Round-trip Support for Invasive Software Composition Systems

Jendrik Johannes, Roland Samlaus, and Mirko Seifert

Technische Universität Dresden, Computer Science Department,
Nöthnitzer Str. 46, 01187 Dresden, Germany
{jendrik.johannes,roland.samlaus,mirko.seifert}@tu-dresden.de

Abstract. The ever increasing complexity of software systems promotes the reuse of software components to a topic of utter importance. By reusing mature parts of software, large systems can be built with high quality. The Reuseware Composition Framework can compose components written in arbitrary software languages. Based on metamodeling these components are merged invasively. But, even though language independent composition is powerful to compose complex systems, one must consider that composition is not the only activity in developing a working systems by reuse. Many tests and validations can only be performed on the composed system. At that point, it is hard to (a) know from which component an error originates and (b) ascertain what the implications of changing something in the composed system are.
This paper presents an approach to propagate changes back to the correct source components and discusses the possible implications of changes made to composed systems. Furthermore, the implementation of the approach as an extension to the Reuseware Composition Framework is presented using two example applications.

Key words: Round-trip Engineering, Invasive Software Composition, Traceability

1 Introduction

To handle the increasing complexity software projects are faced with, many ways of building software were introduced. Some of these are driven by abstraction, for example the evolution from low-level programming languages like Assembler to high level languages like Java. Others are focused on reuse of existing parts of software. Composition systems somewhat belong to both categories and provide means for software developers to build more complex systems in less time.

Composing software can be performed on different levels of abstraction (e.g., by merging models versus composing code) and in various ways. Depending on the component model, the composition mechanism and the composition language, different kinds of composition systems can be built. Invasive Software Composition (ISC) [1] is one such composition technique that promotes composing software artefacts at development time based on their grammar. Since software is usually expressed using formal languages, composing sentences of these

languages based on the underlying grammar seems natural. The term *invasive* emphasizes that ISC allows composition on a very fine granularity. Components can thus drastically change their behavior after composition.

The approach of ISC has shown its applicability for a variety of composition problems [1–3]. However, while ISC is a powerful technique, there is a major drawback, which is not in particular specific to ISC, but rather applies to many composition systems, that must be considered. Composed software systems eventually run as a whole. Even if components are carefully isolated, they may interact with each other. Thus, whenever a failure indicates an error in the system, the question at hand is which component contains the problem. In particular using ISC can complicate answering this question, because fragments—the components of ISC—can be woven on a very fine-grained basis and employing multiple composition steps. Determining which fragment contains an error is far from trivial and very tedious if performed manually.

In addition to the problem of localizing the source component for a specific error, there is a need for changing composed software directly rather than fixing the source fragments. Instead of making repetitive changes to components and recomposing, developers want to state which composition result they want. Fragments should then be adopted automatically. Intuitively, propagating changes to source fragments can have drastic implications. Fragments might be used multiple times in one composition program or even by multiple composition programs, in particular when they are reused from libraries. Thus, changes to fragments must be carefully analysed.

The contribution of this paper is threefold. First, we show how extending ISC with tracing mechanisms, can support the mapping of composed artefacts to their source components. Second, we systematically analyse the implications changes to composed programs can have. In particular we discuss how arising conflicts can be resolved. Third, an implementation of the approach is presented.

This paper is structured as follows: Section 2 briefly explains the mechanisms used by ISC and how Reuseware implements these to compose both textual code and graphical model fragments. Afterwards the problems of tracing components and changing composed artefacts are discussed in Sect. 3. Then, in Sect. 4, we present how the Reuseware Composition Framework was extended to support the backpropagation of changes to source fragments. An evaluation follows in Sect. 5 where composition and editing of two different languages (Java and UML Activity Diagrams) is presented. After a comparison to related work (Sect. 6), we conclude and shed some light on future work in Sect. 7.

