics of the software, although its structure can change significantly.

edges to the old version of `Bank` are redirected to either the new `Bank` (as is the case for the generalisation relationship) or `Agency` (as is the case for the association relationships). In parallel, the vertical evolution modifies the diagram by associating a new class `Safe` with `Bank`. When merging both parallel changes, the question arises whether `Safe` should remain associated with `Bank`, or whether it should be associated with `Agency` instead. This cannot be decided autonomously by the merge algorithm, because it depends on the meaning attached to `Safe`. Hence, input from the user is necessary in order to resolve the conflict. In this case, we will decide to associate `Safe` with `Agency` during the merge, but it is easy to find examples where the opposite decision should be taken. Nevertheless, a conflict table approach can still be used for detecting structural conflicts.

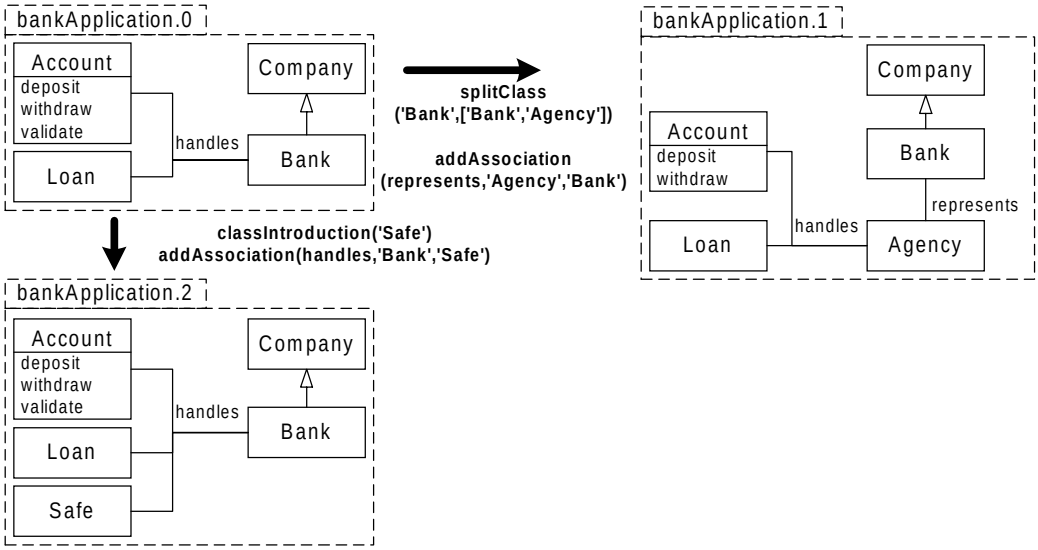


Figure 2: Structural conflict

Semantic conflicts occur when parallel evolutions give rise to unexpected interactions, in the sense that the merged result does not behave as expected. To illustrative such a conflict, consider the `Account` class, which contains a number of methods `deposit`, `withdraw` and `validate`. Initially, `withdraw` invokes `validate` to verify whether it is safe to withdraw money from the account (i.e., the credit limit has not been reached, and the amount that is withdrawn is not too high). In the horizontal evolution, the behaviour implemented in `validate` is 'inlined' in `withdraw` for efficiency reasons. In the vertical evolution, `Account` is being protected with a password to prevent unauthorised users from withdrawing money. To achieve this, the implementation of `validate` is refined to incorporate password checking. Unfortunately, when we merge both parallel evolutions, it is still possible to withdraw money without a password, because the horizontal evolution step removed the invocation from `withdraw` to `validate`.

3 A Uniform Framework for Software Merging

We propose a uniform framework for software evolution and merging that is implemented in the logic programming language Prolog. Prolog enables fast prototyping, supports multi-way querying facilities, and its powerful unification and backtracking mechanism allows us to express most conflict detection and resolution rules straightforwardly. Moreover, the uniform syntax of facts and rules makes it possible to use them in a transparent way.

The framework can be divided in two main parts: a part that manages evolution and merging in a *domain-independent* way, and a *domain-specific* customisation that specifies the translation to a particular domain (e.g., implementation code, design models, software architectures). To reason about domain-specific software artifacts, the Prolog tool is linked to a software repository or database in which all these artifacts are stored. By establishing a direct coupling between records in the database and Prolog facts, we can reason about the stored software artifacts in a uniform and transparent way.

