Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution

Tom Mens

Programming Technology Lab, Department of Computer Science,
Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium
tommens@vub.ac.be

Abstract. This paper presents a formal approach for managing unanticipated software evolution. Labelled typed nested graphs are used to represent arbitrarily complex software artifacts, and conditional graph rewriting is used for managing evolution of these artifacts. More specifically, we detect structural and behavioural inconsistencies when merging parallel evolutions of the same software artifact. The approach is domain-independent, in the sense that it can be customised to many different domains, such as software architectures, UML analysis and design models, and software code.

1 Introduction

When looking at current-day CASE tools and software development environments, we observe that most of them provide poor support for evolution or no support at all. If evolution support is provided, it is usually ad hoc and restricted to a single phase in the software life-cycle.

Even version control tools [4] do not adequately deal with evolution. When merging parallel evolutions of the same software artifact, the best they can do is detect structural inconsistencies in the result of the merge [27]. A number of research prototypes exist that can also detect behavioural inconsistencies [2, 3], but these approaches restrict themselves to a specific language.

We believe that it is essential to have a tool that allows us to detect behavioural merge conflicts in a uniform way. It should be customisable to software artifacts in different phases and different domains without needing to modify the underlying formalism or algorithms. In this paper we present such an approach, based on the technique of reuse contracts [18, 24]. Because this technique for dealing with unanticipated evolution has already been customised to a number of different domains, including class collaborations, UML class diagrams and software architectures, it seems to be a suitable candidate to express our ideas.

By representing arbitrary software artifacts by means of *graphs*, and evolution of these software artifacts by means of *conditional graph rewriting*, it becomes possible

to express the ideas behind reuse contracts directly using the formal properties of the graph rewriting formalism.

2 Documenting Evolution

To be able to manage unanticipated evolution of software artifacts, the evolution step needs to be documented explicitly in a disciplined (i.e., formal) way. To express the specific way in which a software artifact is modified, different types of modification can be identified by specifying a *modification type* that imposes extra restrictions on the evolution step. Modification types are fundamental to disciplined evolution, as they give us relevant information for identifying merge conflicts when merging parallel evolutions of the same software artifact.

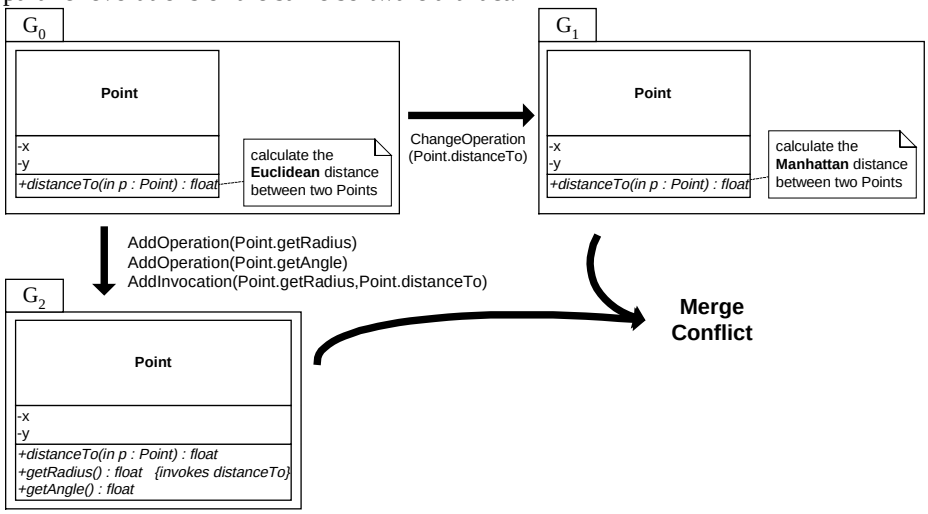

Fig. 1. Documenting parallel evolutions of the same software artifact allows us to detect behavioural incompatibilities.

To explain more clearly how merge conflicts can be detected, consider the example of Fig. 1, where a very simple UML class diagram is being modified by different persons during collaborative software development. Two parallel modifications are made to a *Point* class containing two attributes *x* and *y*, and one operation *distanceTo* which calculates the Euclidean distance between two points (the receiver and the argument). One software developer modifies this basic behaviour by reimplementing *distanceTo* so that it calculates the Manhattan distance instead. Independently, and completely unaware of these changes, a second software developer extends the functionality of the *Point* class by introducing two new operations *getRadius* and *getAngle* for working in polar coordinates. Since *getRadius* corresponds to the Euclidean distance to the origin, it can be calculated by performing a self send to *distanceTo*.

When merging these parallel modifications, a behavioural conflict arises, because the *getRadius* operation does not behave as it should anymore. Indeed, because *getRadius* invokes *distanceTo*, which now calculates the Manhattan distance, a different result is obtained than before (although the coordinates of the point have not changed). Such an unanticipated behavioural incompatibility is called a *merge conflict* (more specifically, an *inconsistent target* conflict). It cannot be detected by straightforward merging approaches, because they do not take behavioural information into account.

While the example above is very simple, in practice the evolving software artifacts will be more complex, and many different changes will be made simultaneously. Additionally, the merge conflict mentioned above is only one of the many different kinds that can arise. This makes it unfeasible to detect merge conflicts manually. Therefore, semi-automated tool support for detecting such behavioural incompatibilities is essential.

One should note that the detection of behavioural merge conflicts in general is an undecidable problem. Therefore, the best we can do is take a conservative approach that gives us a safe approximation of all potential behavioural conflicts. In other words, we can only generate conflict warnings rather than actual conflicts.

