Transformational Software Evolution by Assertions

Dr. Tom Mens*
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - Belgium
Tom.Mens@vub.ac.be

ABSTRACT

This paper explores the use of software transformations as a formal foundation for software evolution. More precisely, we express software transformations in terms of assertions (preconditions, postconditions and invariants) on top of the formalism of graph rewriting. This allows us to tackle scalability issues in a straightforward way. Useful applications include: detecting syntactic merge conflicts, removing redundancy in a transformation sequence, factoring out common subsequences, etc.

1. INTRODUCTION

Software evolution is one of the most important problems in software engineering, because of its inherent complexity and because of the lack of a solid formal foundation. In an attempt to provide such a foundation, this paper elaborates on the paradigm of *transformational software evolution*. In this paradigm, evolution is achieved by means of explicit software transformations that can be manipulated directly. This gives rise to a wide range of interesting ways to improve support for evolution.

One area of interest lies in support for merging parallel evolutions of the same software [3, 9]. Software merging is needed when separate lines of software development are carried out in parallel and have to be merged at regular intervals. Because this is a complex time-consuming process, automated support tools are essential. Unfortunately, most existing merge tools either lack flexibility or expressive power. To counter this problem, we need to establish the formal foundations of software merging first. For this purpose, graph rewriting appears to be a promising lightweight formalism [11].

Software transformations are also useful to provide support for refactoring application frameworks in a behaviour-preserving way. Refactorings improve the design or structure of object-oriented frameworks, making them more robust towards evolution [13, 14, 16].

For merging as well as refactoring, there is a need to express evolution transformations in a scalable way. Indeed, in practice, the software that is being developed as well as the software transformations that are applied to it can be quite large.

A promising formal approach which has not yet been thoroughly explored is the use of *assertions* for expressing software transformations. In [16], pre- and postconditions were used to express refactoring transformations. In [11], pre- and postconditions were attached to software transformations to detect merge conflicts. This paper performs a more thorough investigation, and shows how assertions allow us to express software transformations in a uniform and scalable way.

2. CONDITIONAL GRAPH REWRITING

We represent software artifacts (whether it be analysis, architecture, design or implementation artifacts) in a uniform way as graphs [10]. This enables us to use the powerful formalism of *conditional graph rewriting* [4, 5, 6, 11] for representing evolution transformations.

2.1 Graphs and Graph Rewriting

Graphs provide a simple yet expressive formalism for representing software. *Nodes* in a graph can represent any kind of software entity (classes, modules, objects, methods, variables, statements, etc...), while *edges* express dependencies between these entities (data-flow, control-flow, containment relationships, etc...). Each node and edge has a *label* and a *type* attached to it.

Definition. Let *NodeID* be the set of node identifiers, *EdgeID* the set of edge identifiers, *Label* the set of node and edge labels, and *Type* the set of node and edge types. A **graph** G is a tuple $(V, E, source, target, label, type)$ consisting of a node set $V \subseteq NodeID$ and an edge set $E \subseteq EdgeID$ with $V \cap E = \emptyset$; functions $source: E \rightarrow V$ and $target: E \rightarrow V$; and functions $label: V \cup E \rightarrow Label$ and $type: V \cup E \rightarrow Type$.

For example, in graph R depicted in Figure 1, $V=\{a,c\}$, $E=\{f\}$, $label(a)$=area, $type(a)$=***operation***, $label(f)$=*uses*, $type(f)$=***uses***, $source(f)=a$ and $target(f)=c$. We distinguish types from labels by writing types in boldface.

Since graphs represent software artifacts, evolution of these artifacts can be expressed by *graph rewriting*. Because we will manipulate graph rewritings explicitly, they should be decoupled from the actual graphs to which they are being applied. This is achieved by introducing the notion of a ***graph production*** $P: L \rightarrow R$ that transforms a source graph L into a target graph R. In order to apply this production to an initial graph G, a match $m: L \rightarrow G$ is needed to specify which part of the initial graph G is

---

* Postdoctoral Fellow of the Fund for Scientific Research – Flanders (Belgium) (F.W.O.-Vlaanderen)

[3] Primitive productions *Relabel* and *Retype* can be used for nodes as well as edges. We often use the notation *RelabelN* and *RetypeN* (resp. *RelabelE* and *RetypeE*) to stress that we are changing the label or type of a node (resp. edge).

being transformed. Together, *P* and *m* uniquely define a ***graph rewriting*** $G \Rightarrow_{P,m} H$. This graph rewriting also induces a co-match *m*: R→H* that specifies the embedding of *R* in the result graph *H*.

As an example, consider the graph rewriting of Figure 1. The match *m: L→G* maps node *a* of *L* on node *2* of *G*. The co-match *m*: R→H* additionally maps node *c* of *R* on node *3* of *H*, and edge *f* of *R* on edge *f* of *H*.