In this section, we will explain the domain-independent framework, while section 4 will elaborate on how to customise the framework to specific domains.

3.1 Managing Evolution

3.1.1 Attributed graphs

As an underlying model for representing software artifacts in a uniform way we use *attributed graphs*. This means that the only entities that are used are nodes and directed edges. Nodes and edges have a label, while edges have an additional source and target node. Nodes and edges also have a type, that will be used to impose domain-specific constraints on them. All nodes and edges specify the name of the graph to which they belong, to be able to deal with different versions of a graph. Finally, nodes and edges may be qualified with attributes that can store any kind of additional information, such as the name of a modifier: `modifier('Tom')`; a version number: `version('1.0')`; the status of the node or edge: `status(deleted)`; or the date and time of a modification: `timestamp('2000/04/01 – 16:39:25')`. Below we see part of the internal representation of the graph `h` of Figure 1 that contains two nodes `Bank` and `Company` of type `class` and an edge `hasChild` of type `generalisation` between these nodes.

```
graph(h).
node(h,'Bank',class,[version('1.0')]).
node(h,'Company',class,[version('1.0')]).
edge(h,hasChild,'Company','Bank',generalisation,[version('1.1'),modifier('Tom')]).
```

3.1.2 Primitive evolution productions

Because we want to provide uniform support for software evolution, we need to be able to make arbitrary changes to a graph. Therefore, a predefined set of primitive and non-overlapping *evolution productions* is provided. Each production has its own specific meaning and notation, as shown in Table 1. From a formal point of view, these primitive productions are sufficient to specify any kind of change to a given graph.

In Table 1 and in the rest of this paper, we use the following conventions. Like in Prolog, words starting with uppercase letters represent variables. Because Prolog is untyped, we use naming conventions to clarify the meaning of the various arguments of a predicate. Variables representing *labels* start with `L` (e.g., `Ln`, `Lm`, …), variables representing *types* start with `T`, and variables representing *attribute lists* start with `A`.

`extension(Ln,Tn)`	add a node with label `Ln` and type `Tn`
`cancellation(Ln,Tn)`	remove a node with label `Ln` and type `Tn`
`relabelNode(Ln,Lm)`	change the label of a node from `Ln` to `Lm`
`retypeNode(Ln,Tn,Tm)`	change the type of a node with label `Ln` from `Tn` to `Tm`
`changeNodeAttributes(Ln,A1,A2)`	change the list of qualified attributes of a node with label `Ln` from `A1` to `A2`
`refinement(Le,Ls,Lt,Te)`	add an edge with label `Le` and type `Te` between a source node with label `Ls` and a target node with label `Lt`
`coarsening(Le,Ls,Lt,Te)`	remove an edge with label `Le` and type `Te` between a source node `Ls` and a target node `Lt`
`relabelEdge(Le,Ls,Lt,Lf)`	change the label of an edge with source node `Ls` and target node `Lt` from `Le` to `Lf`
`retypeEdge(Le,Ls,Lt,Te,Tf)`	given an edge with label `Le`, source node `Ls` and target node `Lt`, change its type from `Te` to `Tf`
`changeEdgeAttributes (Le,Ls,Lt,A1,A2)`	change the list of qualified attributes of an edge with label `Le`, source `Ls` and target `Lt` from `A1` to `A2`

Table 1. Primitive evolution productions

3.1.3 The evolve predicate

The predicate **`evolve(Gi,Gr,P,M,Conflicts)`** is used to transform an initial graph `Gi` into a result graph `Gr` by applying one of these primitive productions `P`. To specify the name of the person performing the evolution, a fourth argument `M` is used. This is especially useful in the case of

collaborative software development, where different modifiers can make changes to the same software artifact. Finally, the last argument `Conflicts` returns a list of conflicts in those cases where the evolution fails. This is for example the case when certain preconditions assumed by the production `P` are not satisfied by the initial graph `Gi`. A simple example of an evolution step, corresponding to the vertical arrow of Figure 1 is:

```
evolve(g,h,refinement(hasChild,'Company','Bank',generalisation),'Tom',Conflicts)
```

It asks to apply a refinement to the initial graph `g` in order to obtain a new graph `h`. Depending on what the initial graph `g` looks like, this evolution either succeeds or fails. In case of success, `Conflicts` is bound to the empty list `[]`. In case of failure, `Conflicts` is bound to `[applicability(refinement(hasChild,'Company','Bank',generalisation),'Tom')]` to indicate that the refinement proposed by Tom is not applicable to the initial graph. This would for example be the case when one of the nodes `Company` or `Bank` did not yet exist, or when there would already exist a `generalisation` relationship between `Company` and `Bank`.