2 Background: Invasive Software Composition

ISC was first introduced in [1]. It proposes a composition technique based on source code rewriting. The components in an ISC system are source code fragments that may contain *variation points*. Variation points are places were other fragments can be inserted during composition.

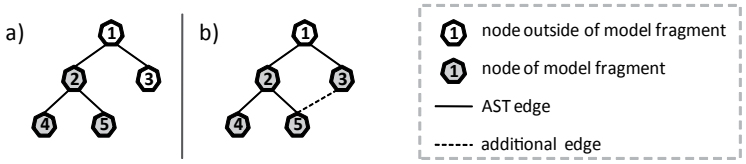

Fig. 1. Reuse of partial ASTs (a) and model fragments (b)

ISC variation points are typed based on the grammar of the language used to write the fragments. The grammar of a programming language, for instance, might define the concept `method`. In an ISC system, variation points for methods can be defined at the same positions where `methods` usually occur to be later replaced by concrete `method` fragments. This concept of grammar-based modularization originates from the programming language BETA [4]. Previous works [2, 5] on ISC generalised and extended the concepts of BETA. These extensions were implemented in the first version of the Reuseware Composition Framework[1], with which fragment composition systems for arbitrary languages can be built based on their context-free grammar.

Recently, Reuseware was extended to support languages defined by metamodels rather than context-free grammars [3]. A metamodel here is essentially a typegraph that describes the non-context-free structure of a language such that sentences of that language can be represented as graphs with a distinct spanning tree. Any metamodel described in the metalanguages EMOF [6] or Ecore (defined by the Eclipse Modeling Framework (EMF)[2]) fulfills these properties. An example is the UML2 metamodel [7]. A UML class diagram is an example for a sentence in the UML2 language.

The consideration of context increases the power of ISC. Reuseware can now be instantiated for arbitrary modelling languages and used in combination with a variety of tools in Model-Driven Software Development (MDSD), where language engineering by metamodeling is part of the development process.

In contrast to a traditional fragment (that is an instance of a context-free grammar—a tree called Abstract Syntax Tree (AST)), a fragment that is an instance of a metamodel is a graph with an underlying tree structure. Hence, it is an AST with additional edges as, for example, obtained after name analysis on the AST. Models in MDSD can be represented by such graphs. We use this terminology in this paper and refer to a (possibly partial) graph with the term *model fragment* or just *fragment* for short. Consequently, a node of such a graph is sometimes called *model element* or *element* for short.

When an AST fragment is reused in traditional ISC, a root node is reused together with all its children. The root node of an AST fragment can be any node of a complete AST (since fragments can be partial sentences). In Fig. 1a for example, Node 2 can be reused only together with Node 4 and 5.

[1] http://reuseware.org/history
[2] http://www.eclipse.org/emf

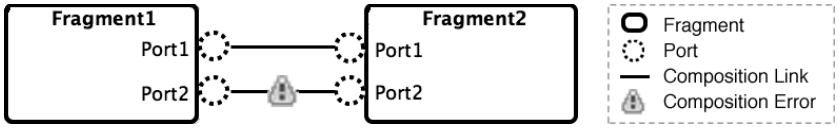

Fig. 2. A composition program defined in Reuseware's composition program editor

When reusing a model fragment, the interconnections between nodes need to be considered. The model fragment shown in Fig. 1b for instance, is comprised of a tree and an additional edge. If one wants to reuse Node 2, Node 3 also needs to be reused since it is connected with Node 5 which is a child of Node 2. An model fragment can thus have several root nodes that need to be treated together during composition.

Two kinds of nodes are considered during composition: *reference points* and *variation points*. Reference points are root nodes of model fragments (where one fragment can have several roots) and variation points are nodes that may be replaced during composition. Reference and variation points are grouped into *ports*. All ports defined by a fragment make up its *composition interface*. Concrete compositions are defined by *composition programs* where ports of different model fragments are connected by *composition links*. Ports may only be connected if the reference and variation points behind them match; that is, there is a sufficient number of reference points to replace the variation point, where the types—defined by the metamodel—of the points match.