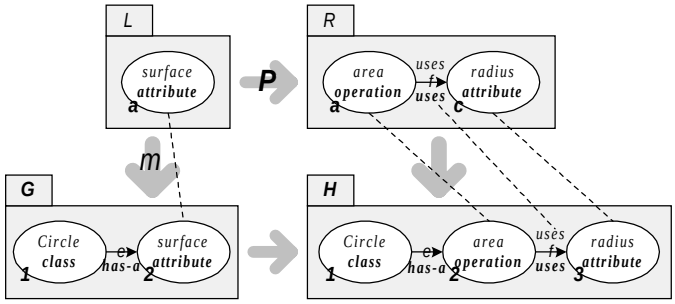

Figure 1: An example of a graph rewriting

2.2 Assertions

Assertions are well established in the software community as a formal way to specify the behaviour of programs [7, 12]. Three kinds of assertions are distinguised. *Preconditions* must be satisfied for a certain operation to be applicable. *Postconditions* are guaranteed to be true after the operation has been applied. *Invariants* are assumptions that remain unaltered by the operation.

Another distinction is made between *positive* assertions, that indicate the presence of a certain property, and *negative* assertions that indicate its absence. Table 1 presents the positive assertions that can be expressed in our graph formalism, together with the notation used throughout this paper. Negative assertions are precisely the opposite: they express the absence of some entity in a graph, and are denoted by a minus sign. E.g., *–source(E,N)* expresses that edge *E* does not have node *N* as its source.

Table 1: Positive assertions

Positive assertion	Notation
A node or edge with identifier *Id* should be present	*+Id*
Edge *E* should have node *N* as its source	*+source(E,N)*
Edge *E* should have node *N* as its target	*+target(E,N)*
A node or edge *Id* should have label *L*	*+label(Id,L)*
A node or edge *Id* should have type ***T***	*+type(Id,**T**)*

We also want to express more general constraints like: "node *N* does *not* have *any* outgoing edges" or "node *N* is the target of *at least one* edge". The former constraint is expressed as *-source(*,N)*, and the latter as *+target(*,N)*. All positive wildcard assertions used in this paper are enumerated in Table 2. Negative wildcard assertions are merely the negation of their positive equivalents. For example, *-source(*,N)* is the negation of $\exists E \in EdgeID: source(E) = N$, i.e., $\forall E \in EdgeID: source(E) \neq N$

Table 2: Positive wildcard assertions

Positive assertion	Notation
$\exists E \in EdgeID: source(E) = N$	*+source(*,N)*
$\exists E \in EdgeID: target(E) = N$	*+target(*,N)*
$\exists N \in NodeID: source(E) = N$	*+source(E,*)*
$\exists N \in NodeID: target(E) = N$	*+target(E,*)*
$\exists L \in Label: label(Id) = L$	*+label(Id,*)*
$\exists \mathbf{T} \in Type: type(Id) = \mathbf{T}$	*+type(Id,*****)*

Some assertions automatically imply other assertions. For example, the absence of a node implies the absence of any label or type for this node, as well as the absence of any incoming or outgoing edges for this node. These implicit assertions are called *derived assertions* and are mentioned in Table 3. Whenever we specify a set of assertions *S*, we assume that *all derived assertions are also included in this set, even if they are not specified explicitly*.

Table 3: Derived assertions

Assertion	Derived Assertions
-N	*-label(N,*), -type(N,*****), -source(*,N), -target(*,N)*
-E	*-label(E,*), -type(E,*****), -source(E,*), -target(E,*)*
+source(E,N)	*+E, +N*
+target(E,N)	*+E, +N*
+label(Id,L)	*+Id*
*+type(Id,**T**)*	*+Id*

2.3 Conditional Graph Productions

The main distinction between our approach and the "common" use of assertions [7, 12, 15] is that we do not use assertions to attach behavioural constraints to programs. Instead, we use assertions to represent evolution transformations (as in [11, 16]). In other words, we attach assertions to graph productions rather than to graphs themselves.

Each assertion can be used either as precondition, postcondition or invariant of a graph production *P*. The sets of all these assertions are denoted by ***Pre(P)***, ***Post(P)*** and ***Inv(P)*** respectively. We also use the shorthand notations ***Before(P)*** = $Pre(P) \cup Inv(P)$ and ***After(P)*** = $Post(P) \cup Inv(P)$.

Given a graph rewriting $G \Rightarrow_{P,m} H$, one can easily write an algorithm that calculates the minimal set of assertions that determines the production *P*. For example, in Figure 1 we can identify the following minimal assertions:

*Pre(P) = {-c, -f, +label(a,surface), +type(a,**attribute**)}*

Inv(P) = {+a, -source(,c)}*

*Post(P) = {+label(a,area), +type(a,**operation**), +c, +label(c,radius), +type(c,**attribute**), +f, +source(f,a), +target(f,c), +label(f,uses), +type(f,**uses**)}*

If necessary, extra assertions can be added to these sets in order to restrict the applicability of production *P* to a smaller set of initial graphs. For example, if we would impose the extra invariant -

target(,a)*, *P* would not be applicable anymore to the graph *G* of Figure 1.