In order to perform the actual evolution, `evolve` proceeds in four steps:

1. `checkProduction`: check the well-formedness of the production `P`;
2. `preconditions`: validate the required preconditions of the production `P` in the initial graph `Gi`.
3. `apply`: apply the production `P` to the initial graph `Gi`.
4. `postconditions`: validate whether the postconditions of `P` are satisfied in the result graph `Gr`.

This approach implies that we need to specify well-formedness checks, pre- and postconditions, as well as application rules for each of the primitive productions of Table 1. For example, the rules for the production `refinement` look as follows:

```
checkProduction(refinement(Le,Ls,Lt,Te)) :-
  isEdgelabel(Le), isEdgetype(Te), isNodelabel(Ls), isNodelabel(Lt).

preconditions(refinement(Le,Ls,Lt,Te),Gi) :-
  presentNode(Gi,Ls), presentNode(Gi,Lt), absentEdge(Gi,Le,Ls,Lt).

apply(production(refinement(Le,Ls,Lt,Te),Gi,M) :-
  assertEdge(Gi,Le,Ls,Lt,Te,M).

postconditions(refinement(Le,Ls,Lt,Te),Gr) :-
  presentNode(Gr,Ls), presentNode(Gr,Lt), presentEdge(Gr,Le,Ls,Lt,Te).
```

`checkProduction` basically determines whether all arguments of the `refinement` production have the correct type. The predicates `preconditions` and `postconditions` test whether certain nodes or edges are present (positive conditions) or absent (negative conditions) before and after application of the production. Finally, whenever we `apply` the `refinement`, we add a new `edge` fact to the fact base by calling the `assertEdge` predicate.

3.1.4 Composite evolution productions

Because using only primitive productions does not scale up in more complex situations, we also allow to apply an entire *sequence* of productions at once. For example, we can redirect the source node of a given `association` edge from `Bank` to `Agency` by applying the following **production sequence**:

```
[ coarsening(handles,'Bank',  'Account',association),
  refinement(handles,'Agency','Account',association)  ]
```

The `evolve` predicate performs this production sequence by sequentially applying each of its constituent productions.

To deal with frequently occurring production sequences we can also create *predefined* sequences, which will be referred to as **composite productions**. For example, the production sequence mentioned above can be abbreviated to `redirectSource(handles,'Bank','Account','Agency',association)`. It is an instance of a composite production that is defined as follows:

```
composite( redirectSource(Le,Ls,Lt,Lnew,Te),
           [ coarsening(Le,Ls,Lt,Te), refinement(Le,Lnew,Lt,Te) ] ).
```

From the outside, primitive and composite productions cannot be distinguished since they behave in exactly the same way. For example, we can also attach preconditions and postconditions to composite productions. A direct consequence of this transparent treatment of primitive and composite productions is that composite productions can be used in a similar way as database transactions: the production is either applied as a whole, or not applied at all if one of the constituents of the composite production is not applicable for some reason.

3.2 Managing parallel evolution

3.2.1 The merge predicate

The `evolve` predicate alone is insufficient if we need to combine parallel evolution steps that have been made by different software developers to the same initial graph. In that case, both parallel evolutions need to be *merged*. This can give rise to inconsistencies or unexpected interactions because both evolutions may not be compatible with each other. For this reason, we implemented a predicate **`merge(Gi,Gr,P1,P2,M1,M2,Conflicts)`**. Like `evolve`, it requires an initial graph name `Gi` and a result graph name `Gr`. It also requires two graph productions `P1` and `P2`, provided by two different developers `M1` and `M2`, respectively. In the example of Figure 1, the following merge command should be invoked:

```
merge(g, merged_g, addGeneralisation('Company','Bank'),
      changeElementName('Company','Institute'), 'Tom', 'Kim', Conflicts )
```

Note that the evolution operations `addGeneralisation` and `changeElementName` used in this example are nothing more than domain-specific variants of the primitive graph productions `refinement(hasChild,'Company','Bank',generalisation)` and `relabelNode('Company','Institute')`, respectively. How exactly domain-specific evolution operations are translated into graph productions will be explained in Section 4.3.