Figure 2 shows a composition program defined in Reuseware's composition program editor. The rounded boxes represent fragments and the dotted circles attached to them are the ports of their composition interfaces. Each port has a unique name and internally maps to one or many elements inside the fragment. The lines between ports represent composition links. If ports that do not match are linked, the composition link is marked with a warning.

A composition program can be executed to merge single model fragments into a complete model by manipulating elements behind ports that are linked. If several model fragments are involved, the composition is executed stepwise following the algorithm outlined in [3]. For large composition programs, the origin of single nodes inside the invasively merged result is hard, sometimes impossible, to determine. This problem is tackled in this paper. More details about Reuseware and its composition concepts are explained in the following sections where needed. For further details please consult [3].

3 Approach

Establishing Round-trip support for a composition system can also be considered building a decomposition system. If one can break up an assembled software into its original parts, an inverse operator is available, and changes can be easily made to the composite and transformed back to separate parts. Unfortunately, exactly

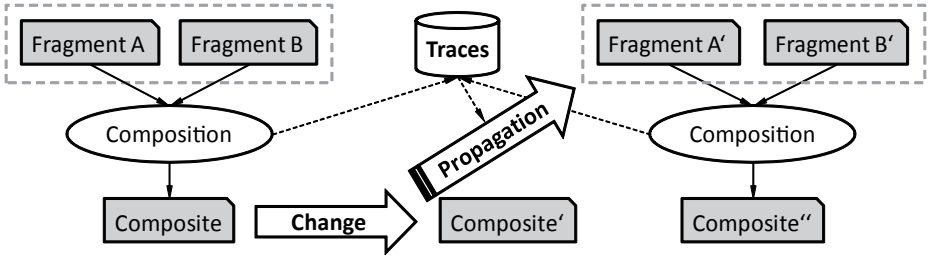

Fig. 3. Round-trip support using Backpropagation

finding such an inverse operator is not easy or even impossible for arbitrary composition systems in general.

Thus, instead of seeking for an inverse operator (i.e., a decomposition operator), a different way seems more promising for ISC. Inspired by [8], which presents Round-trip support for aspect weaving of textual languages, we choose to trace the composition of software artefacts as illustrated in Fig. 3. More precisely, tracing means that the composition process is observed and each composition step is recorded to a log. By doing so, we can obtain a mapping from elements in the composite to the respective elements in the source fragment. In [8] this mapping is established on a very low level—using line numbers and column positions. To propagate changes made to composites back to the respective source fragment such a mapping can be consulted. Instead of making the change to the composite and applying an inverse operator, the changes are made at the correct positions in the respective source fragment and the composition is reexecuted. This step is called *Replay* in [8]. Note that the procedure shown in Fig. 3 is not restricted to two source fragments or a one-step composition.

What may sound simple at first, turns out to be more complex. First, recording the composition steps must be accomplished. This is merely a technical issue and the easiest part. Second, the recorded log must be sufficient to map every element from a composite to its source. This is self-evident for existing elements, but different for elements that are manually added to a composite. These do not have counterparts in the source fragments. Nevertheless, additions must be handled properly to obtain full Round-trip support. Third, once a change has been propagated back to the correct source fragment, the rerun of the composition can cause problems. Changes may alter other parts of the composite, modify other composites that use the same fragments or even cause composition to fail, because the composition interface was changed.

In this section we examine these questions in detail. Following the previous problem description, Sect. 3.1 describes how the composition process can be traced in ISC systems. Subsequently, Sect. 3.2 classifies changes that can be made to composed artefacts. As mentioned before, conflicts can arise when the composition is executed again. These conflicts are discussed in Sect. 3.3.

3.1 Tracing Composition Program Execution

The composition technique employed by ISC is insertion of ASTs into each other. Recent extensions (see Sect. 2) also allow to compose graphs instead of trees. Thus, a composition program inserts graphs into graphs. These graphs are typed (i.e., nodes are instances of meta classes) and own attributes (properties of meta classes). Tracing a composition of such graphs essentially means storing information about the origin (source fragment) of each node.