The innovative idea of reuse contracts is that potential behavioural conflicts *can* be detected in a very straightforward way, by explicitly documenting each modification. For example, the horizontal evolution step in Fig. 1 is expressed by a graph derivation $G_0 \Rightarrow G_1$ which does not only specify the original version and the evolved version, but also formally documents the way in which G_1 is obtained from G_0. This documentation is given by a graph production *ChangeOperation(Point.distanceTo)*. It specifies that the *distanceTo* operation is modified in some way. The vertical evolution step, described by a derivation sequence $G_0 \Rightarrow G_2$ which is composed of three graph productions. Two *AddOperations* specify the addition of *getRadius* and *getAngle*, respectively. *AddInvocation(Point.getRadius,Point.distanceTo)* specifies that the implementation of *getRadius* performs a self send to *distanceTo*. It is the combination of the vertical *AddInvocation* and the horizontal *ChangeOperation* that gives rise to the merge conflict. *AddInvocation* shows that an extra call to *distanceTo* is added, while independently the behaviour of *distanceTo* is modified in an incompatible way.

In the remainder of this paper we will illustrate that graph rewriting techniques provide a very suitable domain-independent mechanism for expressing evolution and dealing with merge conflicts.

3 Formal Foundation

3.1 Graphs

Because we want the reuse contract technique to be applicable to many different domains, we need to choose a formalism that is general enough, yet still intuitive to work with.

We have chosen to represent software artifacts by *labelled nested typed graphs* for various reasons. *Graphs* are an intuitive, visually attractive, general and mathematically well-understood formalism. The edges in a graph are used to represent all kinds of software dependencies, such as data-flow and control-flow dependencies. *Types* are introduced as a classification mechanism to distinguish different types of nodes and edges with similar characteristics. *Nesting* is used as a means to reduce the complexity of a graph, by allowing nodes to contain graphs themselves (cf. [9], [21]). The labelled graphs we use are similar to those in [7], except that our graph labels also contain a set of constraints. Moreover, we require some extra injectivity conditions on the node and edge labels. See [19] for more detailed information.

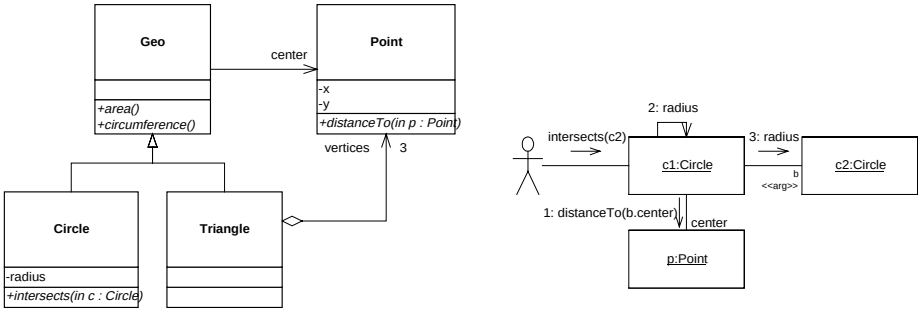

Fig. 2. Example of a UML class diagram and related collaboration diagram.

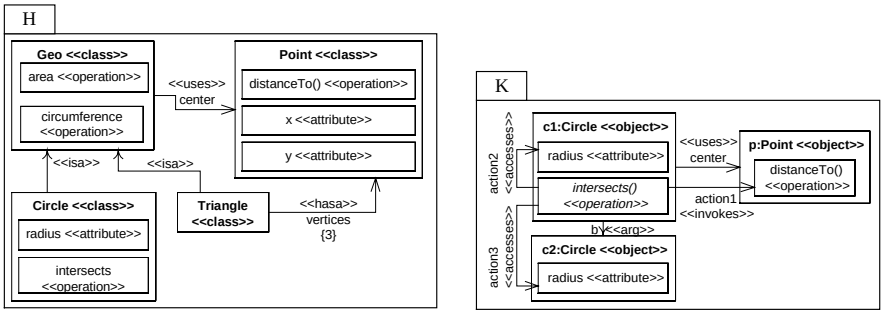

Fig. 3. Graph representations corresponding to the UML diagrams.

As an illustration of the graphs we use, consider the UML class diagram and collaboration diagram of Fig. 2, which represent part of some graphical object-oriented framework. Their underlying graph representations are depicted in Fig. 3. Names of classes, attributes, operations and objects in the UML diagrams correspond to node labels in the graph. Each of these nodes has a corresponding type, which is either *«class»*, *«operation»*, *«attribute»* or *«object»*. Associations, aggregations and specialisations between classes correspond to edges with types *«uses»*, *«hasa»* and *«isa»*, respectively. *«isa»*-edges have no labels. Message sends in the collaboration diagram correspond to *«invokes»*-edges or *«accesses»*-edges in the underlying graph representation, depending on whether they refer to an *«attribute»*-node or

«operation»-node. *«operation»*-nodes and *«attribute»*-nodes are always nested inside a *«class»*-node or *«object»*-node. Finally, the *«hasa»*-edge labelled *vertices* contains an extra constraint, denoted between curly braces {}, to express a multiplicity requirement in the corresponding class diagram (namely that each triangle contains 3 points as vertices).