When the merge fails, a list of `Conflicts` is returned. Three kinds of merge conflicts can be distinguished. *Syntactic merge conflicts* arise when `P1` cannot be applied after `P2` or vice versa, because this would give rise to an ill-formed result graph `Gr`. *Structural conflicts* appear when two parallel evolutions make incompatible changes to the structure of a program. *Semantic conflicts* arise when the result of the merge does not behave as expected, although the result graph `Gr` is syntactically correct. The next subsections elaborate on each of these kinds of conflicts.

3.2.2 Detecting syntactic conflicts

As already mentioned in section 2, the paradigm of operation-based merging allows us to detect *syntactic conflicts* in a straightforward and efficient way. It suffices to compare those pairs of primitive contract types `P1` and `P2` that are involved in the merge by looking up the associated conflict in a conflict table. For example, suppose that we apply the following merge:

```
merge(g,merged_g,refinement(hasChild,'Company','Bank',generalisation),
      cancellation('Company',class),'Tom','Kim',Conflicts )
```

This particular merge gives rise to a syntactic conflict. The detection of the conflict proceeds as follows. First, the merge algorithm tries to apply the `refinement` to `g` which gives rise to some intermediate result graph. Next, the algorithm tries to apply the `cancellation` to this intermediate graph. This fails because the precondition that `Company` should have no adjacent edges does not hold. Hence, a general `applicability(cancellation('Company'),'Kim')` conflict is generated, that is transformed into a more specific `SyntacticConflict` by invoking the predicate `lookupConflicts`:

```
lookupConflicts( applicability(cancellation('Company'),'Kim'),
                 refinement(hasChild,'Company','Bank',generalisation),
                 cancellation('Company',class),
                 SyntacticConflict).
```

Internally, `lookupConflicts` invokes the `conflictTable` predicate to compare the two primitive productions that are involved. In this particular case, the following clause of `conflictTable` will be triggered, giving rise to an `undefinedSource` conflict.

```
conflictTable(ApplConflict,cancellation(Ls,Ts),refinement(Le,Ls,Lt,Te),
              undefinedSource(Ls,Ts,Te)).
```

This more specific conflict, together with the modification and modifier that caused it, is then bound to the variable `SyntacticConflict`, and presented to the user:

```
syntacticConflict(undefinedSource('Company',class,generalisation),
                  modification(cancellation('Company',class)),modifier('Tom'))
```

In some situations we can even go further than merely detecting a conflict. For example, when we perform the merge of subsection 3.2.1, we get a syntactic conflict `relabeledSource('Company', 'Institute')`, but in this case the conflict can be resolved automatically by imposing a certain order on the operations: if we first apply `addGeneralisation`, and next `changeElementName`, we don't get a syntactic conflict anymore. In other words, for certain combinations of operations, the merge algorithm may be refined to include *default conflict resolution strategies*.

Finally, the use of *composite productions* also has an impact on how syntactic conflicts can be detected. Due to space constraints, however, we cannot discuss this issue here.

3.2.3 Detecting structural conflicts

Figure 2 of section 2 illustrated a *structural inconsistency*. It arises when one evolver performs a restructuring operation, while another evolver makes a change that interacts with this restructuring. Unlike with syntactic conflicts, the merge does not necessarily gives rise to an ill-formed result graph. Nevertheless, we would like to detect these situations as they may give rise to problems in future evolution steps. In order to detect structural inconsistencies, we can use the same technique as for syntactic conflicts: we can compare restructurings with other operations to see whether they give rise to unexpected interactions, and look up the associated structural conflict in a conflict table. Hence, the only thing that needs to be done is to extend the conflict table with restructuring operations and their associated conflicts.

