Queensland University of Technology
Brisbane Australia

This may be the author's version of a work that was submitted/accepted for publication in the following source:

Hearnden, David, Lawley, Michael, & Raymond, Kerry
(2006)
Incremental Model Transformation for the Evolution of Model-Driven Systems.
Lecture Notes in Computer Science, *4199*, Article number: MoDELS 2006321-335.

This file was downloaded from: https://eprints.qut.edu.au/225037/

© Consult author(s) regarding copyright matters

This work is covered by copyright. Unless the document is being made available under a Creative Commons Licence, you must assume that re-use is limited to personal use and that permission from the copyright owner must be obtained for all other uses. If the document is available under a Creative Commons License (or other specified license) then refer to the Licence for details of permitted re-use. It is a condition of access that users recognise and abide by the legal requirements associated with these rights. If you believe that this work infringes copyright please provide details by email to qut.copyright@qut.edu.au

Notice: *Please note that this document may not be the Version of Record (i.e. published version) of the work. Author manuscript versions (as Submitted for peer review or as Accepted for publication after peer review) can be identified by an absence of publisher branding and/or typeset appearance. If there is any doubt, please refer to the published source.*

https://doi.org/10.1007/11880240_23

QUT Digital Repository:
http://eprints.qut.edu.au/

This is the author's version published as:

Hearnden, David, Lawley, Michael, & Raymond, Kerry (2006) ***Incremental Model Transformation for the Evolution of Model-Driven Systems.*** In: 9th International Conference, MoDELS 2006, October 1-6, 2006, Genova, Italy.

Copyright 2006 Springer

Incremental Model Transformation for the Evolution of Model-Driven Systems

David Hearnden[1], Michael Lawley[2], and Kerry Raymond[2]

[1] School of ITEE, University of Queensland, Australia,
hearnden@itee.uq.edu.au
[2] Queensland University of Technology, Australia,
{m.lawley,k.raymond}@qut.edu.au

Abstract. Model transformations are an integral part of model-driven development. Incremental updates are a key execution scenario for transformations in model-based systems, and are especially important for the evolution of such systems. This paper presents a strategy for the incremental maintenance of declarative, rule-based transformation executions. The strategy involves recording dependencies of the transformation execution on information from source models and from the transformation definition. Changes to the source models or the transformation itself can then be directly mapped to their effects on transformation execution, allowing changes to target models to be computed efficiently. This particular approach has many benefits. It supports changes to both source models and transformation definitions, it can be applied to incomplete transformation executions, and *a priori* knowledge of volatility can be used to further increase the efficiency of change propagation.

1 Introduction

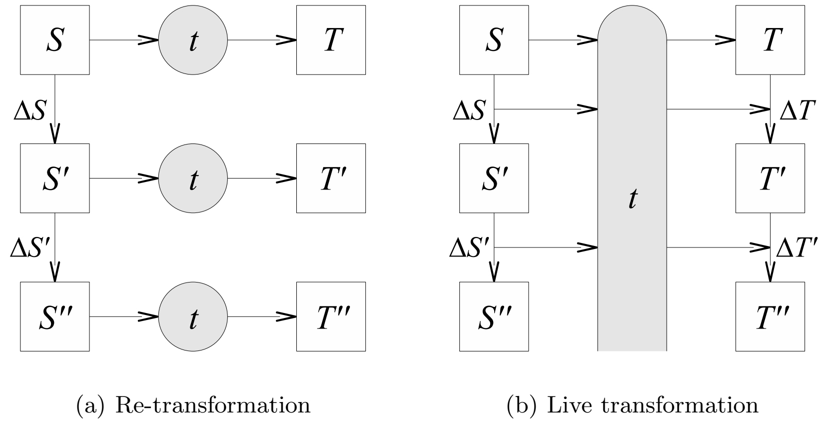
In model-driven systems, the evolution and synchronisation of source and target models often relies on the automated maintenance of transformation relationships. Large models or complex transformation specifications can cause transformation execution time to become quite significant, impeding this process. Live transformation execution is an incremental update technique designed to address these issues.

1.1 Incremental Updates

In broad terms there are two approaches to incremental updates. The first approach involves re-running the entire transformation, producing new output models that must then be *merged* with the previous output models. Updating models *in situ* is a special case of this approach, where the merge is performed implicitly. In this approach the context from the original transformation is lost, which is why a merge strategy is necessary in order to recreate that context. The feasibility of model merging for incremental transformations is heavily dependent on the traceability features of the transformation language.

