Transformation dependency analysis A comparison of two approaches

T. Mens[†] — **G. Kniesel**[‡] — **O. Runge**[*]

[†] *Service de Génie Logiciel, Université de Mons-Hainaut*
Av. du Champ de Mars 8, 7000 Mons, Belgium
tom.mens@umh.ac.be

[‡] *Computer Science Department III, University of Bonn*
Römerstr 164, 53117 Bonn, Germany
gk@informatik.uni-bonn.de

[*] *Technische Universität Berlin, 10587 Berlin, Germany*
olga@cs.tu-berlin.de

ABSTRACT. *Transformation dependency analysis is crucial to provide better tool support for current-day software development techniques – two prominent examples are program refactoring and model transformation. Unfortunately, it is unclear how existing tools that provide generic support for these techniques relate to each other, due to their difference in terminology, concepts and formal foundations (graphs versus logic). This article reports on the results of an experimental comparison between two tools:* AGG *and* Condor. *Among others, we noticed a performance advantage of several orders of magnitude for the logic-based approach.*

RÉSUMÉ. *L'analyse de dépendance de transformations est essentiel pour améliorer le support de techniques actuelles de développement logiciel – deux exemples sont la restructuration de programmes et la transformation de modèles. Or, il est peu clair comment les outils de support actuels peuvent être comparés, à cause d'une différence de terminologie, de concepts utilisés, et de fondations formelles (graphes versus logique). Dans cet article nous discutons d'une comparaison experimentale de deux outils :* AGG *et* Condor. *Entre autres, nous avons trouvé un gain d'efficacité de plusieurs ordres de magnitude en utilisant l'approche basée sur la logique.*

KEYWORDS: *software evolution, conditional transformation, dependency analysis, graph transformation, logic programming.*

MOTS-CLÉS : *évolution logicielle, transformation conditionnelle, analyse des dépendances, transformation de graphes, programmation logique.*

1. Introduction

Program transformation has been, and still remains, an essential ingredient of software engineering, as all compiler technology heavily depends on it. An active research domain within program transformation is the activity of *refactoring*, whose goal is to improve the software structure while preserving its external behaviour (Fowler, 1999). Another important research trend is *model transformation*, which is becoming increasingly relevant with the emergence of model-driven architecture and model-driven software engineering. As observed by (Sendall *et al.*, 2003), model transformation is the heart and soul of model-driven engineering.

One of the key challenges in any transformation-based approach is the composability issue: given a set of transformations one would like to apply, are there any dependencies between these transformations that dictate a particular order of use, are there any mutual dependencies that preclude their joint use, and if not, is it possible to come up with an 'optimal' transformation sequence (or even several)?

Given the increased importance of software transformation at all levels of abstraction, the goal of this paper is to investigate to which extent the analysis of dependencies between software transformations can be supported by state-of-the-art tools. We are aware of only two tools with the ability to perform transformation dependency analysis in a language-independent manner: *AGG* (Taentzer, 2004) and *Condor* (Kniesel *et al.*, 2003).

The crucial idea enabling transformation dependency analysis in both tools is that each transformation T can be specified in terms of a set of *preconditions* and a set of *postconditions*. The preconditions are conditions that need to be satisfied before the transformation can be applied, whereas the postconditions are those conditions that are valid after application of the transformation.

Unfortunately, this is as far as the commonalities go. The terminology, concepts and formal foundations used by both tools are completely disjoint making them hard to compare. *AGG* is based on graph theory and provides means to detect *critical pairs*, *parallel conflicts* and *sequential dependencies*. *Condor* is based on logic and provides means to detect *triggering* and *inhibition* dependencies. A software engineer who is not an expert in both graph transformation theory and logic would have a hard time deciding which tool is most appropriate for his tasks.

For ultimate insight, a thorough formal comparison of the theory underlying the two tools would be necessary. However, given the heterogeneity of the approaches, this is a rather challenging task. In this article we show that highly useful results can be obtained based on a purely experimental comparison. To software engineers these results give practically useful guidelines for the selection of a tool. To researchers they open up some new research questions and help to focus future, more formal comparisons, on the most interesting differences.

The remainder of this article is structured as follows. In Section 2 we substantiate the need for dependency analysis with a motivating example from the domain of pro-

gram refactoring. Then we describe our experimental setup in terms of a simple, yet sufficiently generic, case study and a list of objective comparison criteria (Section 3). In Section 4 and 5 we introduce the basic concepts of each tool and describe its use on the common example. Finally, we evaluate both tools based on the selected comparison criteria (Section 6). We show that both tools yield the same results but differ in subtle ways in their expressive power. Most surprisingly, we found a performance advantage of several orders of magnitude for the logic-based approach.

2. Motivating example: Program refactoring

As a motivating example of the need for transformation dependency analysis, consider the example of program refactorings, which are transformations that change the program structure while preserving its external behaviour (Fowler, 1999; Mens *et al.*, 2004).