The node types and edge types used in Fig. 3 also have to satisfy some constraints. We already mentioned the constraint that an *«operation»*-node or *«attribute»*-node must always be nested in a *«class»*-node or *«object»*-node. Other obvious constraints are that *«hasa»*-edges can only be placed between *«class»*-nodes, and similarly for *«isa»*-edges and *«uses»*-edges. All these constraints can be expressed formally in a so-called *type graph* [5, 6, 9]. From an intuitive point of view, the type graph is a metagraph which puts extra restrictions on the kind of graphs that are allowed. Type graphs are very important to customise our formalism to different domains. For each specific domain, a type graph must be defined that expresses the well-formedness rules that must hold for all the domain-specific software artifacts.

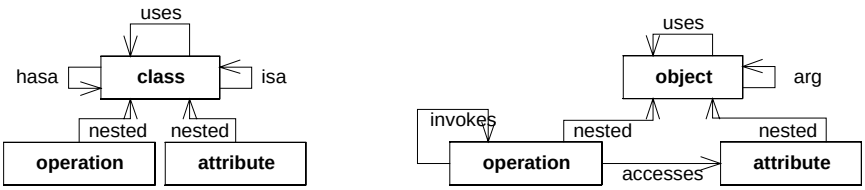

Fig. 4. Type graphs *ClassTypes* and *ObjectTypes* expressing well-formedness constraints for UML class diagrams and collaboration diagrams.

In Fig. 4, the type graphs *ClassTypes* and *ObjectTypes* for the graphs *H* and *K* of Fig. 3 are shown. Labels in the type graph correspond to types in the graph. A notable exception is *nested*, which is an edge label in the type graphs, although there is no explicit *«nested»*-type. As opposed to UML class diagrams, in collaboration diagrams we allow *«accesses»*-edges to be put from an *«operation»*-node to an *«attribute»*-node, and *«invokes»*-edges between two *«operation»*-nodes (to specify operation invocations, self sends and super sends). In this way, the structural information expressed in a class diagram becomes supplemented with additional behavioural information.

It should be noted that some constraints cannot be expressed easily using a type graph. For example, the restriction that an inheritance (isa) hierarchy should by acyclic is difficult to express. As a pragmatic solution, this restriction can be attached as an extra well-formedness constraint to the isa-edge in the type graph.

For a complete formal definition of the graphs and type graphs that we use, we refer to [19]. All definitions can be given in a category-theoretical way. For example, ***Graph*** defines a category with unlabelled graphs as objects and node-preserving and edge-preserving morphisms, ***LGraph*** defines a category with labelled graphs as objects, and *partial* graph morphisms as morphisms. If, additionally, the morphisms are label-preserving, we get a subcategory ***LGraphL***. Typed graphs also form a category ***LTGraph(T)*** with partial morphisms, which is parameterised with the type

graph ***T***. For each specific domain, another type graph ***T*** is used, as shown in Fig. 4. Each ***T***-typed graph *G* has a corresponding ***LGraph***-morphism *type: G→T* assigning a type to each node and edge of *G*. ***LTGraph(T)*** also contains subcategories for label-preserving morphisms, type-preserving morphisms, and both.

3.2 Graph Rewriting

Since graphs are used to represent software artifacts, graph rewriting is a natural choice to represent *evolution* of these artifacts. The research area of graph rewriting has a large mathematical backing [8, 10, 11, 12]), while it remains fairly intuitive in use. We use the *algebraic single-pushout* approach towards *conditional* graph rewriting [13, 14, 15], where application conditions are used to determine when a certain production is applicable to a given graph. This is essential to detect merge conflicts between incompatible evolutions of the same artifact.

From now on we will use the word *graph* instead of *labelled typed nested graph*, because all the definitions presented in [17] are proven in the general category of *graph structures*, which encompasses ordinary graphs, hypergraphs, typed graphs, etc.

Basically, a graph rewriting is defined in terms of a **graph production** *p: L→R* which transforms a left-hand side *L* into a right-hand side *R* by means of some transformation *p*. The actual graph rewriting is obtained by applying the transformation *p* in the context of a larger graph *G*. Therefore, a **match** *m: L→G* is needed to specify how the left-hand side *L* is embedded in *G*. Given *p* and *m*, we can define a **graph derivation** $G \Rightarrow_{p,m} H$ by applying *p* in the context of *G*. By sequentially applying a number of productions, we obtain a **derivation sequence** $G \Rightarrow^{+} H$.

Mathematically, the result graph *H* of a derivation $G \Rightarrow_{p,m} H$ is obtained by calculating the *pushout* of *p: L→R* and *m: L→G*. This definition corresponds to the single-pushout approach to graph transformations [17]. The pushout of *p* and *m* gives rise to two new morphisms *p*: G→H* and *m*: R→H*.

Fig. 5 illustrates this approach by means of an example. We start from a graph *G* that satisfies the type graph *ClassTypes* of Fig. 4. *G* contains *«class»*-nodes *Circle* and *Triangle*, which both have *«operation»*-subnodes *circumference* and *area*. Additionally, both *«class»*-nodes are the source of a *«uses»*-edge *center* with as target a *«class»*-node *Point*. *Point* contains subnodes *distanceTo*, *x* and *y*. Finally, *Circle* has an extra *«attribute»*-subnode *radius*, while *Triangle* is the source of an additional *«hasa»*-edge with label *vertices*. The production *p: L→R* factorises the common behaviour of *Circle* and *Triangle*. Instead of letting *Circle* and *Triangle* directly access the *Point* node, a common parent *Geo* is introduced through which all communication takes place. *Geo* captures the commonalities of *Circle* and *Triangle* by defining the *circumference* and *area* nodes. In this way, redundancy is removed, and the design is made more reusable. Note that *L* does not need to specify the *radius* subnode of *Circle*, the subnodes of *Point* or the *vertices* edge from *Triangle* to *Point*, since these are not required for performing the transformation. The match *m: L→G* is a total label-preserving and type-preserving graph morphism.