The second approach involves preserving the transformation context from the original transformation, thus obviating a merge strategy to recreate it. A *live transformation* does not terminate, rather it continuously maintains a transformation context such that the effects of changes to source inputs can be readily identified, and the necessary recomputation performed.

Figure 1 illustrates these two approaches. In Figure 1(a), each successive update to S requires a complete re-transformation t producing new versions of T. If *in-situ* updates are desired, then a merge is required. In Figure 1(b), the transformation t is continuous, starting from an initial transformation from S producing T. Each successive source update ΔS is mapped directly to a target update ΔT. The transformation t does not terminate as such, but rather goes through phases of activity when S is changed.



(a) Re-transformation (b) Live transformation

Fig. 1. Incremental Update Strategies

The advantage of the second approach is that it is far more efficient, especially for small changes, and is thus more suitable for the rapid update of transformation outputs. On average, the amount of computation necessary is proportional to the size of the input changes and the output changes. This is particularly important for model-driven tools in an incremental development methodology, where models are constantly evolving and constant synchronisation is necessary for consistency. Another advantage of the second approach is that it is a more direct solution for finding the changes to outputs required in response to changes to inputs, as opposed to finding the actual outputs themselves. For a model evolution tool, this may be an important distinction. Consider the task of selecting, from a set of possible source changes under consideration, the change that produces the *smallest* consequent change on the target models.

The cost of the second approach is that the execution context must be constantly maintained. Unless there are a large number of large transformations

being maintained, this is unlikely to be a significant problem, and section 3.1 discusses how the space cost can be scalably traded for computation time should the context become too large.

1.2 Transformation Languages

We restrict our analysis to logic-based transformation languages; these languages turn out to be the most suitable for live transformation.

Of the declarative paradigms, logic languages have an advantage over functional languages because program data has a direct and clear effect on program computation. There is a single inference rule (resolution), that provides sufficient power for computational completeness. With resolution, program data has a direct influence on the evaluation process. One could say that logic languages have *data-driven* evaluation.

While functional languages are also typically classed as declarative, they are less suitable for live transformations than logic languages because the effect of program data is less clear. Reduction operations for functional evaluation are driven by the state of the expression being reduced, so the effect of program data is not direct.

There have been a variety of languages and techniques proposed in response to the MOF 2.0 Query / View / Transformation Request For Proposals [1], the majority of which have emphasised declarative definitions for transformations. The current adopted QVT specification [2] is a hybrid of declarative and imperative languages, with the declarative level being sufficiently powerful to be executable. The DSTC's submission to the QVT RFP [3] presents a transformation language that is completely declarative and can be executed with an open-source tool, Tefkat [4] [5].

The incremental update techniques presented here have been investigated in the context of Tefkat; however because of their foundational nature they should be applicable to any declarative rule-based transformation language, such as the QVT specification.

1.3 Related Work

Incremental update techniques have been extensively researched for deductive databases. The specific problem they address is the maintenance of materialised views in response to changes to base relations. The solution that has been most influential [6] involves transforming the deductive rules that define a view into *delta*-rules that define how additions and deletions to queried data could be transformed to additions and deletions to the view. There have been several variations on this theme (e.g. [7]), however as discussed in [8] they follow the same basic strategy.

The live transformation approach presented in this paper adopts a fundamentally different strategy by addressing the incremental update problem in terms of the execution context of a canonical logic engine. Instead of deriving a new

transformation to perform the incremental updates, this approach tries to isolate the effects of updates on the dynamic computation structures used for logical evaluation. This should theoretically enable more efficient update propagation as it is a more direct approach, however the price paid is that an implementation must be tightly integrated with the internal structures of a particular transformation engine rather than only being dependent on language semantics. Recent developments in incremental evaluation of tabled logic programs [9] [10] are also adopting an engine-oriented approach.

1.4 Overview of Paper

Section 2 describes SLD resolution, the theoretical basis for the evaluation of logic languages. Sections 2.3 and 2.4 respectively present the extensions required to preserve dependency information and the algorithms used to respond to changes to input models. Section 3 discusses optimisations that can be performed to further increase update efficiency, as well as how the strategy described in section 2 can be extended to allow incremental updates in response to changes to transformation definitions as well as input models. Finally, section 4 illustrates an example of live transformation execution.