3.2.4 Detecting semantic conflicts

The fact that two parallel evolutions can be merged into a well-formed result graph `Gr` is no guarantee that everything will behave correctly. The parallel evolution steps may still interact in undesired ways. If this happens, we say that a *semantic conflict* has occurred. A typical example of a semantic conflict is the unexpected introduction of cycles (possibly leading to infinite recursion) because two parallel developers independently introduce an edge in the opposite direction. Such a conflict can be detected by looking in the result graph for particular occurrences of graph patterns that were not yet present in the initial graph. For example, the code below specifies how a cycle introduction should be detected:

```
conflictPattern(Gr, cycleIntroduction(Gr,Ls,Lt,T)) :-
  edge(Gr,Le,Ls,Lt,T,C), edge(Gr,Lf,Lt,Ls,T,C2),
  differentModifier(C,C2).
```

Basically, what happens is that in the graph `Gr` the occurrence is checked of two arbitrary edges with the same type `T` but in the opposite direction, and such that both edges were introduced by different modifiers. As can be seen in the code above, detecting occurrences of conflict patterns is extremely simple in Prolog, thanks to its built-in unification and backtracking mechanism.

3.3 Other evolution predicates

Besides the `evolve` and `merge` predicates that deal with software evolution and merging, respectively, many more predicates are needed for dealing with other aspects of software evolution. The following useful predicates have also been implemented, or are currently being implemented.

`undo(Gr,Gi,P,M)` can be regarded as the inverse of `evolve`. Suppose we have used `evolve` to calculate a result graph `Gr` by applying a production `P` (primitive, composite or sequence) to an initial graph `Gi`, and that we have deleted the initial graph `Gi`. Then we can still reconstruct `Gi` by starting from the result graph `Gr` and applying the production `P` in the opposite direction. This is possible because each of the primitive productions in Table 1 has an inverse. E.g., the inverse of `extension(Ln,Tn)` is `cancellation(Ln,Tn)` and the inverse of `relabelEdge(Le,Ls,Lt,Lf)` is `relabelEdge(Lf,Ls,Lt,Le)`.

`compact(ProdList,CompressedList)` compacts a primitive production sequence `ProdList` by removing all redundant productions. For example, if an edge between two nodes is first added and removed later, the two corresponding productions `refinement` and `coarsening` are redundant and can be removed. Compaction is useful to reduce storage space, to increase the efficiency of evolution and merging, and to remove intermediate syntactic conflicts. Additionally, `compact` rearranges all remaining primitive productions in the sequence to a canonical form, by putting all productions of the same kind together. For more details, see [Mens1999].

`diff(Gi,Gr,ProdList)` calculates the difference between two graphs as a sequence of primitive productions.

4 Domain-Specific Extensions

While our uniform framework implements the basic mechanisms for dealing with evolution and merging of attributed graphs, additional rules are needed for the tool to work in a domain-specific context. Irrespective of the particular domain to which the framework is customised, one should always follow the four steps below:

1. Map the software artifacts that reside in an external repository to Prolog facts. More precisely, translate all entities and relationships in terms of nodes and edges.
2. Specify the different types of nodes and edges that can be distinguished, and declare well-formedness constraints that hold between the different types.
3. Specify which evolution operations (primitive as well as composite) are useful in the considered domain, specify domain-specific pre- and postconditions, and translate the operations into domain-independent graph productions.
4. Translate the syntactic and semantic conflicts that are generated by the domain-independent conflict detection algorithm into conflicts that are more meaningful for the particular domain.

The following subsections discuss these steps in more detail. In the context of a research project with our industrial partner Getronics, we customised the framework to the particular domain of UML class diagrams. This allowed us to experiment with evolution and merging on an existing design repository containing about 600 classes. In parallel to this experiment, customisations to other domains, such as software architectures, have been explored as well [Mens1999, Romero1999].

4.1 Mapping of repository data

Because the considered case was too large to fit in Prolog memory, we were forced to store the class diagrams in an external repository. We linked our tool to this repository by relying on the Prodata™ interface. This interface provides a tight coupling between Prolog and any ODBC-compliant database management system [Lucas1997b]. As such, we retained all advantages of using standard Prolog, while still being able to reason about externally stored software artifacts. However, because accessing the data of an external repository through ODBC was less efficient than when storing the data directly in memory, we needed to improve the efficiency by defining highly optimised SQL queries (at the cost of loosing generality, reusability or portability).

To translate the domain-specific software artifacts into domain independent graph nodes and edges, some translation predicates were needed. Both a predicate `node_db`, which dynamically retrieves an artifact from the repository, and a predicate `edge_db`, which dynamically looks up some artifact relationship in the repository, were defined in terms of a predefined ODBC predicate `db_sql_select` which executes an SQL statement over the database. The implementation of these SQL statements strongly relies on the internal database representation and the considered software artifacts.