The ability to analyse dependencies between refactorings will allow tool developers to improve refactoring support in a significant way. To illustrate this, consider the following scenario. Assume that a software architect wants to restructure a software application and knows exactly which refactorings she wants to apply. She might even have an idea of an application order that might seem intuitive. However, with any non-trivial refactorings and any refactoring sequence of non-trivial length it will be impossible to confirm her intuition. There might be non-obvious influences of one refactoring on another that impose a particular order of application, or that might even prevent the joint application of these refactorings.

Ideally, a tool based on transformation dependency analysis will be able to provide such concrete feedback, by computing all possible dependencies and conflicts between refactorings. For example, it would be able to determine whether a user-defined selection of refactorings is applicable for a given program, and it would propose an optimal order (i.e., one satisfying all dependencies) in which to apply these refactorings. Similar analyses are even possible in a program-independent way, helping to determine generally applicable refactoring sequences, when assembling larger refactorings from smaller ones in a refactoring editor (Kniesel *et al.*, 2004).

This scenario is not just a sketch of an idea. Both tools that we compared are able to express and analyze program refactorings. (Kniesel *et al.*, 2004) describe how to represent refactorings as sequences of conditional transformations. A representative selection of refactorings has been expressed as graph transformations in *AGG*[1]. This selection constitutes a number of typical program refactorings: PullUpMethod, PullUpVariable, MoveMethod, MoveVariable, RenameClass, RenameMethod, RenameVariable, AddParameter, RemoveParameter, CreateSuperclass. The detection of sequential dependencies among these refactorings with the aid of *AGG* is reported in

1. When downloading AGG from its webpage `http://tfs.cs.tu-berlin.de/agg`, the refactoring example can be found in the Examples folder.

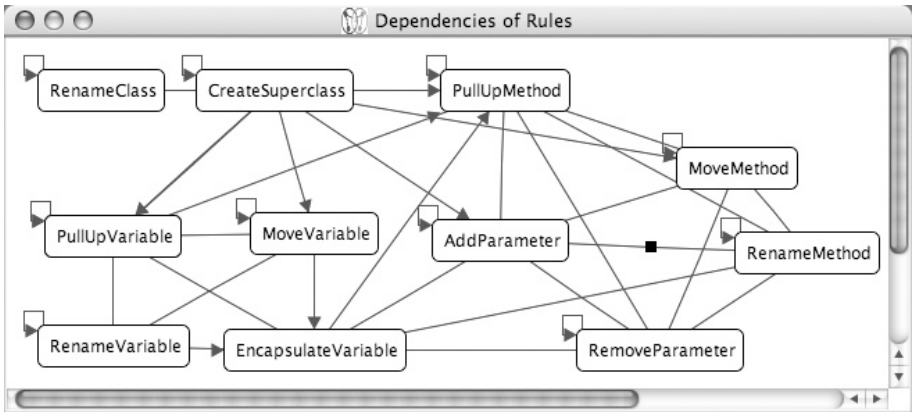

Figure 1. *'Sequential dependencies' for a representative set of refactorings.*

(Mens *et al.*, 2005) and (Mens *et al.*, 2006). The dependency graph showing all such dependencies computed by *AGG* version 1.3 is shown in Figure 1.[2]

3. Experimental Setup

3.1. *Common Example*

We initially started our comparison by implementing the aforementioned refactorings as conditional transformations. Technically this was fast and easy. However, we quickly realised that it was not a good approach for a comparison, since we spent by far most of the time on discussing and settling some subtleties in the semantics of the individual refactorings in order to make sure that both implementations were indeed equivalent. This motivated us to base our comparison on a much simpler, canonical example with a well-understood, indisputable semantics.

For our experiment, we therefore relied on the canonical representation of software artefacts as labelled, typed graphs, introduced in the PhD thesis of Tom Mens (Mens, 1999). Each node and edge in a graph has a name and a type (which are both represented as simple strings). The name of each node must be unique in the entire graph. Edges with the same source and target node must also have a unique name. Types need not be unique, i.e., more than one node (or edge) can have the same type.

Based on this representation, we implemented in both tools the following minimal set of transformations: *AddNode*, *DeleteNode*, *AddEdge*, *DeleteEdge*, *RetypeNode*, *RetypeEdge*, *RenameNode* and *RenameEdge*. These 8 transformations are very

2. An explanation of 'sequential dependencies' is given in Section 4.

simple, but together they allow one to make arbitrarily complex changes to any given graph structure. Any complex transformation can always be decomposed into a sequence of these basic transformations.

3.2. *Comparison Criteria*

In order to compare the tools *AGG* and *Condor*, we used the following objective criteria:

– *Expressiveness.* How expressive are the tools? Are there certain transformations that cannot be expressed due to the particular notation or language imposed by the tool?

– *Precision.* How precise are the results obtained with the tool? Does the tool generate any false positives or false negatives?

– *Genericity.* How generic is the tool? How easy is it to use the tool for representing and transforming different types of software artefacts?

– *Performance.* What is the performance of the tool?

– *Mechanisms.* What are the underlying mechanisms used to perform transformation analysis?