2 Live Propagation

In this section we consider extending a transformation engine based on the standard mechanism for the interpretation of logic languages: SLD resolution. The evaluation of a declarative rule-based transformations is driven by a search for solutions to a *goal*. This search can be conceptualised as a tree, and this tree can be used to represent the trace of a transformation execution. As mentioned previously, because resolution is data-driven, the dependencies of program execution on input models (and also the transformation itself) have a clear manifestation, and can be recorded for later analysis. Our strategy involves recording these dependencies on source model information so that changes to the tree can be made efficiently in response to changes to source models or the transformation definition, as opposed to rebuilding the tree from scratch with a re-transformation. Changes to the search tree can then be readily mapped to consequent changes in target models.

2.1 SLD resolution

SLD resolution is a deduction rule used for the execution of logic programs. It is a restriction of the general resolution principle [11] (the S stands for *Selection*, L for *Linear*, and D for *Definite* clauses).

Given a goal G consisting of a set of atomic literals and a ruleset R consisting of a set of rules and/or facts, two choices are made. A literal a from G and a rule r from R are *selected* such that a unifies with r (there exists a variable substitution θ such that $a\theta = h\theta$, where h is the head of rule r). The atom a in

G is then replaced with the body of rule r, then the most general unifier (mgu) of a and h is applied, giving a new goal G'. The process continues until the goal is empty ($\Box$), and the composition of all the unifiers, Θ, is then a solution for the goal G. In other words, $G\Theta$ is a fact that can be deduced from the ruleset R. SLD resolution is sound and complete, so no wrong solutions are produced and all solutions can be deduced.

$$
\begin{aligned}
&f_1 : class(c_1)\\
&f_2 : class(c_2)\\
&f_3 : class(c_3)\\
&f_4 : super(c_3, c_1)\\
&f_5 : owns(c_1, p_1)\\
&f_6 : owns(c_3, p_2)\\
&\\
&r_1 : owns(C, P) \leftarrow\\
&\quad super(C, C'), owns(C', P)
\end{aligned}
$$

(a) Facts and rules.

$$
\begin{array}{ll}
class(C), owns(C, P) & \\
\Rightarrow \underline{class(C)}, owns(C, P) & [f_3, \{C \mapsto c_3\}]\\
\Rightarrow \underline{owns(c_3, P)} & [r_1, \{C \mapsto c_3\}]\\
\Rightarrow \underline{super(c_3, C')}, owns(C', P) & [f_4, \{C' \mapsto c_1\}]\\
\Rightarrow \underline{owns(c_1, P)} & [f_5, \{P \mapsto p_1\}]\\
\Rightarrow \Box &
\end{array}
$$

(b) SLD resolution (one solution).

Fig. 2. Resolving a goal against a rule set.

Consider the ruleset in Figure 2(a). Facts f_1 to f_3 describe three classes, c_1, c_2 and c_3, where c_3 is a subtype of c_1 (f_4). c_1 directly owns property p_1 and c_3 directly owns property p_2 (facts f_5 and f_6), and rule r_1 describes the transitive ownership of inherited contents, thus c_3 indirectly owns p_1 too. Figure 2(b) illustrates the resolution of a goal, $class(C), owns(C, P)$, that is a query for classes and their contents. In the first resolution, the selected literal (underlined) is $class(C)$ and the selected fact is f_3. These unify to produce a mgu $\{C \mapsto c_3\}$, and replacing $class(C)$ with the (empty) body of f_3 followed by the application of the mgu results in the new goal $owns(c_3, P)$. Three more resolutions are applied, resulting in an empty goal ($\Box$), indicating that a solution has been found. The composition of the unifiers (taking care to distinguish copies of C) results in the unifier $\{C \mapsto c_3, P \mapsto p_1\}$, representing one particular solution to the goal (class c_3 has property p_1). Note that in this example just one solution to the goal is found; there are others. By selecting different facts and rules, resolution can be used to find any solution to a goal.

2.2 SLD Trees

In SLD resolution, there are two non-deterministic choices that must be made at each resolution step: a literal must be selected, and a matching rule found. If we remove the second choice and instead resolve against *every* rule that matches the selected literal, then the resulting structure is an *SLD tree*. SLD trees represent all resolution paths, and therefore contain all solutions to a goal. The leaves of the tree are either success nodes ($\Box$) indicating a solution, or failure nodes that have non-empty goals but can not be resolved further ($\times$).

An SLD tree for the previous example is shown in Figure 3. The nodes and edges have been labelled (n_i,e_i) only for future reference. Note that SLD trees are not unique; they depend on the *selection rule* that is used to select a literal from the goal.