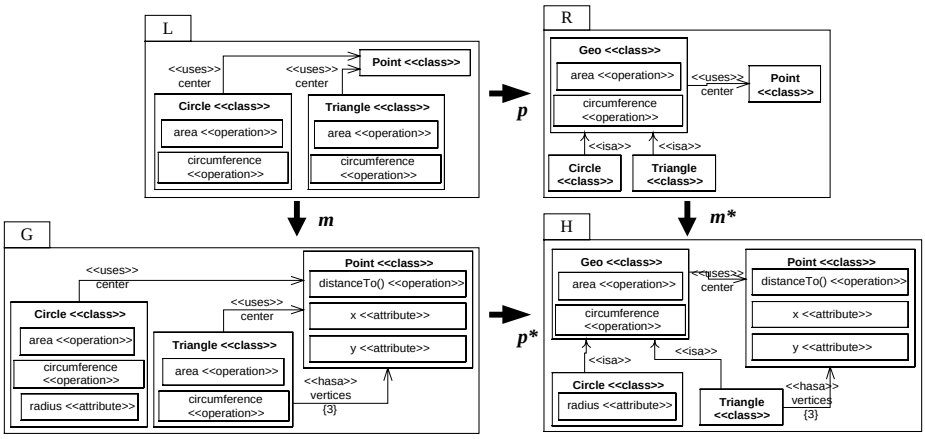

Fig. 5. Example of a graph derivation

Although it is not apparent from the above example, graph productions are also allowed to change the type of nodes and edges, as long as these retypings preserve the constraints imposed by the type graph. For example, if we use the type graph *ClassTypes* we can change *«attribute»*-nodes to *«operation»*-nodes, but we are not allowed to change a *«class»*-node to an *«attribute»*-node or *«operation»*-node since this would breach some of the edge type constraints.

In order to formally define merge conflicts, we need the notion of parallel and sequential independence. Intuitively, two (parallel) graph derivations $G \Rightarrow_{p1, m1} G_1$ and $G \Rightarrow_{p2, m2} G_2$ starting from the same graph G are **parallel independent** if they can be applied one after the other. A similar notion of **sequential independence** says that the order in which two productions p_1 and p_2 are applied in a derivation sequence $G \Rightarrow_{p1,m1} G_1 \Rightarrow_{p2,m2} G_2$ is irrelevant. Obviously, there is a close relation between parallel and sequential independence. Two parallel independent derivations can always be sequentialised, and lead to a unique result graph (under certain injectivity constraints) which is independent of the order in which the productions are applied. This property is usually referred to as the **local confluence property**, and it is essential when merging parallel evolutions of the same software artifact.

Because the above formalism of graph rewriting is not expressive enough for our purposes, we also need to attach **application conditions** to productions [13, 14, 15]. The above properties and definitions that hold for ordinary graph rewriting are directly generalisable to conditional graph rewriting. Intuitively, application conditions impose additional restrictions on a graph derivation $G \Rightarrow_{p,m} H$. In the case of application **preconditions**, the production $p: L \rightarrow R$ can only be applied in the context of G if additional constraints, specified by a morphism $c: L \rightarrow L'$ are satisfied. These constraints can be **positive**, which means that the match $m: L \rightarrow G$ must satisfy the conditions imposed by c. If the constraints are **negative**, the match m should never satisfy the conditions imposed by c. This can be stated formally by demanding that there is no morphism $s: L' \rightarrow G$ that makes the diagram on the left of Fig. 6 commute.

As a concrete example, *L'* on the right of Fig. 6 presents two preconditions that could be attached to the production *p: L→R* of Fig. 5. A negative precondition states that there should be no node with label *Geo* present in *G*. It is depicted by a dashed striked-through ellipse surrounding the prohibited node. A positive precondition states that there should be at least one *«hasa»*-edge from *Triangle* to *Point*. Since both preconditions are indeed satisfied in Fig. 5, the conditional production can be applied.

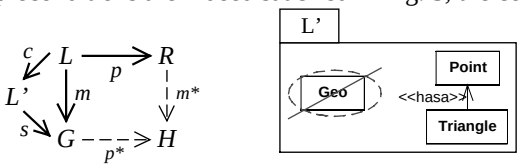

Fig. 6. Application preconditions.

To stress the fact that we work with application conditions, we will talk about **conditional productions** and **conditional derivations**. A conditional production is defined as a couple *(p: L→R, ApplCond(p))* where *ApplCond(p)* specifies the set of application conditions attached to a production *p: L→R*. For the sake of the presentation, we restrict ourselves to application *pre*conditions in this paper.

4 Domain-independent Formalism for Evolution

4.1 Modification Types

Using the formalism of conditional graph rewriting, the **modification type**, which specifies the kind of modification that takes place, is defined as a parameterised conditional production. An example has already been shown in Fig. 5. Although in this particular example, the production *p: L→R* preserves labels and types, this is not required in general. Moreover, *p: L→R* is a *partial* morphism, since it is not defined for all nodes and edges of *L*. Some nodes and edges on the left-hand side (the ones that are deleted) do not have a counterpart on the right-hand side. More specifically, the subnodes *circumference* and *area* of *Circle* and *Triangle* do not have a counterpart in *R*, and similarly for the edges *(center,Circle,Point)* and *(center,Triangle,Point)*.