Section 4 introduces the basics of graph transformations and explores the *AGG* tool using this setup. Section 5 then explains and evaluates the *Condor* tool in the same way. Section 6 compares both tools according to the avobe criteria. Finally, section 7 concludes.

4. *AGG*: A graph-transformation based approach

In this section we give a short, informal introduction to the relevant notions from graph transformation theory, present *AGG* and describe its use on our evaluation scenario.

4.1. *Graph transformation*

In graph transformation theory (Ehrig *et al.*, 1999), a graph transformation T is an abstract transformation rule that can be applied in different contexts. Given a concrete graph G, the transformation T may be applied in different ways, because there may be different *matches* of its left-hand side in G. The actual application of a graph transformation is therefore uniquely defined by the combination of the transformation rule T and its match m in the concrete context graph G.

A *parallel conflict* between two transformations T_1 and T_2 occurs if T_1 and T_2 can both be applied to the same host graph (with matches m_1 and m_2, respectively), but after applying T_1 (with match m_1) it is no longer possible to apply T_2 (with match

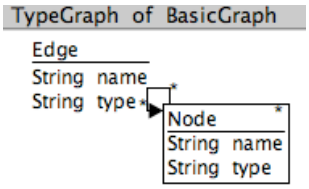

Figure 2. *The* AGG *type graph for the experimental setup. Restrictions regarding the uniqueness of names are expressed as additional graph constraints (not shown here).*

m_2). In other words, a parallel conflict simply means that T_1 and T_2 cannot be serialised in any given order. Parallel conflicts can be detected by comparing the preconditions of T_1 and T_2. If these preconditions overlap, then this implies that both transformations will make a change that can be in conflict with the other one.

A *sequential dependency* from transformation T_2 (with match m_2) to transformation T_1 (with match m_1) occurs if T_2 can be applied after application of T_1, but it is not possible to apply T_2 (using the same match m_2) without having performed T_1 first. Sequential dependencies can be detected by comparing the precondition of T_2 with the postcondition of T_1. This allows us to detect whether T_2 relies on information that is introduced by T_1, or that T_2 relies on the absence of certain items that have been removed by T_1.

The above definitions are informal ones. To be really precise, an introduction to graph transformation theory would be needed, but this is beyond the scope (and page limits) of this paper. For more formal details, we refer to (Heckel, 1995).

4.2. *Analysis with AGG*

In this section we explain how dependency analysis can be carried out with the *AGG* tool. AGG version 1.3.0 has been used for our experiments since the preceding versions did not support sequential dependency analyis yet.

As a first step, a type graph (i.e., a metamodel) needs to be specified. It is used to determine whether a given software artefact (in our case study, a graph structure) is well formed. The type graph for the given case study is shown in Figure 2.

As a second step we need to specify the 8 basic graph transformations in *AGG*. We only show the transformations AddEdge and RetypeEdge in Figure 3. Both consist of a left-hand side (representing the positive part of the precondition) and a right-hand side (representing the postcondition). The transformation is implicitly specified as the difference between the left-hand-side and the right-hand-side. In addition, a graph transformation may have any number of negative application conditions. For example, AddEdge also contains a negative precondition *EdgeNotPresent*, stating that

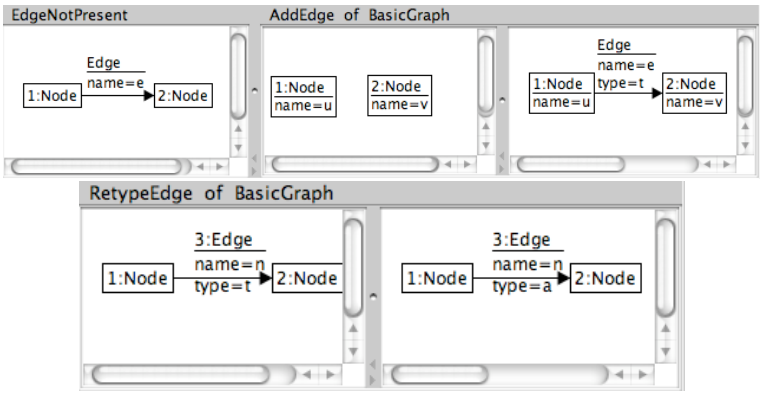

Figure 3. *Two graph transformations in* AGG*: AddEdge and RetypeEdge.*

it is not possible to introduce a new edge between two nodes if an edge with the same name already exists between those nodes. For more details about how to specify graph transformations we refer to the AGG webpage[3].

Detection of parallel conflicts is achieved in *AGG* by means of *critical pair analysis*. Each critical pair specifies a minimal graph representing a potential parallel conflict situation. Figure 4 shows the critical pair table generated by *AGG*. The numbers in each field of the table indicate the number of conflict situations that have been found between each pair of transformations. For example, there are two conflicts between AddEdge and DeleteNode, because the addition of an edge prevents the deletion of its source node and its target node. Likewise, there are two conflicts between DeleteNode and AddEdge, because the deletion of a node prevents the addition of a new edge having this node either as source or as target.