Following the notation of Perry [15], the assertions for production *P* are depicted as ellipses in Figure 2, while *P* is represented as a grey rectangle. Preconditions appear on the upper horizontal side of the rectangle, postconditions on the lower horizontal side, and invariants on the vertical sides. For positive assertions, the + sign is omitted in the figures. When they are needed, derived assertions are depicted by dashed ellipses. Finally, we abbreviated the last five postconditions of *P* to *(f,a,c,uses,**uses**)*.

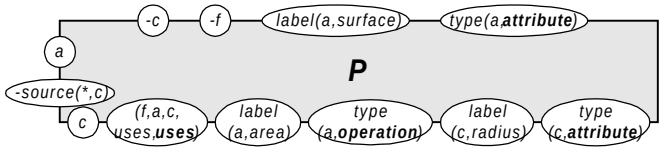

Figure 2: Graphical notation of a conditional production

[11] expressed every possible graph transformation in terms of a number of primitive productions that are sufficient to express any kind of change to a graph. For example, *AddEdge(f,a,c,uses,**uses**)* adds an edge *f* from *a* to *c* with label *uses* and type ***uses***. Table 5 shows all primitive productions and their corresponding assertions.[3]

Table 5: Primitive graph productions

Graph Production	Pre	Inv	Post
AddNode (N,L,T)	*-N*	*-source(*,N)* *-target(*,N)*	*+N* *+label(N,L)* *+type(N,**T**)*
AddEdge (E,N_s,N_t,L,T)	*-E*	*+N_s* *+N_t*	*+E* *+label(E,L)* *+type(E,**T**)* *+source(E,N_s)* *+target(E,N_t)*
DropNode (N)	*+N*	*-source(*,N)* *-target(*,N)*	*-N*
DropEdge (E,N_s,N_t)	*+E* *+source(E,N_s)* *+target(E,N_t)*	*+N_s* *+N_t*	*-E*
Relabel (Id,L_1,L_2)	*+label(Id,L_1)*	*+Id*	*+label(Id,L_2)*
Retype (Id,$\mathbf{T_1}$,$\mathbf{T_2}$)	*+type(Id,$\mathbf{T_1}$)*	*+Id*	*+type(Id,$\mathbf{T_2}$)*

3. PRODUCTION SEQUENCES

3.1 Well-formedness

A *production sequence* is a sequence of graph productions that can be applied successively. It is well-formed if the assertions imposed by a production in the sequence do not contradict assertions imposed by earlier productions.

Definition. A production sequence $P_1; P_2; ..; P_n$ is **well-formed** if $\forall A_k \in Before(P_k)$ with $k \in \{2..n\}$: if ($\exists A_i \in After(P_i)$ with $i<k$ such that A_i contradicts A_k) then ($\exists A_j \in After(P_j)$ with $i<j<k$ such that $A_j = A_k$). Otherwise, the production sequence is **ill-formed**.

Table 6 mentions all possible *contradicting assertions.* For example, the sequence $P_1; P_2$ = *AddNode(a,surface,**attribute**);* *AddNode(a,area,**attribute**)* is ill-formed because $+a \in After(P_1)$ contradicts $-a \in Before(P_2)$. The sequence $P_1; P_2; P_3$ = *AddNode(a,l_1,t_1); RelabelN(a,l_1,l_2); RelabelN(a,l_2,l_3)* is well-formed because the contradiction between $+label(a,l_1) \in After(P_1)$ and $+label(a,l_2) \in Before(P_3)$ is absorbed by $+label(a,l_2) \in After(P_2)$.

Table 6: Contradicting assertions

Assertion	Contradicts	where
+A	*-A*	*+A is some arbitrary positive assertion*
+source(E,N_1)	*+source(E,N_2)*	$N_1 \neq N_2$
+target(E,N_1)	*+target(E,N_2)*	$N_1 \neq N_2$
+label(Id,L_1)	*+label(Id,L_2)*	$L_1 \neq L_2$
+type(Id,T_1)	*+type(Id,T_2)*	$T_1 \neq T_2$

3.2 Detecting Syntactic Merge Conflicts

Ill-formed production sequences can be used to detect syntactic merge conflicts. These typically occur when different software developers are making changes to the same software in parallel, and these changes need to be merged.