For tree-based compositions it is sufficient to store the root node whenever a tree is inserted into another. This was accomplished in [8] using the line and column information. If graph structures are composed, tracking each node may be necessary. Technically, the composition system must be extended to support the creation of log records whenever an atomic composition step is executed. For each step a log entry must be created, that encapsulates information about which fragment was inserted into which other fragment and the respective location.

By doing so, we obtain a mapping from elements in the composite to their counterparts in the source fragment. In a multi-step composition this may not necessarily be a direct mapping, but an indirect one. Nevertheless, this is all information needed to map nodes to their source fragment. Intuitively, this mapping can be used when attributes of a node (e.g., its name) is changed. But, there are other types of modifications where the backpropagation is not that easy. Thus, we will discuss the possible types of changes in the next section.

3.2 Backpropagating Different Types of Changes

One can distinguish between three different kinds of change operations: updates, insertions and deletions. Updates are changes performed on attributes, insertions refer to adding an element to the composite and deletions correspondingly refer to the removal of an element.

Updates can be handled by mapping the node, containing the changed attribute, back to its source using the traces. Thus, the attribute can simply be changed in the original node and the change is propagated back successfully.

For deletions and insertions the scenario is more complex. Here, we distinguish between modifications "within" and "between" fragments. To explain the difference, lets recapitulate the structure of the composites. As we operate on graphs, composites contain nodes connected by edges. Each node corresponds to a single source fragment. If fragments contain multiple nodes, which is usually the case, these nodes can be found connected to each other in the composite. Now, adding or deleting elements in such a coherent part of the composite is called an *internal edit*. In contrast, some elements in the composite reside on variation points. Modifying such elements is called an *interface edit*.

Insertions of new elements can have different intents according to these edit types. Since new elements did not exist before, they never have a counterpart in the source fragments. Obviously, inserted elements must be added to the source fragments in such a way that recomposition leads to a composite containing the inserted element at the correct position. For internal edits this corresponds to

Fig. 4a. The node that is added to the composite (depicted by a bordered grey circle) must be added to fragment B to obtain the same result after composing again. If it is added to one of the other fragments (A or C), the composition result would not only be different, but not contain the new node at the position where it was added. Thus, internal insertions can and must be handled by adding the new node to the respective source fragment.

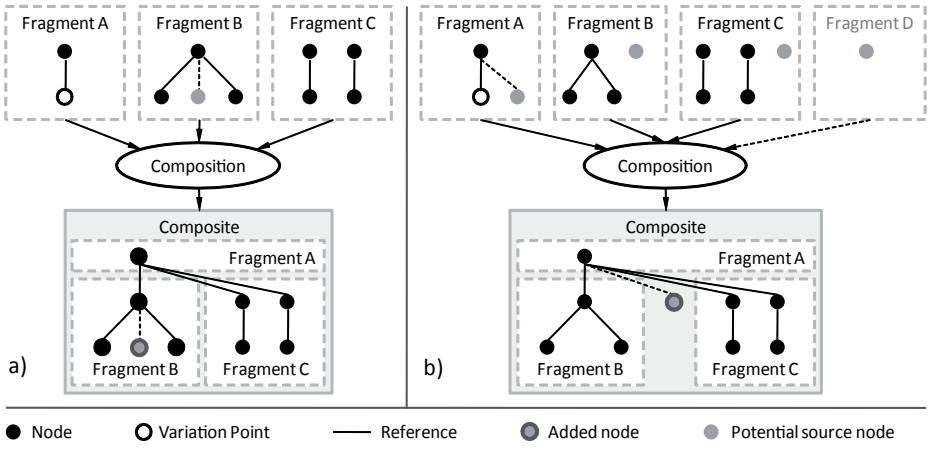

Fig. 4. Possible insert scenarios

However, some node insertions are interface edits. Consider, for example, Fig. 4b. An insertion of a new node between (depicted by a bordered grey circle) can not be clearly assigned to fragment A, B or C. The new node could be added one fragment or the other—recomposition would create the same composite. Adding this node could even mean that a new fragment D must be created containing the just added node.