From the critical pair table, a conflict graph can be generated automatically, as shown in Figure 5.

AGG also allows the computation of sequential dependencies, the results of which are shown in tabular format in Figure 6. A sequential dependency is computed by inverting the first transformation, and finding its critical pairs with the second transformation. Observe that the table of sequential dependencies is not symmetrical. For example, AddEdge depends on AddNode (but not the other way around), since adding a node enables the introduction of a new edge to or from this node afterwards. Similarly, DeleteNode depends on DeleteEdge (but not the other way around) since removing an edge enables the removal of the node, if no other edges are connected to this node. Again, a dependency graph is generated automatically, as shown in Figure 7.

3. `http://tfs.cs.tu-berlin.de/agg`

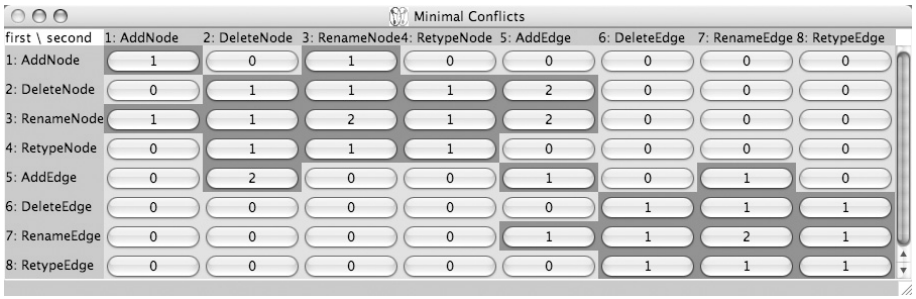

Figure 4. *Detection of potential parallel conflicts (i.e., critical pairs) in* AGG. *Note that the only asymmetric situation is between* AddEdge *and* RenameNode.

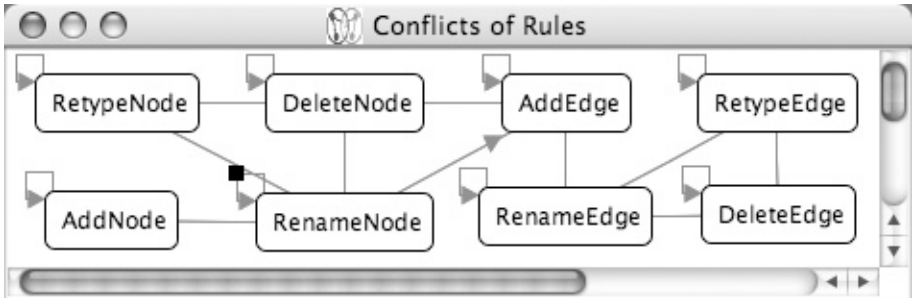

Figure 5. *Dependency graph showing all potential parallel conflicts (i.e., critical pairs) in* AGG. *Undirected edges are used to represent bidirectional conflicts.*

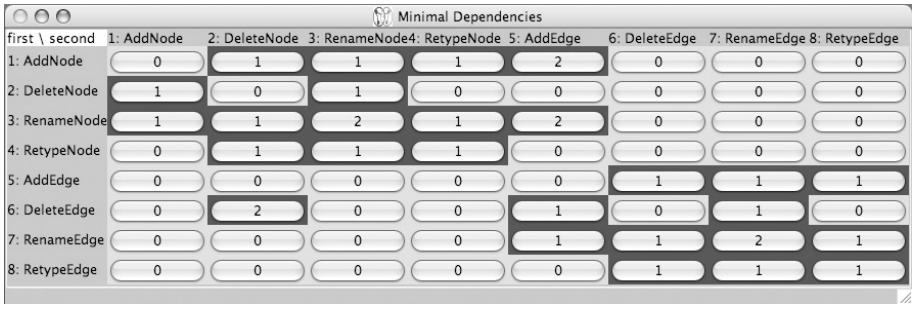

Figure 6. *Detection of potential sequential dependencies in* AGG.

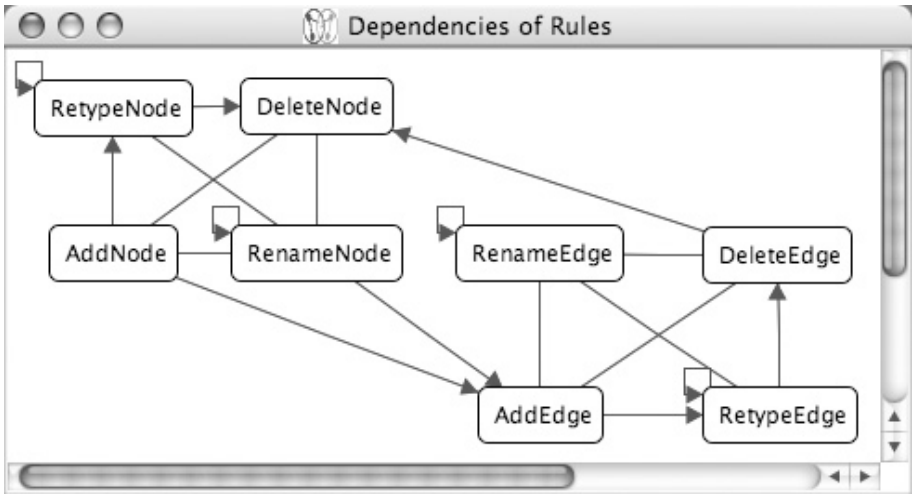

Figure 7. *Dependency graph showing all potential sequential dependencies in* AGG. *Undirected edges are used to represent bidirectional dependencies.*