Using the formalism of conditional graph rewriting, software merging can be formalised [11] by the notion of *parallel independence* [5]. Intuitively, two graph rewritings are parallel independent if they can be sequentialised in any order without changing the end result. Unfortunately, this definition does not specify what to do when two graph rewritings *cannot* be merged (read: sequentialised). If this is the case, we say that they give rise to a *syntactic conflict*. For example, suppose that graph *G* contains a node, and production P_1 removes this node while P_2 independently adds an edge originating from this node. This yields a syntactic conflict since trying to merge both parallel evolutions would lead to an edge without a source.

Definition. Two graph rewritings $G \Rightarrow_{P1,m1} H_1$ and $G \Rightarrow_{P2,m2} H_2$ lead to a **syntactic conflict** if the production sequence $P_1; P_2$ (or $P_2; P_1$) is ill-formed.

By comparing the different kinds of assertions that hold for P_1 and P_2, we can easily determine when a syntactic conflict occurs. It suffices to find a contradicting assertion between $After(P_1)$ and $Before(P_2)$, using Table 6. For example, for the primitive productions of Table 5 we identify the following syntactic conflicts:

- **Prohibited node removal** if $-v \in After(P_1)$ and $+v \in Before(P_2)$. This is for example the case if P_1 = *DropNode(v)* and P_2 = *AddEdge(e,v,w,l,t).* One cannot add an edge with a certain source node if this node has been removed before. **Prohibited edge removal** is defined similarly.
- **Dangling source** if $+source(e,v) \in After(P_1)$ and $-source(e,v) \in Before(P_2)$. This is for example the case if P_1 = *AddEdge(e,v,w,l,t)* and P_2 = *DropNode(v).* One cannot remove a node that still has outgoing edges. **Dangling target** is defined similarly.
- **Prohibited node introduction** if $-v \in Before(P_2)$ and $+v \in After(P_1)$. **Prohibited edge introduction** is defined similarly.

- **Prohibited relabeling** if $+label(id,l_1) \in After(P_1)$ and $+label(id,l_2) \in Before(P_2)$. **Prohibited retyping** is defined similarly.

For approaches that can detect *semantic conflicts* rather than syntactic conflicts, we refer to [1, 2, 8].

3.3 Dependencies

Between the productions in a sequence we can determine *dependencies* based on which assertions are satisfied by assertions of productions earlier in the sequence. These dependencies will be used to address scalability issues in section 4.

Definition. Let $P_1; P_2; ..; P_n$ be a well-formed production sequence and $i<j$. An assertion $A_j \in Before(P_j)$ **is satisfied by** an assertion $A_i \in After(P_i)$ if $A_j = A_i$.

We can distinguish four satisfaction dependencies:

- $A_i \in Post(P_i)$ and $A_j \in Pre(P_j)$: P_j modifies (or removes) an entity that was already modified (or introduced) by P_i. For example, P_j = *DropEdge(e,b,c)* depends on P_i = *AddEdge(e,b,c,uses,**uses**)* because P_j removes the edge e that was introduced by P_i. This is detected by $+e \in Post(P_i) \cap Pre(P_j)$
- $A_i \in Post(P_i)$ and $A_j \in Inv(P_j)$: P_j relies on an entity that is modified by P_i. For example, P_j = *AddEdge(e,b,c,uses,**uses**)* depends on P_i = *AddNode(c,radius,**attribute**)* because $+c \in Post(P_i) \cap Inv(P_j)$
- $A_i \in Inv(P_i)$ and $A_j \in Pre(P_j)$: P_j modifies an entity that was relied on by P_i. For example, P_j = *DropNode(b)* depends on P_i = *DropEdge(e,b,c)*
- $A_i \in Inv(P_i)$ and $A_j \in Inv(P_j)$: P_j relies on the same entity as P_i. For example, P_j = *RetypeN(a,**attribute**,**operation**)* depends on P_i = *RelabelN(a,surface,area)*

The first three satisfaction dependencies are **strong dependencies** because changing the order of P_i and P_j yields an ill-formed production sequence. For example, we cannot add an edge between two nodes if one of these nodes is not yet present. Graphically, strong dependencies are represented by a solid line from A_j to A_i.

The fourth dependency is a **weak dependency**, because P_i and P_j can still be commuted without affecting the end result. For example, it is irrelevant whether we first relabel a node and then retype it or vice versa. Weak dependencies are represented by a dotted line from A_j to A_i.

Figure 4 shows all satisfaction dependencies in a sequence of three primitive productions. There is a strong dependency from the invariant $+b$ of the second production to the postcondition $+b$ of the first production, and from the precondition *type(b,attribute)* of the second production to the postcondition *type(b,attribute)* of the first. Finally, there is a weak dependency from the invariant $+b$ of the third production to the same invariant of the second production.