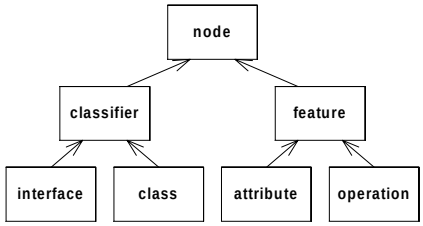
Following the above approach, all artifacts and relationships in the repository are represented as *virtual nodes and edges*, and our framework does not need to distinguish between nodes and edges that are retrieved transparently from the database (`node_db` and `edge_db`), and nodes and edges that are stored directly as facts (`node_fact` and `edge_fact`) in Prolog. Everything is treated as an ordinary `node` or `edge`.

When customising the framework to another domain, we simply need to provide another external repository containing the domain-specific software artifacts, and redefine the predicates `node_db` and `edge_db` for doing the appropriate translations.

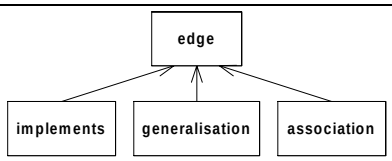
4.2 Domain-specific well-formedness constraints

Next to translating the domain-specific software artifacts into a graph representation, we need to specify domain-specific constraints that have to hold in these graphs. In order to express these well-formedness constraints, we first specify the different types of software artifacts using a *type hierarchy*. In the domain of class diagrams we distinguish the following types of nodes: `class`, `interface`, `operation`, `attribute`, … Some of these types (such as `class` and `interface`) have commonalities, so they have a common supertype (in this case, `classifier`). All type information, as well as the subtyping relationship between all node types and edge types, is specified using facts of the form `subtype(Subtype,Supertype)`. We hereby assume that `node` and `edge` are the root of the two type hierarchies.

```
subtype(classifier,node).
subtype(feature,node).
subtype(interface,classifier).
subtype(class,classifier).
subtype(attribute,feature).
subtype(operation,feature).
```




```
subtype(implements,edge).
subtype(generalisation,edge).
subtype(association,edge).
```



Types are used to impose domain-specific constraints on nodes and edges. For example, an edge of type `generalisation` is only allowed between two nodes of the same type, which must be a subtype of `classifier`. Such a constraint can be checked using the predicate **`checkTypeConstraint`**`(G,TypeConflicts)` that makes use of the following occurrence of the `edgeTypeConstraint` predicate:

```
edgeTypeConstraint(edge(G,Le,Ls,Lt,generalisation,_),TypeError) :-
  not (node(G,Ls,T,_),node(G,Lt,T,_),isSubtype(classifier,T)),
  TypeError = generalisationConstraint([Le,Ls,Lt]).
```

From a formal point of view, the approach taken above corresponds to specifying a *type graph*, and requiring that each graph satisfies the constraints imposed by this type graph [Mens2000a].

4.3 Domain-specific evolution operations

In order to reason about evolution in a particular domain, we use domain-specific evolution operations instead of the general graph productions introduced in section 3.1.2. This requires a translation of the domain-specific operations into graph productions using the predicate `translateOperation(Operation,Production)`. We already mentioned in subsection 3.2.1 how the evolution operations `changeElementName` and `addGeneralisation` of Figure 1 are translated into graph productions `relabelNode` and `refinement`, respectively. Similarly, in Figure 2, **`addAssociation`** is translated into `refinement`, while **`classIntroduction`** becomes `extension`. The code for all these translations is presented below:

```
translateOperation(changeElementName(Old,New),relabelNode(Old,New)).
translateOperation(addGeneralisation(Parent,Child),
    refinement(hasChild,Parent,Child,generalisation)).
translateOperation(addAssociation(Name,From,To),
    refinement(Name,From,To,association)).
translateOperation(classIntroduction(Name),extension(Name,class)).
```

Some complex domain-specific operations cannot be translated directly into a primitive graph production but need to be specified in terms of a composite production. This is for example the case with the operation **`redirectAssociationSource`**`(handles,'Bank','Account','Agency')` which is translated into the composite production `redirectSource(handles,'Bank','Account','Agency',association)` that was introduced in section 3.1.4.