Table 1 summarises all detected conflicts and dependencies. One can observe many situations that lead to both a conflict and a sequential dependency. As a concrete example, consider the two transformations RetypeEdge and DeleteEdge. On the one hand, both transformations give rise to a parallel conflict if they are applied to the same edge. If we try to delete the edge, and in parallel try to change its type, it is not clear how these two incompatible changes can be combined. There is also a sequential dependency, since the application of RetypeEdge gives rise to a new opportunity of applying DeleteEdge (namely, the deletion of the edge with the new type).

5. *Condor*: A logic-based approach

In this section we give a short, informal introduction to relevant notions of conditional transformations (CTs, for short). We explain the notions of inhibition dependency and triggering dependency underlying *Condor*, and we describe the use of the tool in our evaluation scenario.

5.1. *Conditional transformation*

Conditional transformations (Kniesel, 2006; Kniesel *et al.*, 2004) work on a representation of programs as logic terms. Every node in the generalized abstract syntax tree of a program is represented by a *P*rogram *E*lement *T*erm (PET). There are no restrictions on the structure of program element terms, making this approach applicable

Table 1. *Summary of parallel conflicts (C) and sequential dependencies (D) detected with* AGG *by means of critical pair analysis.*

1st|2nd	addN	delN	renameN	retypeN	addE	delE	renameE	retypeE
addNode	1 C	1 D	1 D 1 C	1 D	2 D			
deleteNode	1 D	1 C	1 D 1 C	1 C	2 C			
renameN	1 D 1 C	1 D 1 C	2 D 2 C	1 D 1 C	2 D 2 C			
retypeN		1 D 1 C	1 D 1 C	1 D 1 C				
addEdge		2 C			1 C	1 D	1D 1 C	1D
deleteEdge		2 D			1 D	1 C	1 D 1 C	1 C
renameE					1 D 1 C	1 D 1 C	2 D 2 C	1 D 1 C
retypeE						1 D 1 C	1 D 1 C	1 D 1 C

to arbitrary artefacts. The upper part of Table 2 shows a simple Java program and its representation as a set of PETs. The lower part shows the representation of a simple graph. For a PET representation of complete Java we refer to the *JTransformer* website[4]. In the term representation for Java chosen in Table 2, the first argument of every term is the unique identity of that node, and the second argument is the reference to its parent node. Top-down navigation is possible due to the relational nature of logic terms. For instance, the term block(5,3,[6]) can be seen as a parent reference from node 5 to node 3, or as a child reference from node 3 to node 5. Square brackets denote lists.

A conditional transformation (CT) is a program transformation guarded by a precondition, such that the transformation is performed on a given program only if its precondition is true. The precondition can be any closed logic formula containing conjunction, disjunction and negation. The transformation can be any sequence of additions, deletions or replacements of program element terms, corresponding to the primitive operations *add(PET)* (to add a new element), *delete(PET)* (to remove an existing element), and *replace(PETold,PETnew)* (to replace an existing element by another one).

4. *JTransformer* is a program transformation engine for Java based on conditional transformations. It is available as a plugin for *Eclipse* from `http://roots.iai.uni-bonn.de/`.

Original artefact	Term representation

`package demo;`
`class C {`
`  void m( String s) { m(s); }`
`}`
	`package(1, 0, 'demo')`
	`  class(2, 1, 'C')`
	`  method(3, 2, 'm' , void,[4],5)`
	`  param(4, 3, 's' , 100)`
	`  block(5, 3, [6])`
	`  call(6, 5, null, 3 )`
	`  ident(7, 6, 4 )`

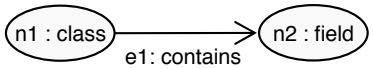	`graphNode(n1, class)`
	`graphNode(n2, field)`
	`graphEdge(e1, n1, n2, contains)`

Table 2. *Representation of Java programs and graphs as logic terms.*

Conditional transformations are specified as logic programs. Each conditional transformation corresponds to a clause of the form `ct(Head,Cond,Trans)`. In order to apply the CT with the head `Head`, the preconditon `Cond` is verified, yielding a set of substitutions for the logic variables contained in it. Then the transformation`Trans` is performed for every computed substitution. Postconditions are not represented explicitly since they can be derived automatically from the precondition and the transformation.