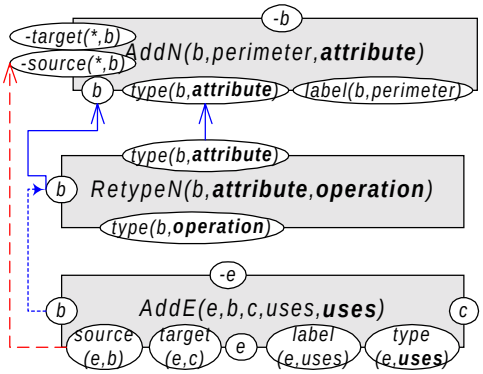

Figure 4: An illustration of satisfaction dependencies

Figure 4 also shows another kind of dependency from the postcondition *+source(e,b)* of the last production to the invariant *-source(*,b)* of the first. In general, some assertions of earlier productions can become captured by a postcondition of a later production, meaning that the earlier assertion can be ignored.

Definition. Let $P_1; P_2; ..; P_n$ be a well-formed production sequence and $i<j$. An assertion $A_j \in Post(P_j)$ **captures** an assertion $A_i \in After(P_i)$ if A_j contradicts A_i.

A capture is also a **strong dependency** in the sense that it prevents P_i and P_j from being commuted. Graphically, such a dependency is represented by a dashed line from postcondition A_j to postcondition (or invariant) A_i. This is illustrated in Figure 4 between *+source(e,b)* and *-source(*,b)*.

The following complex production sequence illustrates all the dependencies introduced before:

*RelabelN(a,surface,area); AddNode(b,perimeter,**attribute**); RetypeN(a,**attribute**,**operation**); RetypeN(b,**attribute**,**operation**); AddNode(c,radius,**attribute**); AddEdge(e,b,c,uses,**uses**); AddEdge(f,a,c,uses,**uses**); DropEdge(e,b,c); DropNode(b)*

Figure 7 displays the assertions of each production in the sequence, together with all dependencies between them. Each assertion is the source of at most one dependency, that always points to the closest preceding assertion on which it depends.

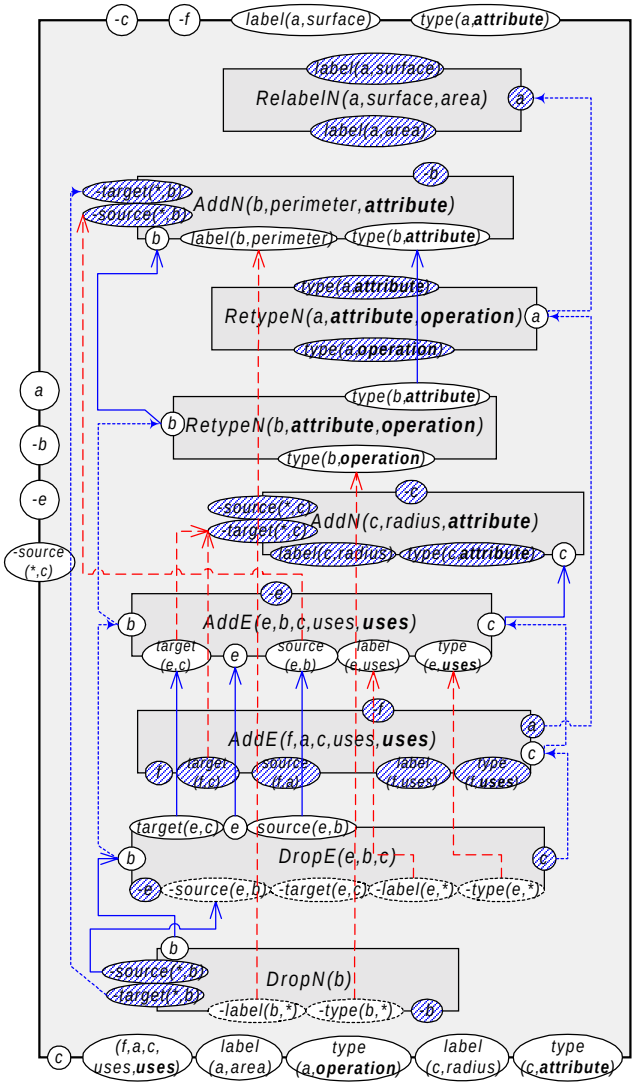

Figure 7: Dependencies in a production sequence

4. COMBINING GRAPH PRODUCTIONS

This section illustrates some important ways in which dependencies between assertions can address scalability issues when using large evolution sequences.

4.1 Composite Graph Production

A first way to address scalability is by treating complex sequences in exactly the same way as primitive productions. For example, the production sequence of Figure 7 can be considered as an atomic production *P*, as long as we are able to determine all of its assertions from the assertions of its constituent productions and the dependencies between them. The assertions of the so-called **composite production** *P* are calculated as follows:

(1) Identify all preconditions *Pre* and invariants *InvPre* that have no *outgoing* dependencies. Omit all derived assertions.

(2) Identify all postconditions *Post* and invariants *InvPost* that have no *incoming* dependencies. Omit all derived assertions.