An even more complex situation occurs with **`splitClass`**`('Bank',['Bank','Agency'])` in Figure 2, which is itself defined as a sequence of *domain-specific* operations `[createClass('Agency'), redirectAssociations('Bank','Agency')]`, each of which must subsequently be translated recursively until everything is expressed in terms of primitive graph productions.

In the same way as we have attached pre- and postconditions to graph productions in section 3.1, we can attach *domain-specific* pre- and postconditions to the evolution operations. To this extent we use the predicates `domainPreconditions(Operation,G)` and `domainPostconditions(Operation,G)`.

4.4 Domain-specific conflict detection

As a final step, we need to compute all domain-specific conflicts. This is done in two phases. First, we apply the domain-independent conflict detection algorithm, and use the `translateConflict(A,B)` predicate to translate all conflicts that are generated during evolution or merging into meaningful

domain-specific error messages. As an example, reconsider the syntactic conflict explained in subsection 3.2.2. Instead of merely reporting an `undefinedSource` conflict, we use the edge type and node type to translate it in a more meaningful `unexistingSuperclass` conflict:

```
translateConflict(      unexistingSuperclass(Ln),
                        undefinedSource(Ln,class,generalisation)).
```

During the translation process, we also filter all generated domain-independent conflicts, and report only those conflicts that are relevant for the particular domain.

In a second phase, we investigate the result of the evolution or merge, and check whether it complies to the imposed domain-specific well-formedness constraints using the predicate `checkTypeConstraint`. Each breach of a type constraint gives rise to a domain-specific syntactic conflict. A typical example is the accidental introduction of multiple inheritance in a language where this is not allowed.

Finally, it is also possible to specify extra domain-specific semantic conflicts, by defining additional conflict detection rules that only hold for the particular domain.

5 Tool Issues

In order for our merge framework to be adopted in practice, it needs to be integrated in an industrial SE environment. To achieve this, many different avenues can be followed.

- One possibility is to integrate our framework with a CASE tool like Select Enterprise™ so that we can reason about evolution of UML models. In our industrial case study this was achieved by exporting class diagrams to an external database using a script, and directly accessing the database from Prolog using an ODBC interface. Other alternatives would be to access the CASE tool repository directly, or to use the standard XML format to interchange UML models between the CASE tool and the Prolog framework.
- Another avenue is to embed our approach in an integrated SE environment like Smalltalk. We are planning to integrate our tool in SOUL, which provides a tight symbiosis between a logic language and the Smalltalk development environment [Wuyts1998]. This will enable our framework to reason about evolution of Smalltalk programs directly.
- A final avenue is to integrate our framework in existing configuration management tools. While these tools provide extensive support for versioning, our approach contributes by providing a more powerful way to do software merging. An obvious prerequisite is the ability of the configuration management system to deal with change-based versioning, of which operation-based versioning is a particular flavour.

Further work is necessary in order to deal with other evolution issues as well. In the context of an industrial research project, we are currently extending the framework to provide support for *co-evolution* [D'hondt&al.2000], where parallel changes are made to the same software artefact at different levels of abstraction (i.e., analysis, design and implementation).

Our framework should also be able to provide support for *impact analysis*, *change propagation*, the *ripple effect*, *version proliferation*, etc… Because most of these problems can be dealt with by using some variant of graphs, our framework can probably be extended in a straightforward way to deal with these issues. For example, we can use program dependence graphs to deal with impact analysis [Bohner&Arnold1996], and graph rewriting to address the propagation of changes and the ripple effect [Rajlich1997].

6 Conclusion

This paper presented a uniform framework for software merging that can easily be customised to different domains, and that allows us to deal with software evolution in a scalable way. We illustrated that a logic language provides an ideal medium for implementing such a framework. By resorting to the underlying model of graphs, and by representing evolution in terms of primitive and composite graph productions, evolution problems could be dealt with independent of any particular domain. A characteristic feature of our approach is the powerful support for software merging, which exceeds that of most currently existing merge tools. By resorting to an operation-based approach, potential syntactic, structural as well as semantic conflicts can be detected automatically in a straightforward way.

The implemented research prototype has been validated for the domain of class diagrams in the context of an industrial case study, but further experiments are needed to validate it in other domains as well.

Also, we still need to address some efficiency issues, and incorporate other evolution issues such as support for impact analysis and change propagation. Better integration with existing SE tools and environments is also required.