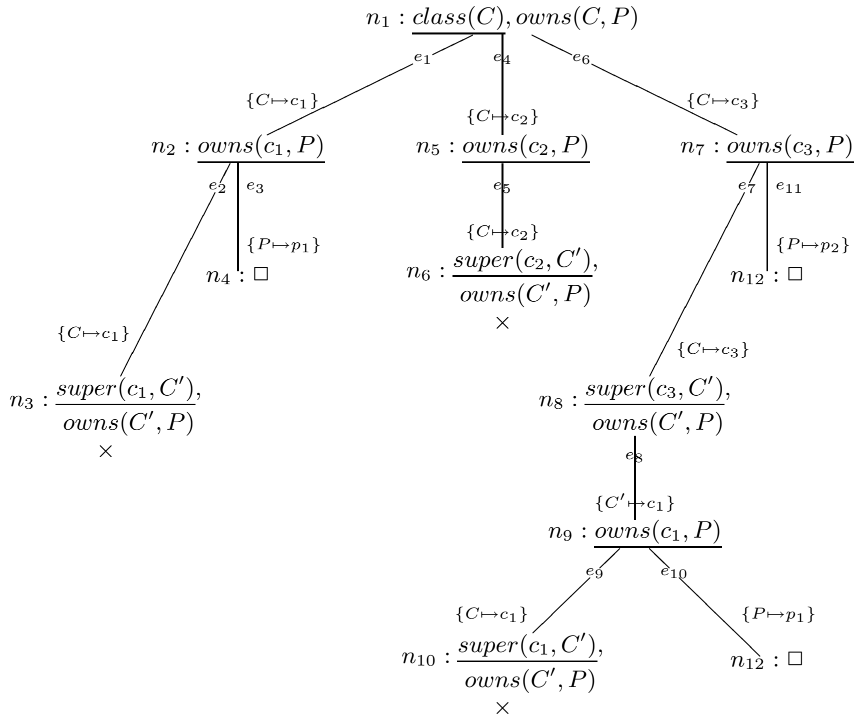


Fig. 3. An SLD tree

The SLD tree forms the basis of an execution environment for logic programs. Often the tree is not explicitly created, but rather exists implicitly via a search strategy. In Tefkat, the SLD trees are explicit. The SLD tree in Prolog, however, exists as a depth-first search.

2.3 Tagging

Our goal is to provide a live execution environment, where changes to source models can be efficiently mapped to changes to target models. From a logical perspective, source models are manifested as a set of facts, and transformations as a set of rules and facts. As those familiar with logic programming are aware, a fact is simply a special case of a rule, so there is no real need to distinguish the two; however if we are only observing changes to source models then we need only be concerned with facts.

Changes to source models after a transformation has occurred are thus manifested as changes to the fact base used by a logic engine. These facts can influence the SLD tree in precisely one way: by unifying with the selected literal of a node's goal, thus spawning an edge in the tree. Additive changes to source models can therefore cause new branches and subtrees to be computed. Deletive changes can cause branches to be pruned. In order to make these incremental changes as efficient as possible, we tag the facts with references to where they are used in the tree, so that the effects of source changes can be made directly.

Two types of information are recorded while nodes query the fact base: the usage of a fact by an edge, and the failure to find a matching fact for a node. The first requires tagging of facts, the second requires tagging of fact signatures (name and arity). For this purpose it is convenient to group facts with the same name and arity into *tables*. Algorithms Solve and Resolve illustrate how such recording can be incorporated into a resolution algorithm (lines 9-10 and 14-16).

```
Resolve(U, R)
 1  while U ≠ ∅
 2      do n ← choose(U)
 3         U ← U − {n}
 4         if goal[n] = □
 5           then solutions ← solutions ∪Solution(n)
 6           else  g ← goal[n]
 7                 l ← selectLiteral(g)
 8                 matches ← Find-Matches(R, l)
 9                 t ← getTable(l)
10                 tableTags[t] ← tableTags[t] ∪ {n}
11                 for each (θ, r) ∈ matches
12                     do e ← Create-Branch(n)
13                        unifier[e] ← θ
14                        if r is a fact
15                          then factTags[r] ← factTags[r] ∪ {e}
16                               fact[e] ← r
17                        n′ ← childNode[e]
18                        goal[n′] ← ((g − {l}) ∪ body(r))θ
19                        U ← U ∪ {n′}
20  return solutions
```

```
Solve(G, R)
1  r ← Create-Root
2  goal[r] ← G
3  solutions ← ∅
4  return Resolve({r}, R)
```