5.2. *CT-based dependency analysis*

The analysis of dependencies between conditional transformations is based on comparing preconditions and postconditions. Given two conditional transformations ct_1 and ct_2. If a literal l in the postcondition of ct_1 is unifiable with a literal in the precondition of ct_2 then executing ct_1 will contribute to making the precondition of ct_2 true. In this case ct_1 is said to potentially *trigger* (enable) ct_2. Similarly, if a literal l in the postcondition of ct_1 is unifiable with a negated version of it in the precondition of ct_2 then executing ct_1 will contribute to making the precondition of ct_2 false. In this case ct_1 is said to potentially *inhibit* (prevent) ct_2.

Both of these dependencies imply a sequential ordering of CT executions. If ct_1 triggers or inhibits ct_2 then ct_1 must be executed before ct_2. In the case of triggering, this guarantees that when ct_2 is executed all the PETs relevant for it already exist. In the case of inhibition, it guarantees that when ct_2 is executed, all the PETs that should *not* be consumed by ct_2 have already been removed. If CTs are executed in the order indicated by the dependency graph, no CT ever needs to be repeated again because of later addition of more triggering PETs and no CT needs to be undone because of later removal of PETs assumed to exist.

There is no special analysis for 'parallel conflicts' in this framework. It is not necessary since the dependency graph formed by inhibition and triggering dependencies

already contains all the relevant information. If the graph is acyclic, its topological sorting automatically determines at least one conflict-free order of the involved CTs. If the graph contains pure triggering cycles, the analysis indicates that the execution of the CTs in these cycles must be iterated until a fixpoint is reached. If the graph contains pure inhibition cycles the analysis diagnoses a conflict, since (direct or indirect) mutual inhibition represents a contradiction. If the dependency graph contains mixed cycles the programmer receives a warning since it is not possible to decide automatically whether these cases are conflicts or acceptable situations.

5.3. *Using Condor*

Condor[5] (Kniesel *et al.*, 2003; Bardey, 2003), is the tool for CT-based dependency analysis developed at the University of Bonn. For our experiment, version 0.3 of its recent reimplementation has been used.

Using *Condor* to analyse CTs starts by determining a term representation of the artefacts to be analyzed and transformed. For our experiment we used the PETs illustrated in the lower part of Table 2. A node with name n and type t is represented as `graphNode(n,t)`. An edge with name e, type t, source node $n1$ and target node $n2$ is represented as `graphEdge(e,n1,n2,t)`.

The conditional transformations *addEdge* and *retypeEdge* on this representation are shown below (compare Figure 3):

```
ct( addEdge(Edge, N1, N2, Type),             % Head
  ( graphNode(N1,_),                         % Condition
    graphNode(N2,_),
    not(graphEdge(Edge, N1, N2, _)) ),
  ( add(graphEdge(Edge, N1, N2, Type))       % Transformation
  ) ).

ct( retypeEdge(Edge, N1, N2, Type, TypeNew), % Head
  ( graphEdge(Edge, N1, N2, Type) ),         % Condition
  ( replace(graphEdge(Edge, N1, N2, Type),   % Transformation
            graphEdge(Edge, N1, N2, TypeNew))
  ) ).
```

Applying *Condor* to the 8 CTs corresponding to our test example, the 32 triggering dependencies and 32 inhibition dependencies shown in Figure 8 and 9 were found.

5. Web page `http://roots.iai.uni-bonn.de/research/condor/`

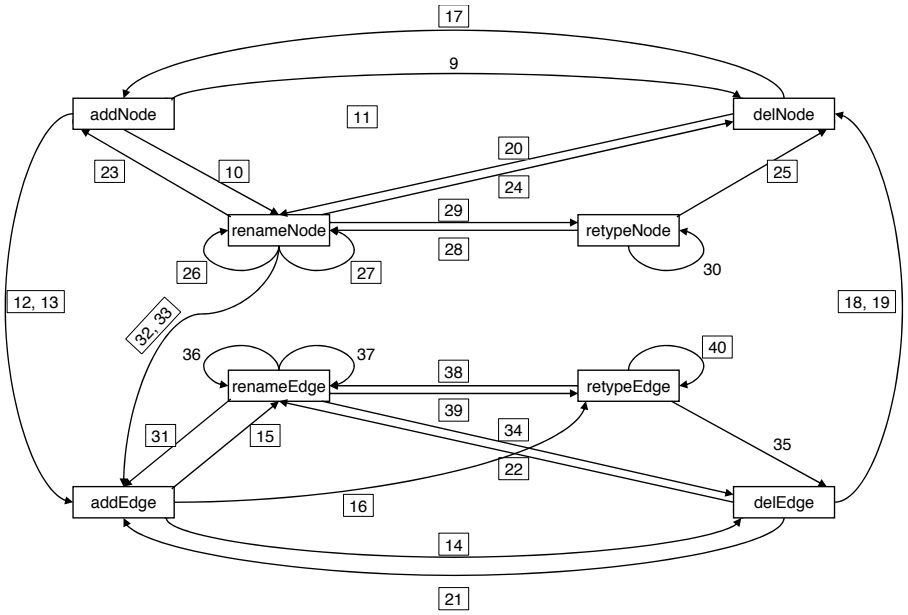

Figure 8. *Dependency graph showing all potential triggerings detected by Condor.*