In order to detect merge conflicts more easily, we restrict ourselves to a limited set of possible modifications that can be made to a graph. The following **primitive modification types** are provided: adding a node or edge to a graph (*AddNode* and *AddEdge*), removing a node or edge from a graph (*DropNode* and *DropEdge*), and changing the type of a node or edge (*RetypeNode* and *RetypeEdge*). The exact definition of the primitive modification types is given in Fig. 7. Each modification type is parameterised with a number of node and edge labels and types. For type parameters, greek letters (ω, υ, τ and ϕ) are used. Only negative application preconditions are needed. Because of space considerations and to increase the readability, these application conditions are not shown in a separate graph *L'*. Instead,

they are mentioned in the left hand-side *L* inside dashed striked-through ellipses. For more details, we refer to [19].

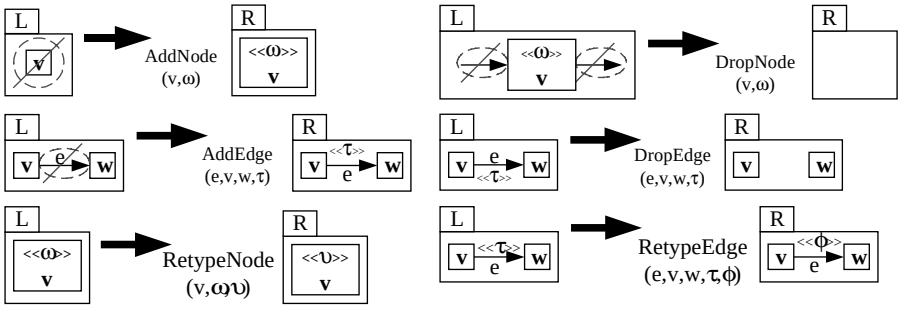

Fig. 7. Primitive Modification Types

Together, the six primitive modification types can be used to express any possible kind of graph modification that does not involve nesting. In Fig. 8, an example is given of a primitive modification type p_1 = *AddEdge(ε,area,radius,«accesses»)*, where ε denotes the empty edge label. It is applied in the context of an *«object»*-node *Circle*, using the type graph *ObjectTypes* of Fig. 4.

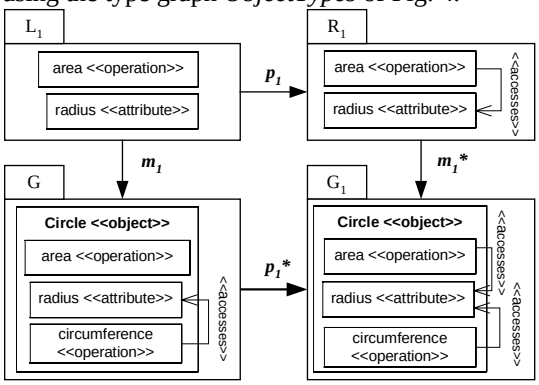

Fig. 8. Example of a primitive modification type *AddEdge(ε,area,radius,«accesses»)*

4.2 Structural Conflicts

The above characterisation of primitive modification types helps us to detect **merge conflicts** when merging parallel evolutions $G \Rightarrow_{p1,m1} G_1$ and $G \Rightarrow_{p2,m2} G_2$ of the same software artifact to obtain a combined result graph *H*. An essential distinction can be made between *structural* conflicts and *behavioural* conflicts.

When the parallel evolutions cannot be merged because the resulting graph would be ill-formed, we say that a **structural conflict** has occurred. Typical examples of this are *name conflicts* when the label or type of the same node or edge is modified twice,

or *dangling references* when a node is removed while independently an edge to this node was added. Formally, a structural conflict is defined by a breach of an application condition, because p_1 is not applicable after p_2 or vice versa. In other words, we have a structural conflict if both productions are not parallel independent. In [19], a complete characterisation is given of the different kinds of structural conflicts that can occur when merging two arbitrary primitive modification types. All possible conflicting combinations can be summarised in a conflict table in order to facilitate conflict detection.

4.3 Behavioural Conflict Warnings

Because structural conflicts can be detected by structure-oriented merge tools such as the one presented in [27], we will not discuss them further here. Instead, we will focus on another kind of conflicts that -as far as we know- cannot be detected by existing merge tools in a domain-independent way.

When two graph derivations $G \Rightarrow_{p1,m1} G_1$ and $G \Rightarrow_{p2,m2} G_2$ do not give rise to a structural conflict, they are parallel independent. This means that they can be sequentialised to $G \Rightarrow_{p1,m1} G_1 \Rightarrow_{p2,n2} H$ or $G \Rightarrow_{p2,m2} G_2 \Rightarrow_{p1,n1} H$ (where $n_1 = p_2^* \circ m_1$ and $n_2 = p_1^* \circ m_2$). Both cases lead to the same unique merged graph H because of the local confluence property for conditional derivations. Nevertheless, the merged graph can still contain some behavioural incompatibilities because of unexpected interactions between both productions. If this is the case, we say that a **behavioural conflict** has occurred.

Because it is inherently undecidable to determine whether the merge of two parallel evolution steps is behaviourally correct, we can only take a conservative approach towards detecting behavioural conflicts. Therefore, our formalism generates conflict warnings rather than detecting actual conflicts. As an example, consider Fig. 9, where the graph G in the middle is modified in parallel by two graph derivations $G \Rightarrow_{p1,m1} G_1$ and $G \Rightarrow_{p2,m2} G_2$. The first derivation has already been explained in Fig. 8. It adds an *«accesses»*-edge from *area* to *radius*, to indicate that the *area* is calculated from the *radius*. The second derivation takes an alternative approach, by deriving *area* from *circumference* (the *area* of a *Circle* can be calculated by integrating its *circumference*). This is represented by a primitive modification type p_2 = *AddEdge(ε,area,circumference,«invokes»)*. In the result graph H obtained by merging both derivations, *area* suddenly accesses *radius* via two different paths. Once directly, and once by way of *circumference*. This is clearly not the intention, since both modifications were introduced for the same purpose, namely providing an implementation of *area*. This particular behavioural merge conflict is called a *double reachability conflict*. To resolve the conflict, we need to decide which of both modifications is the most appropriate, and remove the other one.