(3) Calculate the assertions of the composite production *P*:

$Inv(P) = (InvPre \cap InvPost) \cup (Pre \cap Post)$

$Pre(P) = (InvPre \setminus InvPost) \cup (Pre \setminus Post)$

$Post(P) = (InvPost \setminus InvPre) \cup (Post \setminus Pre)$

In Figure 7, all the assertions in the sets *Pre*, *InvPre*, *Post* and *InvPost* of steps (1) and (2) are represented as shaded ellipses.

The actual preconditions, postconditions and invariants of the composite production *P* are shown as ellipses on the surrounding rectangle of Figure 7. For example, *Pre(P) = {-target(*,c)} ∪ {-c, -f, label(a,surface), type(a,**attribute**)}*, but the assertion *-target(*,c)* is omitted since it can be derived from *-c*.

4.2 Simplifying pairs of productions

Another way to address the scalability is by reducing a production sequence $P_1; P_2; ...; P_n$ by simplifying or eliminating pairs of successive[5] productions $P_i; P_{i+1}$. This is particularly relevant if we rely on a predefined set of productions (as in Table 5). Two kinds of simplifications can be distinguished. A pair of successive productions can be *absorbed* into a single predefined production, or the pair is *redundant* when the constituent productions cancel each other's effect. In the latter case, the pair can be removed without changing the overall behaviour of the graph rewriting. For both situations, a definition and concrete example is presented below.

Definition. A sequence of two graph productions $P_1; P_2$ is **absorbing** if there is a predefined graph production *P* such that $Pre(P) = Pre(P_1; P_2)$, $Post(P) = Post(P_1; P_2)$, and $Inv(P) = Inv(P_1; P_2)$

Figure 8 illustrates an absorbing production pair. Node addition *AddNode(b,perimeter,**attribute**)* followed by node retyping *RetypeN(b,**attribute**,**operation**)* is absorbed into a single node addition *AddNode(b,perimeter,**operation**)*.

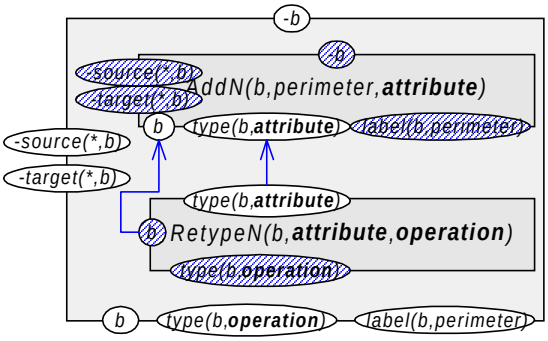

Figure 8: An absorbing production pair

Definition. A sequence of two graph productions $P_1; P_2$ is **redundant** if $Pre(P_1; P_2) = \varnothing$ and $Post(P_1; P_2) = \varnothing$.

With redundant pairs of productions, only the invariant set can be nonempty. Figure 9 illustrates a redundant production pair $P_1; P_2$. A node *b* is added and removed again. The resulting composite

[5] In section 4.4 we discuss the more complex case where redundant or absorbing productions do not directly follow one another in the sequence.

production has an empty set of pre- and postconditions, while *Inv(P_1; P_2) = {-b}*.[6] Also note the capture dependencies originating from *-type(b,*)* and *-label(b,*)*.

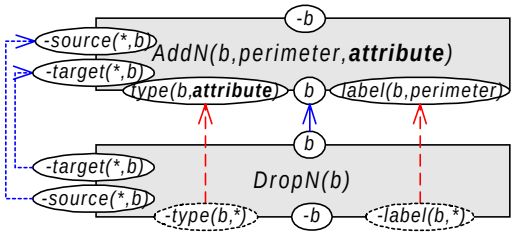

Figure 9: A redundant production pair

4.3 Reordering

If two successive productions in a sequence do not have a strong dependency between them, their order can be changed. When doing this, we need to modify all involved dependencies accordingly. This is illustrated in Figure 11 where we changed the order of the last two productions in the sequence of Figure 4. This was possible because there is only a weak dependency between the two productions that are being commuted. The reordered production sequence has the same overall effect as the original one because the assertions of the corresponding composite production are identical in both cases.