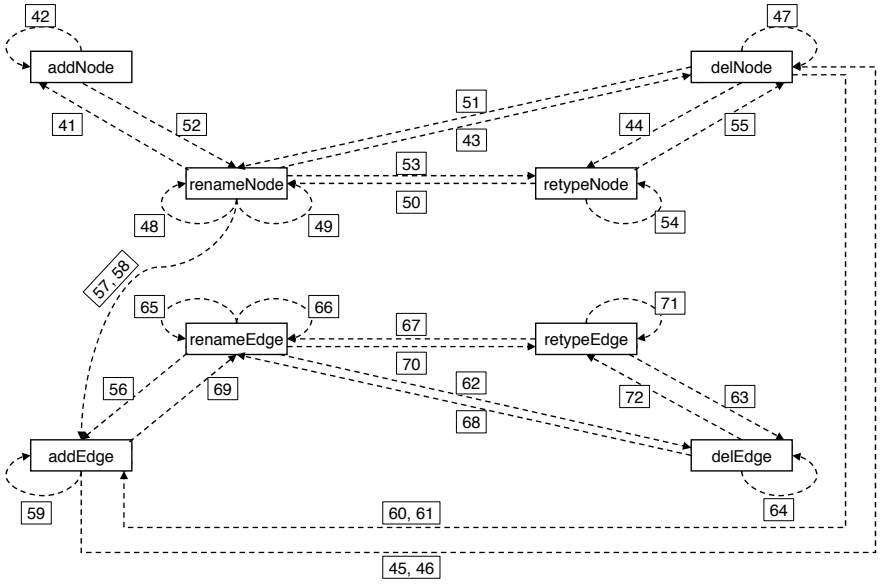

Figure 9. *Dependency graph showing all potential inhibitions detected by Condor.*

6. Comparison

When summarising the results of all dependencies generated by *Condor* in tabular format, we get exactly the same table as the one computed for *AGG* in Table 1. Both tools identified 32 conflicts (resp. inhibition dependencies) and 32 sequential (resp. triggering) dependencies in exactly the same places.

Nevertheless, it took us a couple of iterations before we obtained this one-to-one correspondence between *Condor* and *AGG*. This was mainly due to the fact that both tools use a different representation for expressing the transformations. Because of that, it required a learning process to understand the, often subtle, differences in terminology. In the successive iterations, we gradually removed all false positives and false negatives that occurred as a side-effect of our lack of understanding of these terminological differences.

The first difference was the notion of conflict and dependency used by both tools. According to our experiment an *inhibition dependency* in *Condor* seems to correspond to the notion of a *critical pair (conflict)* in *AGG*. Similary, a *triggering dependency* in *Condor* appears to corresponds to the notion of a *sequential dependency* in *AGG*.

Another distinction concerns the categories of dependencies. In *Condor*, both triggering and inhibition dependencies are classified according to the basic transformation that triggered the dependency: *add*, *delete* and *replace*. In a similar vein, the critical pairs detected in *AGG* can be classified into three conflict categories: *produce/forbid* conflict, *delete/use* conflict, and *change-attribute* conflict. Analysis revealed that there is a one-to-one correspondence between these categories and the ones used by *Condor*. One notable exception to this rule is the triggering dependency of type *add* between AddEdge and DeleteNode, which was identified by *AGG* as a *produce/dangling edge* conflict. Without going into formal details, this has to do with the particular way in which the algebraic approach to graph transformations treats dangling edges.

Below, we provide a comparison of both tools based on the objective criteria put forward in section 3. A summary of this comparison is given in Table 3.

– *Genericity.* Both *AGG* and *Condor* are generic in the sense that they can be used to represent any kind of software artefact.

– *Precision.* As can be seen from the results, both *AGG* and *Condor* allow the detection of parallel conflicts. In fact, all inhibition dependencies detected by *Condor* coincide with all critical pairs detected by *AGG*. Similarly, all triggering dependencies detected by *Condor* coincide with all sequential dependencies detected by *AGG*. Hence, we are confident that the dependency analysis supported by both tools gives reliable results.

– *Expressiveness. AGG* requires to make an explicit difference between the left-hand side of a graph transformation and possible negative application conditions. In *Condor*, all these constraints are put together as a single collection of logic predicates. Negated logical predicates (using the keyword `not`) are typically used to represent negative conditions. This makes *Condor* more expressive than *AGG*, since it allows

Table 3. *Comparison of* AGG *and* Condor. *Only those comparison criteria that reflect a difference between both tools are shown here.*

criterion	*AGG*	*Condor*
technique used	graph transformation critical pair analysis	conditional transformation CT-based dependency analysis
mechanisms	graph matching and graph equivalence	unification and backtracking
performance	16 minutes	0,2 seconds
expressiveness	+	++
implementation language	*Java*	*Prolog*

the specification of any combination of conjunction, disjunction and negation. The graphical syntax of *AGG* prohibits certain of these combinations (one cannot use negation inside a negative application condition, for example).