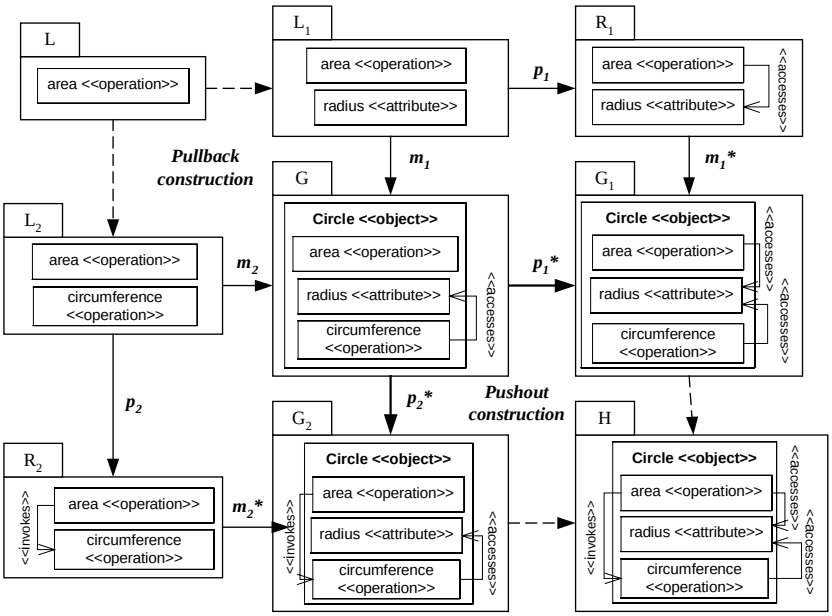

Fig. 9. Example of a double reachability merge conflict

Formally, the notion of behavioural conflict can be defined using the category-theoretical notions of *pushout* and *pullback*. The merge of two graph derivations $G \Rightarrow_{p1,m1} G_1$ and $G \Rightarrow_{p2,m2} G_2$ is defined by the pushout of the two corresponding morphisms $p_1^*: G \rightarrow G_1$ and $p_2^*: G \rightarrow G_2$. A potential behavioural conflict occurs if the pullback of the matches $m_1: L_1 \rightarrow G$ and $m_2: L_2 \rightarrow G$ is not empty, i.e., if the two graph derivations make parallel changes involving the same element. In the example of Fig. 9, we see that the pullback contains the node *area*, which is exactly the node in which the double reachability conflict occurred.

The above definition of behavioural conflict is too coarse-grained, in the sense that it does not give much feedback on why there is a problem or how the conflict can be resolved. Therefore, in [19] a finer-grained characterisation of behavioural conflicts is given, by comparing each pair of primitive modification types that gives rise to a conflict. For example, a *double reachability conflict* arises if we have two *AddEdges* p_1 and p_2 that each add a different edge with the same source and target node.[1] Similarly, a *cycle introduction conflict* arises when we have two *AddEdges* p_1 and p_2 that add an edge in the opposite direction between the same two nodes. Obviously, merging both modifications leads to the unanticipated introduction of cycles in the graph. When dealing with evolution of source code this conflict (which is sometimes called *unanticipated recursion*) is often difficult to detect, especially when no

[1] In the example of Fig. 9, this is only the case if we take the ***transitive closure*** of all edges into account as well, which is a straightforward extension of the formalism.

disciplined approach towards software evolution is taken. Yet another kind of behavioural conflict is the *inconsistent target conflict* illustrated in Fig. 1.

An alternative to detecting behavioural conflicts by comparing couples of primitive modification types is to go and look for the occurrence of *graph patterns* in the merged result graph [19]. This approach has the advantage that it is more scalable, because it does not rely on the specific modification types that have been used.

4.4 Domain-specific Customisations

In practice, many unnecessary behavioural conflict warnings will be generated. To reduce the number of unnecessary warnings, one can resort to more sophisticated conflict detection techniques that take more semantical information into account [2, 3]. We take a similar approach, by fine-tuning the formalism to different domains, and making use of domain-specific knowledge to remove some of the unnecessary conflict warnings.

Customisation of the formalism to a specific domain is straightforward thanks to the use of type graphs. In Fig. 2, 3 and 4 we illustrated this by representing two kinds of UML diagrams, namely a class diagram and a collaboration diagram, as well as their corresponding type graphs. For each domain, we also need to provide domain-specific modifications, and specify how to translate them in terms of the primitive modification types. For example, we can define *AddClass* and *AddOperation* in terms of *AddNode*, and *AddAssociation* in terms of *AddEdge*. This allows us to remove certain unnecessary conflict warnings, such as the cycle introduction conflict which can be ignored in the case of adding associations. Additionally, domain-specific well-formedness constraints allow us to capture some of the detected behavioural conflicts as breaches of these constraints, thus turning a behavioural conflict into a structural one. This is for example the case for a cyclic introduction of «isa»-edges, which gets captured by the well-formedness constraint that the inheritance hierarchy should be acyclic.