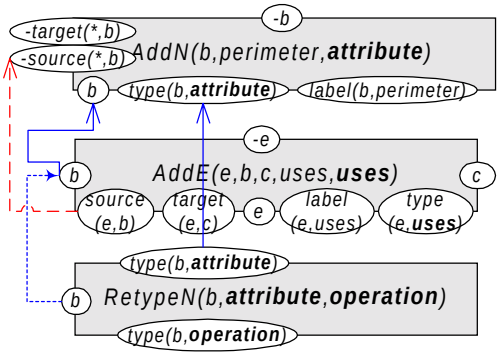

Figure 11: Reordering primitive productions in the sequence of Figure 4

4.4 Removing Redundancy

Reordering can be used to remove redundant and absorbing production pairs in a given sequence, even if the involved productions do not directly follow one another. In this way we can make the production sequence shorter, thus reducing the amount of memory required to store a production sequence (compression); improving the efficiency of algorithms that manipulate production sequences; making the production sequence easier to understand; etc…

Instead of giving the details of the redundancy removal algorithm, we illustrate how it works by means of a nontrivial example. Removing redundancy in the production sequence of Figure 7 yields the production sequence of Figure 12, containing only 4 instead of the original 9 primitive productions:

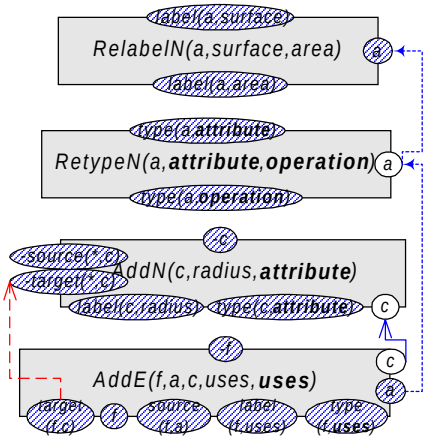

Figure 12: Final result after redundancy removal

This result is achieved by applying the following steps, starting from the production sequence of Figure 7:

1. Reorder of *RetypeN(a,**attribute**,**operation**)* and its immediate successor *RetypeN(b,**attribute**,**operation**)*, making *RetypeN(b,**attribute**,**operation**)* the immediate successor of *AddNode(b,perimeter,**attribute**)*.

2. Transform the absorbing subsequence *AddNode(b,perimeter,**attribute**); RetypeN(b,**attribute**,**operation**)* into a single production *AddNode(b,perimeter,**operation**)*.

3. Reorder of *AddEdge(f,a,c,uses,**uses**)* and its immediate successor *DropEdge(e,b,c)*, making *DropEdge(e,b,c)* the immediate successor of *AddEdge(e,b,c,uses,**uses**)*.

4. Transform the redundant subsequence *AddEdge(e,b,c,uses,**uses**); DropEdge(e,b,c)* into a single trivial production that only consists of invariants: *{-e,+b,+c}*.

5. Remove this trivial production, and redirect the dependencies accordingly.

6. Move the production *DropNode(b)* to directly behind *AddNode(b,perimeter,**operation**)*. This does not require redirection of any dependencies, since *DropNode(b)* only depends on *AddNode(b,perimeter,**operation**)*.

7. Transform the redundant subsequence *AddNode(b,perimeter,**operation**); DropNode(b)* into a single trivial production that only consists of invariants: *{-b}*.

8. Remove this trivial production. This concludes the redundancy removal, since no absorbing or redundant production pairs remain.

4.5 Refactoring Common Subsequences

In the context of team development, tool support is essential, especially when making parallel evolutions or customisations of the same software artifact. We can identify similarities between these changes by factoring out all commonalities between the parallel transformations. This is not only useful for reducing code duplication, but also during software merging to reduce the number of merge conflicts.

[6] The assertions *-source(*,b)*, *-target(*,b)*, *-type(b,*)* and *-label(b,*)* can be ignored as they are derived assertions of *-b*.

Figure 17: Factoring out commonalities in parallel evolutions

Schematically, the idea is represented in Figure 17. If we have two parallel productions *P* and *Q* that are applied to the same initial graph *G*, we can compare their assertions, and construct a new production *C* that contains only the common assertions, while the variable ones are specified in two other productions V_P and V_Q.

4.6 Undo Mechanism

In an industrial-strength software development environment, it should be possible to make changes undone selectively, even if these changes are part of a complex sequence. Suppose we want to undo only one production in a sequence. We cannot simply remove the production and reapply the resulting shorter sequence, because later productions in the sequence may still depend on the removed one. Therefore, we additionally need to remove all later productions that strongly depend on the removed production (either directly or indirectly).

For example, in order to undo *AddNode(b,perimeter,**attribute**)* in the sequence of Figure 7, we also need to undo all its strongly dependent productions *RetypeN(b,**attribute**,**operation**)*, *AddEdge(e,b,c,uses,**uses**)*, *DropEdge(e,b,c)* and *DropNode(b)*.

4.7 Parallelising Independent Subsequences

A final use of dependencies has already been discussed by Roberts [16]. In order to apply large production sequences in a more efficient way, they can be split up in parallel subsequences that can be applied independently from one another. This allows us to parallelise the process of applying complex transformations to a graph. It also makes large evolution transformations more manageable by splitting them up in smaller independent chunks that are more understandable.