While U is non-empty, a node is chosen for expansion and removed from U (line 2). Success nodes are nodes whose goal has been reduced to $\square$ and are

Fact	Edges
$class(c_1)$	$\{e_1\}$
$class(c_2)$	$\{e_4\}$
$class(c_3)$	$\{e_6\}$
$super(c_3, c_1)$	$\{e_8\}$
$owns(c_1, p_1)$	$\{e_3, e_{10}\}$
$owns(c_3, p_2)$	$\{e_{11}\}$

(a) Fact tags

Table	Nodes
$class/1$	$\{n_1\}$
$super/2$	$\{n_3, n_6, n_8, n_{10}\}$
$owns/2$	$\{n_2, n_5, n_7, n_9\}$

(b) Table tags

Fig. 4. Dependencies

a SOLUTION algorithm (elided) is used to compute the composition of all the unifiers used from the root to the success node (line 5). Non-success nodes have a literal selected from their goal (line 7), which is then matched against the rule database to produce a set of matching rules/facts paired with the most general unifier for the match (line 8). The dependency of the node on a table of facts is then recorded (line 10).

A new branch in the tree is created for each of the matching rules/facts (line 12), and the matching rule/fact and unifier are recorded on the edge (line 13). For fact edges, the dependency of the edge on the particular fact that caused its creation is then recorded (lines 14- 16). The selected literal in the goal is replaced with the body of the matching rule/fact, the matching unifier applied, and the result is set as the new node's goal (lines 17-18). The new node is then added to the set of unexpanded nodes, to be expanded on a future iteration. After all the nodes have been expanded, RESOLVE returns the set of unifiers that represent solutions to the goal (line 20). Lines 9, 10 and 14-16 are the only extra work required for the dependency recording.

For brevity, the detail of some used algorithms has been elided. Algorithm CREATE-BRANCH(n) simply creates and returns a branch from node n in the data structure for the resolution tree. FIND-MATCHES(R, l) searches the knowledge base R for rules/facts whose heads unify with l, and returns the set of all such pairs (θ, r).

The SOLVE algorithm builds a tree from scratch by creating a root tree node, setting its goal, and calling RESOLVE. Figures 4(a) and 4(b) show the fact and table tags from the edge and node dependencies for the tree in Figure 3.

2.4 Responding to Change

We consider two types of change to the model and transformation definition: *fact addition* and *fact removal*.

Fact Addition The algorithms for responding to model or transformation change rely on the existing resolution algorithms. Informally, the response to the addition of new facts is to identify nodes in the tree for which resolution needs to be resumed. Algorithm ADD-FACT describes this procedure.

```
ADD-FACT(f)
 1  nodes ← tableTags[getTable(f)]
 2  U ← ∅
 3  for each n ∈ nodes
 4        do l ← selectedLiteral[n]
 5           θ ← UNIFY(l, head(f))
 6           if θ ≠ NIL
 7              then e ← CREATE-BRANCH(n)
 8                   unifier[e] ← θ
 9                   fact[e] ← f
10                   factTags[f] ← factTags[f] ∪ {e}
11                   n′ ← childNode[e]
12                   goal[n′] ← (goal[n] − {l})θ
13                   U ← U ∪ {n′}
14  return RESOLVE(U)
```

ADD-FACT uses the table tags to identify all the nodes with a selected literal of the same name and arity as the added fact f (line 1). The selected literals of each of these nodes are tested against the added fact, in order to find any nodes with goals that match (more formally, unify) with the head of f (line 5). Any nodes found have branches added from them, and they are added to a set of unexpanded nodes U. Note that lines 7- 13 are equivalent to lines 12- 19 from RESOLVE. Finally, resolution is resumed on all those new nodes (line 14).

ADD-FACT returns the set of unifiers from the new success nodes found in response to the addition of a fact. These unifiers represent valid solutions in the context of the new fact database, however they may not all be *new* solutions since other paths in the tree may have already established some of those solutions prior to the fact addition. Therefore the set of solutions returned by ADD-FACT must be compared with the original solutions in order to identify new solutions.