5. Scalability

5.1 Nesting

The six primitive modification types explained in section 4.1 are insufficient to describe all possible modifications of a nested graph. Therefore, we introduce three new primitive modification types specifically for changing the nesting relationship between existing nodes. *Promotion* can be used to pull a node up one level in the nesting hierarchy, and *Demotion* to push a node one level lower. *MoveNode* is used to move a nested node inside a new parent node at the same level as its current parent node. For example, in Fig. 5 we could use *MoveNode(Circle,area,Geo)* to move the *area* node from *Circle* to *Geo*.

Obviously, the new primitive modification types for nesting give rise to new merge conflicts, although we will not discuss them here.

5.2 Composite Modification Types

Because the primitive modification types are too elementary to be practically useful, we need to predefine a number of frequently recurring derivation sequences. For example, we could define *RedirectSource(center,Circle,Point,«uses»,Geo)* as the sequence *DropEdge(center,Circle,Point,«uses»), AddEdge(center,Geo,Point,«uses»)* in Fig. 5. *RedirectSource* is called a **composite modification type**. Similarly, the entire modification *p* of Fig. 5 can be expressed as a composite modification type *CreateSuperclass(Circle,Triangle,Geo)* which is composed out of:

AddNode(Geo,«class»), RedirectSource(center,Circle,Point,«uses»,Geo),
MoveNode(Circle,area,Geo), MoveNode(Circle,circumference,Geo),
DropEdge(center,Triangle,Point,«uses»),
DropNode(Triangle,area), DropNode(Triangle,circumference),
AddEdge(ε,Circle,Geo,isa), AddEdge(ε,Triangle,Geo,isa).

The advantage of composite modification types is that they allow us to fine-tune the conflict detection mechanism. It becomes possible to disregard certain behavioural conflict warnings if they occur in certain composite modification types by making use of the fact that its primitive constituents always appear in a particular combination with each other.

5.3 Normalisation

Another way to scale up the approach is by introducing a *normalisation algorithm*. It has two important purposes. First, it removes redundancy in an arbitrary evolution sequence, such as a node that is added and removed again. Second, it rearranges all derivations in the sequence in a canonical form, by putting all modifications of the same type together in a certain order. Formally, the algorithm is based on the notion of *sequential independence* of conditional graph derivations. The current implementation uses some kind of enhanced bubble-sort algorithm.

Normalisation has many advantages. It compacts arbitrary evolution sequences, thus reducing space and complexity. Another side-effect is that less unnecessary behavioural conflict warnings will be generated during conflict detection. Finally, the canonical form of the resulting derivation sequence is easier to understand, and allows us to answer questions like “Is node *v* removed during this particular evolution sequence?” in an efficient and straightforward way.

6. Conclusion

6.1 Summary

In this paper we explained how the formalism of reuse contracts could be defined on top of conditional graph rewriting with labelled typed nested graphs. This made it possible to deal with evolution of software artifacts in an intuitive and scalable way. More specifically, the approach is useful to detect structural and behavioural conflicts when merging parallel evolutions of the same software artifact.

An essential feature of our domain-independent formalism for software evolution is that it can be customised relatively easily to particular domains of software development. It suffices to specify the domain-specific type graph, express the domain-specific modification operations in terms of primitive or composite modification types, and determine which behavioural conflicts may be disregarded in particular situations. We illustrated our ideas on UML class diagrams and collaboration diagrams, but the approach is generally applicable to any other domain of software development where evolution is important.

6.2 Related Work

Because our approach provides support for detecting conflicts when merging parallel modifications of the same software artifact, it can be seen as an extension of existing merge techniques. Commercially available merge tools work on a purely *textual* basis [22], not taking into account any syntactic or structural information imposed by the programming language. As a result, they only detect physical conflicts, where the same line of code is modified in parallel by different software developers. Some alternatives take more *structural* information into account, such as the visual differencing tool for UML that comes with Rational Rose, and the domain-independent tool proposed in [27] which works with abstract syntax graphs. None of the existing tools seem to deal with *behavioural conflicts*, because this requires more *semantical* information, which is not considered due the additional complexity it gives rise to. One notable exception is [2], where a language-independent formalism is proposed to merge changes of programs based on the semantics rather than the concrete representation. Compared to our approach, the formalism proposed there is significantly more complex, and because of its abstractness it is unable to diagnose and locate conflicts between changes in the concrete representation of the program.

In [28], a category-theoretical approach towards software evolution is given. Although it doesn't specifically use graph rewriting, it contains some similarities to our work. However, the approach restricts itself to software specifications only, and doesn't discuss the important topic of merge conflicts.

There are many transformational approaches to describe the evolution of software artifacts. For example, [1] proposes a number of operations to transform *object-oriented database schemas*, while [21] proposes a number of behaviour-preserving

refactoring transformations for *object-oriented applications*. All these approaches, however, are dedicated to a specific domain, and do not deal with merge conflicts.

Another related area of research that relies on graph rewriting is *dynamic evolution* (or *reconfiguration*) of *software architectures*. While graphs are used to formally represent architectural components and their interconnections, graph rewriting can be used to manage dynamic changes or reconfigurations [16, 20, 25, 26]. An attempt to apply our approach to software architectures is undertaken in [23].

6.3 Future Work

Although we have already performed some basic experiments, we still need to validate our work in the context of a large industrial case study. Other necessary tasks are the integration of our approach in a CASE tool, and using our formalism for creating more sophisticated version control tools.