7 References

[Berzins1994] V. Berzins, "Software Merge: Semantics of Combining Changes to Programs," *ACM Trans. Programming Languages and Systems*, Vol. 16, No.6, 1994, pp. 1875-1903.

[Bohner&Arnold1996] S. Bohner, R. Arnold, *Software Change Impact Analysis*, IEEE Press, 1996.

[Conradi&Westfechtel1998] R. Conradi, B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys*, Vol. 30, No. 2 (June), 1998.

[D'Hondt&al.2000] T. D'Hondt, K. De Volder, K. Mens, R. Wuyts, "Co-Evolution of Object-Oriented Design and Implementation," *Int. Symp. Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2000.

[Feather1989] M. Feather, "Detecting Interference when Merging Specification Evolutions," *Proc. 5th Int'l Workshop Softw. Specification and Design*, ACM Press, 1989, pp. 169-176.

[Fowler1999] M. Fowler, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.

[Horwitz&Reps1992] S. Horwitz, T. Reps, "The Use of Program Dependence Graphs in Software Engineering," *Proc. 14th Int. Conf. Softw. Eng.*, pp. 392-411, ACM Press, 1992.

[Lippe&vanOosterom1992] E. Lippe, N. van Oosterom, "Operation-Based Merging," *Proc. 5th ACM SIGSOFT Symp. Softw. Development Environments, ACM SIGSOFT Softw. Eng. Notes*, Vol. 17, No. 5, 1992, pp. 78-87.

[Lucas1997a] C. Lucas, *Documenting Reuse and Evolution with Reuse Contracts*, PhD Dissertation, Vrije Universiteit Brussel, September 1997.

[Lucas1997b] R. Lucas, *LPA WIN-PROLOG 4.0 Prodata Interface*, Manual, Keylink Computers Ltd., 1997.

[Mens1999] T. Mens, *A Formal Foundation for Object-Oriented Software Evolution*, PhD Dissertation, Vrije Universiteit Brussel, September 1999.

[Mens&al.1999] T. Mens, C. Lucas, P. Steyaert, "Supporting Disciplined Reuse and Evolution of UML Models," *«UML»'98 - Beyond The Notation: Selected Papers of «UML»'98 International Workshop*, LNCS 1618, pp. 378-392, Springer-Verlag, 1999.

[Mens2000a] T. Mens, "Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution," *Proc. Int'l Agtive '99 Conf.*, LNCS, Springer-Verlag, 2000.

[Mens2000b] T. Mens, "A State-of-the-Art Survey on Software Merging," Submitted to *IEEE Trans. Softw. Eng.*, March 2000.

[Mens&D'Hondt2000] T. Mens, T. D'Hondt, "Automating Support for Software Evolution in UML," *Automated Software Engineering Journal* 7:1, Kluwer Academic Publishers, February 2000.

[Opdyke1992] W. Opdyke, *Refactoring Object-Oriented Frameworks*, doctoral dissertation, Univ. of Illinois at Urbana-Champaign, 1992. Tech. Report UIUC-DCS-R-92-1759.

[Rajlich1997] V. Rajlich, "A Methodology for Software Evolution," *Journal of Software Maintenance*, Vol. 9, 1997, pp. 103-125.

[Roberts&al.1997] D. Roberts, J. Brant, R. Johnson, "A Refactoring Tool for Smalltalk," *J. Theory and Practice of Object Systems*, Vol. 3, No. 4, 1997, pp. 253-263.

[Romero1999] N. Romero, *Managing Evolution of Software Architectures with Reuse Contracts*, EMOOSE Dissertation, Vrije Universiteit Brussel, 1999.

[Steyaert&al.1996] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt, "Reuse Contracts: Managing the Evolution of Reusable Assets," *Proc. OOPSLA '96, ACM SIGPLAN Notices*, Vol. 31, No. 10, 1996, pp. 268-286.

[Westfechtel1991] B. Westfechtel, "Structure-Oriented Merging of Revisions of Software Documents," *Proc. 3rd Int'l Workshop on Softw. Configuration Management*, ACM Press, 1991, pp. 68-79.

[Wuyts1998] R. Wuyts, "Declarative Reasoning About the Structure of Object-Oriented Systems," *Proc. TOOLS USA '98*, pp. 112-124, IEEE Computer Society Press, 1998.