For example, the production sequence of Figure 12 can be parallelised into the following independent subsequences:

*RelabelN(a,surface,area); RetypeN(a,**attribute**,**operation**)* and

*AddNode(c,radius,**attribute**); AddEdge(f,a,c,uses,**uses**)*

5. RELATED WORK

Perry was one of the first to use assertions for dealing with certain aspects of software evolution. In [15] he describes a *semantic interconnection model* that uses assertions to annotate software artifacts. This model is used to detect the effects of changes by recursively determining the assertions that are affected by the change. In our approach, we do not use assertions for expressing the behaviour of software artifacts themselves, but to express semantic dependencies between the evolution transformations instead.

If we focus on formal support for merging parallel evolutions, our work is closely related to [9]. Lippe and van Oosterom propose an *operation-based merge technique* that uses software transformations (called operations) to represent evolution, and detects and resolves merge conflicts using the information contained in these transformations. Dependency information between transformations is used to address the issue of scalability, but assertions are not used to identify the dependencies.

The research in this paper is a logical consequence of the work on *reuse contracts* [17]. Mens [10, 11] provides a formalism for reuse contracts that uses pre- and postconditions to express graph transformations and relies on formal properties of conditional graph rewriting [4, 5, 6].

The research of Roberts [16] is also closely related. Pre- and postconditions are used to express refactoring transformations (which are usually behaviour-preserving), and some scalability issues are addressed as well.

6. CONCLUSION

Typed graphs, combined with graph transformations that are based solely on assertions (i.e., preconditions, postconditions and invariants) provide a general formalism for software evolution. Assertions make it easy to detect syntactic merge conflicts between parallel evolution transformations, and allow us to define composite graph transformations in an intuitive and straightforward way. Dependencies between the assertions allow us to address several scalability issues, such as changing the order in a transformation sequence, removing redundant transformations in a sequence, and extracting a common subsequence from two (or more) given transformation sequences.

The approach seems very promising, but still needs to be validated in a large-scale case study. Also, the underlying formalism can be extended in many ways: a notion of subtypes could be introduced; more complex assertions could be defined; the productions could be made more generic; etc…

7. REFERENCES

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

2 D. Binkley, S. Horwitz, and T. Reps, *Program Integration for Languages with Procedure Calls*, ACM Trans. Softw. Eng. and Methodology, Vol. 4, No. 1, 1995, pp. 3-35.

3 M. S. Feather. *Detecting Interference when Merging Specification Evolutions*. Proc. Int. Workshop Softw. specification and design, pp. 169-176, ACM Press, 1989.

4 A. Habel, R. Heckel, G. Taentzer. *Graph Grammars with Negative Application Conditions*. Fundamenta Informaticae, Special Issue on Graph Transformations, 26(3,4): 287-313, IOS Press, June 1996.

5 R. Heckel. *Algebraic Graph Transformations with Application Conditions*. Dissertation, Technische Universität Berlin, 1995.

6 R. Heckel, A. Wagner. *Ensuring Consistency of Conditional Graph Grammars: A Constructive Approach*. Lecture Notes in Theoretical Computer Science 1 (1995), Elsevier Science, 1995.

7 C.A.R. Hoare. *An axiomatic approach to computer programming*. Comm. ACM 12(10): 576-580, 583. ACM Press, October 1969.

8 D. Jackson, D.A. Ladd. *Semantic Diff: A Tool for Summarizing the Effects of Modifications*, Int. Conf. Softw. Maintenance, IEEE Press, 1994.

9 E. Lippe, N. van Oosterom. *Operation-based Merging*. Proc. Fifth ACM SIGSOFT Symp. Softw. Development

Environments. ACM SIGSOFT Softw. Eng. Notes, 17(5): 78-87, ACM Press, 1992.

10 T. Mens. *A formal foundation for object-oriented software evolution.* PhD Dissertation, Vrije Universiteit Brussel, September 1999.

11 T. Mens. *Conditional graph rewriting as a domain-independent formalism for software evolution*. Proc. Int. Agtive '99 Conference, LNCS 1779: 127-143, Springer-Verlag, 2000.

12 B. Meyer. *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.

13 W.F. Opdyke. *Refactoring object-oriented frameworks*, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Technical Report UIUC-DCS-R-92-1759, 1992.

14 W.F. Opdyke, R.E. Johnson. *Creating abstract superclasses by refactoring*, Proc. ACM Computer Science Conference, pp. 66-73, ACM Press, 1993.

15 D.E. Perry. *Software Interconnection Models*. Proc. Int. Conf. Softw. Eng., IEEE Press, 1987.

16 D. Roberts. *Practical Analysis for Refactoring*. PhD Dissertation, University of Illinois at Urbana-Champaign, 1999.

17 P. Steyaert, C. Lucas, K. Mens, T. D'Hondt. *Reuse Contracts: Managing the Evolution of Reusable Assets*. Proc. OOPSLA '96, SIGPLAN Notices 31(10): 268-286, ACM Press, 1996.