While the formalism explained in this paper is very promising, it can still be augmented in many ways. From a language point of view, one useful extension would be to add an *encapsulation* mechanism to nested graphs, to specify which nodes and edges are visible to the outside [9]. A *parameterisation* mechanism could also be introduced, to deal with concepts like template classes or template methods.

Another way to enhance the expressiveness of graphs would be to use *hyperedges*. On the one hand, this would allow edges that have more than one source node and target node. On the other hand, it would allow us to nest graphs into edges as well.

The type graphs we use are restricted to one level only. A slight generalisation would allow us to define type graphs of type graphs as well, and so on ad infinitum [9]. An interesting practical application of this would be to detect inconsistencies in UML diagrams when the UML metamodel itself evolves. To achieve this it suffices to apply our formalism on the level of type graphs.

For each of the generalisations proposed above, it needs to be checked whether they still preserve the formal requirements needed for our approach. Basically, this means that the definitions of pullback, pushout, parallel and sequential dependence, and the confluence property should still be valid.

Acknowledgements. I thank Andy Schürr, Bernard Westfechtel and the anonymous referees for their useful suggestions. I also thank my colleagues at the Programming Technology Lab for their support. Finally, I thank the participants of the Agtive workshop for their interesting remarks.

References

1. Banerjee, J., Chou, H.-J., Kim, H. J., Korth, H. F.: Semantics and implementation of schema evolution in object-oriented databases. SIGMOD RECORD, Vol. 16(3). ACM Press (1987) 311-322
2. Berzins, V.: Software merge: semantics of combining changes to programs. TOPLAS, Vol. 16(6). ACM Press (1994) 1875-1903
3. Binkley, D., Horwitz, S., Reps, T.: Program Integration for Languages with Procedure Calls. ACM Transactions on Software Engineering and Methodology, Vol. 4(1). ACM Press (1995) 3-35
4. Conradi, C., Westfechtel, B.: Version Models for Software Configuration Management. ACM Computing Surveys, Vol. 30(2). ACM Press (1998).
5. Corradini, A., Montanari, U., Rossi, F.: Graph processes. In [8] (1996) 241-265
6. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, J.: The category of typed graph grammars and their adjunction with categories of derivations. In [11] (1996)
7. Ehrig, H., Habel, A., Kreowski, H.-J., Parisi-Presicce, F.: From graph grammars to high-level replacement systems. In [10] (1991) 269-291
8. Engels, G., Ehrig, H., Rozenberg, G. (eds.): Special issue on graph transformations. Fundamenta Informaticae, Vol. 26(3,4). IOS Press (1996)
9. Engels, G., Schürr, A.: Encapsulated hierarchical graphs, graph types and meta types. Joint Compugraph/Semagraph Workshop on Graph Rewriting and Computation. Electronic Notes in Theoretical Computer Science, Vol. 2. Elsevier (1995)
10. Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.): Proceedings 4th International Workshop on Graph Grammars and their Application to Computer Science. Lecture Notes in Computer Science, Vol. 532. Springer-Verlag (1991)
11. Proceedings 5th International Workshop on Graph Grammars and their Application to Computer Science. Lecture Notes in Computer Science. Springer-Verlag (1996)
12. Proceedings 6th International Workshop on Theory and Application of Graph Transformation. Lecture Notes in Computer Science. Springer-Verlag (1998)
13. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. In [8] (1996) 287-313
14. Heckel, R.: Algebraic graph transformations with application conditions. Dissertation, Technische Universität Berlin (1995)
15. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars – A constructive approach. Lecture Notes in Theoretical Computer Science, Vol. 1. Elsevier Science (1995)
16. Hirsch, D., Inverardi, P., Montanari, U.: Modelling software architectures and styles with graph grammars and constraint solving. In Donohoe, P. (ed.), Software Architecture. Kluwer Academic Publishers (1999)
17. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science, Vol. 109 (1993) 181-224
18. Lucas, C.: Documenting reuse and evolution with reuse contracts. PhD Dissertation, Department of Computer Science, Vrije Universiteit Brussel, September (1997)
19. Mens, T.: A formal foundation for object-oriented software evolution. PhD Dissertation, Department of Computer Science, Vrije Universiteit Brussel, September 1999.
20. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering, Vol. 24(7). IEEE Press (1998) 521-553
21. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD Thesis, University of Illinois at Urbana-Champaign, Technical Report UIUC-DCS-R-92-1759 (1992)

21. Poulovassilis, A., Levene, M.: A nested-graph model for the representation and manipulation of complex objects. ACM Transactions on Information Systems, Vol. 12(1). ACM Press (1994) 35-68
22. Rigg, W., Burrows, C., Ingram, P.: Configuration management tools. Ovum Ltd. (1995)
23. Romero, N.: Managing architectural evolution with reuse contracts. Masters Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1999.
24. Steyaert, P., Lucas, C., Mens, K., D'Hondt, T.: Reuse contracts - Managing the evolution of reusable assets. In proceedings of OOPSLA '96. ACM SIGPLAN Notices, Vol. 31(10), ACM Press (1996) 268-286.
25. Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In [12] (1998)
26. Wermelinger, M.: Towards a chemical model for sofware architecture reconfiguration. IEEE Proceedings – Software, Vol. 145(5). IEEE Press (1998) 130-136
27. Westfechtel, B.: Structure-oriented merging of revisions of software documents. In proceedings of 3rd International Workshop on Software Configuration Management. ACM Press (1991) 68-79
28. Wiels, V., Easterbrook, S.: Management of evolving specifications using category theory. In proceedings of Automated Software Engineering Conference '98. IEEE Press (1998) 12-21