– *Performance.* The difference in execution time needed to calculate all conflicts and dependencies in both tools was dramatic. In *AGG*, it took more than 15 minutes to compute all results, whereas *Condor* required less than a second! This difference is largely due to the inherent computational complexity of the underlying mechanisms used by both tools.

– *Mechanisms.* The basic mechanism used in *AGG* to apply transformations and to analyse dependencies between graph transformations is subgraph matching and graph equivalence. Unfortunately, this is known to be NP-complete. *Condor*, on the other hand, benefits from the linear complexity of unification.

7. Conclusion

Based on the experiments performed in this paper, we conclude that it is feasible to detect dependencies and conflicts within a given set of program transformations. Analysis of the corresponding dependency graph will allow us to provide better tool support for a wide range of software engineering activities, including program refactoring and model transformation. Being based on a language-independent representation (using either graphs or logic predicates), the analysed approaches are applicable to a wide range of software artefacts at any level of abstraction. Exploiting all the options provided by this generality remains a topic for future work.

Probably the most important thing we can learn from the comparison of *AGG* and *Condor* relates to their dramatic difference in performance. We experienced the same difference in performance when analysing more complex transformations, such as the program refactorings mentioned in section 2. It seems that the powerful mechanism of graph matching used by *AGG* is very inefficient and, as such, not very suitable for

the purpose of performing transformation dependency analysis. Indeed, the same analysis performed with *Condor* showed a performance improvement of several orders of magnitude. It seems that the lightweight mechanism of unification offered by *Prolog* suffices. As a consequence, it may be worthwhile to investigate in the future to which extent the use of logic-based conditional transformations provides an efficient alternative to graph transformation tools.

8. References

Bardey U., *« Abhängigkeitsanalyse für Programm-Transformationen »*, Diploma thesis, CS Dept. III, University of Bonn, Germany, February, 2003.

Ehrig H., Engels G., Kreowski H.-J., Montanari U., Rozenberg G. (eds), *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, 1999. Volumes 1–3.

Fowler M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

Heckel R., *« Algebraic Graph Transformations with Application Conditions »*, Master's thesis, Technische Universität Berlin, 1995.

Kniesel G., A Logic Foundation for Conditional Program Transformations, Technical report n° IAI-TR-2006-01, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, January, 2006.

Kniesel G., Bardey U., Static Dependency Analysis for Conditional Program Transformations, Technical report n° IAI-TR-03-03, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, August, 2003.

Kniesel G., Koch H., « Static composition of refactorings », *Science of Computer Programming*, vol. 52, n° 1-3, p. 9-51, 2004.

Mens T., A Formal Foundation for Object-Oriented Software Evolution, PhD thesis, Vrije Universiteit Brussel, Belgium, September, 1999.

Mens T., Taentzer G., Runge O., « Detecting structural refactoring conflicts using critical pair analysis », *Electronic Notes in Theoretical Computer Science*, vol. 127, n° 3, p. 113-128, April, 2005.

Mens T., Taentzer G., Runge O., « Analyzing Refactoring Dependencies Using Graph Transformation », *Software and Systems Modeling*, 2006. To appear.

Mens T., Tourwe T., « A Survey of Software Refactoring », *IEEE Transactions on Software Engineering*, vol. 30, n° 2, p. 126-162, February, 2004.

Sendall S., Kozaczynski W., « Model Transformation: The heart and soul of model-driven software development », *IEEE Software*, vol. 20, n° 5, p. 42-45, September-October, 2003.

Taentzer G., « AGG: A Graph Transformation Environment for Modeling and Validation of Software », *Proc. AGTIVE 2003*, vol. 3062 of *Lecture Notes in Computer Science*, Springer-Verlag, p. 446-453, 2004.

ANNEXE POUR LE SERVICE FABRICATION
A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LES ACTES :

   LMO 2006

2. AUTEURS :

   T. Mens† *— G. Kniesel*‡ *— O. Runge**

3. TITRE DE L'ARTICLE :

   Transformation dependency analysis
   A comparison of two approaches

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :

   Transformation dependency analysis

5. DATE DE CETTE VERSION :

   January 20, 2006

6. COORDONNÉES DES AUTEURS :

   – adresse postale :
   † Service de Génie Logiciel, Université de Mons-Hainaut
   Av. du Champ de Mars 8, 7000 Mons, Belgium
   tom.mens@umh.ac.be

   ‡ Computer Science Department III, University of Bonn
   Römerstr 164, 53117 Bonn, Germany
   gk@informatik.uni-bonn.de

   * Technische Universität Berlin, 10587 Berlin, Germany
   olga@cs.tu-berlin.de
   – téléphone : +33(0)4 92 94 27 48
   – télécopie : +33(0)4 92 94 28 96
   – e-mail : Roger.Rousseau@unice.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :

   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.22 du 04/10/2005.

8. FORMULAIRE DE COPYRIGHT :

   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`