Fact Removal In response to the removal of a fact f, all the edges in the tree that were created because of a match with a selected literal must be identified. The subtrees rooted at these edges must then be removed, which involves removing all the dependency information from that subtree as well as identifying solutions that may have been removed. Similarly to ADD-FACT, REMOVE-FACT returns the set of solutions established by success nodes that have now been removed, however other success nodes in the remaining tree may also establish some of those solutions, so again they must be compared with the original solutions in order to identify invalidated solutions.

```
REMOVE-FACT(f)
1  edges ← factTags[f]
2  oldSolutions ← ∅
3  for each e ∈ edges
4        do oldSolutions ← oldSolutions ∪PRUNE-EDGE(e)
5           DELETE-BRANCH(e)
```

REMOVE-FACT is straightforward. All the edges dependent on the removed fact f are deleted from the tree, however a pruning step occurs (line 4) before the branch removal (line 5). This pruning step removes dependencies recorded for the subtree, as well as accumulating solutions from success nodes in that subtree. Mutually recursive algorithms PRUNE-EDGE and PRUNE-NODE define this procedure.

PRUNE-EDGE(e)
1 $f \leftarrow fact[e]$
2 **if** $f \neq$ NIL
3   **then** $factTags[f] \leftarrow factTags[f] - \{e\}$
4 **return** PRUNE-NODE($childNode[e]$)

PRUNE-NODE(n)
1 $oldSolutions \leftarrow \emptyset$
2 **if** $isSuccess(goal[n])$
3   **then** $oldSolutions \leftarrow oldSolutions \cup \{n\}$
4 $t \leftarrow getTable(selectedLiteral[n])$
5 $tableTags[t] \leftarrow tableTags[t] - \{n\}$
6 **for** each $e \in childEdges[n]$
7   **do** $oldSolutions \leftarrow oldSolutions \cup$PRUNE-EDGE($e$)
8 **return** $oldSolutions$

PRUNE-EDGE simply removes the edge from the potential fact dependency in which it appears, and then prunes the child node. PRUNE-NODE accumulates a solution if it encounters a success node (line 3), then removes the node from the table dependency in which it appears (line 5), and then recursively prunes its child edges, accumulating their solutions (line 7).

2.5 Negation

So far, we have only analysed SLD resolution, which does not allow *negative* literals to appear in rule bodies. In other words, rules that rely on the *absence* or the *falsity* of facts may not be used. SLD resolution can be extended to *general* clauses, which do allow negative literals in goals and rule bodies; however extra restrictions are required in order to preserve soundness and completeness.

The easiest extension to SLD resolution to allow negative literals is to use the closed-world assumption, where all unprovable facts are considered false. This allows us to treat *negation as failure*, so to prove a literal $\neg p(X)$ it is sufficient to show that there is no proof of $p(X)$. To achieve this, a separate tree is created, and if the tree *finitely fails*, then $p(X)$ is considered false and hence $\neg p(X)$ true. However if a solution is found in this separate tree, then a proof of $p(X)$ has been found, so $\neg p(X)$ is false, and hence the node that spawned the separate tree fails.

This extension is often referred to as SLDNF (SLD with Negation as Failure). SLDNF introduces a fundamental change to the structure of the resolution tree.

Instead of a single tree there is now a forest of negation trees plus one positive tree (the root tree). Nodes with a negative selected literal are 'connected' with a negation tree constructed to prove the positive literal. These connections must be maintained as part of the forest.

The algorithms from section 2.4 only apply to SLD resolution and are monotonic: Add-Fact can only add more solutions and Remove-Fact can only invalidate previous solutions. If SLDNF resolution is used instead, then monotonicity is lost, and incremental updates become more complex. The addition of facts may result in the removal of branches (and hence the removal of solutions), and the removal of facts may result in the addition of branches (and hence the addition of solutions). It turns out that the algorithms Add-Fact and Remove-Fact require only minor modifications in order to achieve this behaviour. The update phase then iterates between tree pruning and tree expansion until a fixed point is reached.

3 Discussion

In this section we discuss two ways to further optimise incremental updates, and how the techniques from section 2.3 can be extended to also allow incremental changes to transformation definitions.

3.1 Incomplete transformation context

The price for the efficiency of live transformation is the maintenance of the transformation context (the SLDNF trees) and the dependency tables. Previously it was assumed that the context was complete, i.e. the SLDNF trees and dependency tables were completely preserved. This complete context may be costly for large and complex transformations where there may be hundreds of thousands of tree nodes, and hundreds or even thousands of facts and rules.

The live transformation strategy can accommodate an incomplete context with some extensions to the algorithms presented in section 2.4. Arbitrary subtrees can be collapsed into a single 'collapsed' node, with all the dependency information condensed on that node. The space of that subtree is then reclaimed, but the aggregated tags preserve the dependency information. There is a computational cost only if the dependency information identifies that the collapsed node has been potentially affected, and then the entire subtree must be recomputed. However because collapsing can be performed at any point in a tree, it is quite a scalable trade-off.

The trick to making effective choices for node collapsing is to recognise that some facts in a model are more stable than others. For example, a person's name is less likely to change than their height or weight. We use the term *volatility* to describes the likelihood of change for a fact or rule. It is obviously most beneficial to collapse subtrees that are non-volatile. With good estimates of fact volatility (either explicitly provided or obtained via heuristics), an intelligent engine can reduce the size of the transformation context while still providing the efficiency for most incremental changes.

3.2 Ordering of volatile literals

The volatility of different facts and rules can be leveraged in an even more fundamental way. The structure of the resolution trees is completely determined by the *selection rule* that chooses which literal in a node's goal is to be resolved. This structure has a significant impact on the efficiency of the initial transformation and also the efficiency of the incremental updates. If volatile facts are used towards the root of a resolution tree, then changes to those facts involve pruning the entire subtree rooted at the usage of those facts and subsequently regrowing the new subtree. If volatile facts are used towards the leaves of a resolution tree, then the impact of changes to those facts is much less, as the subtrees that are pruned and regrown are smaller.

By providing an engine with such volatility estimates, perhaps user specified or even collected from version histories, the selection rule can choose to expand stable literals first, and volatile literals last, reducing the cost of updates to those volatile facts.

3.3 Rule/Fact equivalence

As mentioned in section 2.3, as far as logic is concerned facts are simply a special type of rule. There is very little in the algorithms of section 2.4 that applies to facts but not rules, and so with some very minor modifications live transformation can be used for rules as well. This is of great importance for the evolution of transformation definitions, since changes to the rules in a transformation can be efficiently propagated to updates on the transformation targets.

4 Live Transformation In Practice

In this section we present some preliminary measurements of the efficiency of incremental model transformation using live resolution trees.

4.1 Sample Transformation

The sample transformation we use to demonstrate live resolution trees is a simplified version of one of the many transformations from an object-oriented class metamodel (such as UML) to a relational database schema metamodel. The complete metamodels have been omitted due to space considerations.

The class metamodel describes classes that own properties which are attributes or references, where attributes are data-valued and references are object-valued. Classes have zero or more superclasses. The relational schema metamodel describes tables that own typed columns, one of which is designated as a primary key.

The transformation maps classes to tables with keys. Properties that are owned directly or indirectly (through inheritance) are mapped to columns of the table corresponding to the property's owning class. For attributes, those columns

are typed by the data type corresponding to the attribute's data type. For references, those columns are typed by the data type corresponding to the type of the primary key of the table mapped from the type of the reference. In other words, references are mapped to foreign key columns. Finally, if a class has an attribute marked as an identity attribute, the column mapped from that class-attribute pair becomes the key column for the class's corresponding table. Otherwise, a primary key column called `ID` is inserted. This sample transformation is useful as it is small enough to be easily understood, but complex enough to involve recursion, transitive closure and negation. The main rules for this transformation are described in Tefkat's concrete syntax in Figure 5.

The output of a Tefkat transformation is the unique minimal model (least fixed point) such that all rules are true. A rule is true if and only if, for all variable bindings for its source terms (`FORALL`, `LINKS`, `WHERE`), the target terms (`MAKE`, `LINKING`) are true. *Patterns*, such as `hasProperty/2`, are equivalent to logical predicates. Tefkat also uses *trackings*, which are essentially named relations, and are the only elements that may be both queried/checked (with `LINKS`) and asserted/enforced (with `LINKING`). For example, the `ClassTable` tracking associates a class with a table, and is asserted in the `LINKING` clause of `ClassToTable`, and is queried by the `LINKS` clauses of the other four rules.

4.2 Sample Execution

The transformation was run on the Ecore metamodel [12], followed by three updates. The first update was a simple renaming of an attribute of `ETypedElement`. The propagated changes involved the renaming of 6 columns, one from each of the tables generated for the 6 subclasses that inherited that attribute. The second update involved the deletion of the `ETypedElement.type` reference. The 6 affected columns were deleted, as were all of their properties. The final change was marking `EClassifier.instanceClassName` as an identity attribute. which caused the most significant structural change. The automatic key columns added to the tables for `EClassifier` and its subclasses were deleted, those tables' keys were set to the columns for the `instanceClassName` attribute, and all columns generated from `EReference`s to `EClassifier` or any of its subclasses (i.e. foreign keys into the `EClassifier` table) had their types changed to `String`, the new type of `EClassifier`'s key column.

Table 1 shows the number of resolution nodes added, removed, and touched in each of the three updates. Node addition and removal are the most significant measurements since those operations involve modifications to the resolution trees and tag structures. Touched nodes are those nodes that were identified by the fact and table tags as being potentially affected, but on closer inspection were not affected.

The number of nodes is correlated with the execution time and space consumption of the incremental transformation. The results clearly show significant performance benefits from live transformation for all three updates.

```
RULE ClassToTable
FORALL  Class c { name: n; }
MAKE    Table t { name: n; }
LINKING ClassTable WITH class = c, table = t;

PATTERN hasProperty(c, p)
WHERE   c.properties = p OR hasProperty(c.super, p);

RULE AttributeTypes
FORALL  Class c, Attribute a { name: n; }
WHERE   hasProperty(c, a)
AND     TypeType   LINKS ooType = a.type, rdbType = rdbtype
AND     ClassTable LINKS class = c, table = t
MAKE    Column col { name : n; table: t; type: rdbtype; }
LINKING AttributeColumn WITH class = c, attribute = a,
                             column = col, type = rdbtype;

RULE ForeignKeyTypes
FORALL  Class c, Reference r { name: n; type: rc; }
WHERE   hasProperty(c, r)
AND     ClassTable  LINKS class = rc, table = ft
AND     TableKey    LINKS table = ft, key = _, type = fktype
AND     ClassTable  LINKS class = c, table = t
MAKE    Column col { name : n; table: t; type: fktype; };

RULE IdKeyColumn
FORALL  Class c, Attribute a
WHERE   hasProperty(c, a) AND a.id = true
AND     AttributeColumn LINKS class = c, attribute = a,
                              column = col, type = keytype
AND     ClassTable      LINKS class = c, table = t
MAKE    Key k { table: t; column: col; }
LINKING TableKey WITH key = k, table = t, type = keytype;

RULE AutoKeyColumn
FORALL  Class c
WHERE   NOT (hasProperty(c, a) AND a.id = true)
AND     ClassTable LINKS class = c, table = t
MAKE    makeRdbType("Auto", auto),
        Column col { name: "ID"; type: auto; table: t;},
        Key k { table: t; column: col; }
LINKING TableKey WITH key = k, table = t, type = auto;
```

Fig. 5. Sample OO to RDB transformation.

Table 1. Number of tree nodes used during live transformation.

#	Forest Size	Added /Removed	Touched	Total	Changed
-	7026	- / -	-	7026 (100%)	100%
1	7026	78 / 78	176	332 (4.7%)	2.2%
2	6828	0 / 198	0	198 (2.9%)	2.9%
3	6760	198 / 206	349	753 (11.1%)	6.0%

5 Conclusion

Incremental updates for declarative rule-based model transformations can be performed efficiently using live transformations. The dependencies of the transformation execution on its inputs can be recorded by tagging resolution trees. These dependencies can then be used to efficiently propagate source changes to target changes. With minor extensions, the algorithms presented in this paper can be used in the presence of negation and tabling, and also for incremental updates to the transformation definitions. Finally, awareness of model volatility can be leveraged to further increase update efficiency.

References

1. OMG: MOF 2.0 Query / Views / Transformations RFP. OMG document ad/02-04-10 (2002)
2. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG document ptc/2005-11-01 (2005)
3. DSTC, IBM, CBOP: MOF Query / View / Transformation Second revised submission. OMG document ad/2004-01-06 (2004)
4. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of Lecture Notes in Computer Science., Springer (2005) 139–150
5. : Tefkat: The EMF transformation engine (2006)
6. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In Buneman, P., Jajodia, S., eds.: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993, ACM Press (1993) 157–166
7. Ceri, S., Widom, J.: Deriving incremental production rules for deductive data. Information Systems **19**(6) (1994) 467–490
8. Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques and applications. IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing **18**(2) (1995) 3–18
9. Saha, D., Ramakrishnan, C.R.: Symbolic support graph: A space efficient data structure for incremental tabled evaluation. In: ICLP. (2005) 235–249
10. Saha, D., Ramakrishnan, C.R.: Incremental evaluation of tabled prolog: Beyond pure logic programs. In: PADL. (2006) 215–229
11. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1) (1965) 23–41
12. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)