We call this the *gap edit problem*. A basic example for this problem can be observed when editing programs that were augmented with advices using Aspect-oriented Programming (AOP). If code is added to the composed program at the begin or end of an advice, it is not clear whether the added code belongs to the core or the advice. We will present a concrete example in Sect. 5.1.

Deletions can be propagated back by removing the corresponding element in the source fragment if they are internal. Interface deletions, however, can have different meanings. For example, if an element is removed that was inserted at a variation point during the composition. In this case the intent of the modification can be (a) to remove the corresponding element in the source fragment, (b) to remove the variation point or (c) to remove the link between the source fragment and the variation point in the composition program. All three cases are equally valid and can not be distinguished automatically.

After looking at the backpropagation of atomic changes we can say that updates can be handled automatically, but insertions and deletions may require developer decisions. However, propagating changes to the source fragment was merely the first step of our approach (see again Fig. 3). After the changes are made to the respective source fragment, the composition is executed again. This may cause further problems as we will discuss in the next section.

3.3 Change Conflicts

As mentioned in the general description of our approach, the second step in synchronizing composites and fragments when change is made to the former, is recomposition. The execution of the composition propagates the changes just made to fragments to the composite. If this succeeds and the changes originally made to the composite are present again, we consider the synchronization to be successful. Unfortunately, this is not always the case.

First, a fragment can be used multiple times in a composition program. Thus, it potentially appears more than once in the composite. Changes made to such a fragment are consequently propagated to all its occurrences in the composite. This corresponds to changing an advice in AOP. While this is appropriate in some scenarios, in others the *intention* behind a change can be completely different. For example, the intention behind a single change might be to modify only the particular part of the composite that was changed directly by the developer.

Second, a fragment can be used in multiple composition programs. Again the execution of the composition will propagate changes to places other than the one that was changed by the developer directly. Similar to the just mentioned first case, the intention of the developer is not clear. Some modifications might be conveniently applied to the whole set of composites, while others are meant to change only exactly the single composite that was manually modified.

Third, modifications may cause a different composition result if they change the composition interface of a fragment. This can happen when elements representing reference or variation points (cf. Sect 2) are changed. The attributes of these elements determine if they represent a reference or variation point. For example, an element named `VP_<port_name>` might represent a variation point and an element named `RP_<port_name>` a reference point (where `<port_name>` identifies the port to which the element belongs). If such an attribute (e.g., `RP_Port1`) is changed (e.g., into `NewName`) it may alter ports (e.g., `NewName` is removed from *Port1* since it no longer starts with `RP_`).

In general, one can say that whenever reexecuting a composition causes more changes than the ones that triggered the synchronization, the intent of the changes must be clarified. Modifications might be meant to apply to all occurrences or only a specific one. The intention of a change can be either specific to a single change or general for a set of changes. In the first case, developers must provide information about their intent for each change. If the intention is more general (e.g., modifications are always meant to be propagated within one composite, but not to others), developers can state their intent once and subsequent synchronizations handle changes automatically.

Now that we have argued that it is impossible to determine the intent of each change fully automatically, two more questions arise:

1. **How to determine the developer's intent?** Developers may not make changes with a clear intent in mind. They rather experimentally perform changes and realize the different implications only after recomposition. Thus, the question is how we can support developers in determining their intent.
2. **How to translate the intent into action?** Once the intent of the developer (e.g., "change this particular element and no other" or "change every occurrence of this element") is found, appropriate actions must be taken. This might require not only modification of the original fragments, but also of the composition programs. The latter is needed if a change should not be propagated to all other occurrences. In this case fragments need to be copied and composition programs need to be adopted to use the copies instead of the original fragments.

4 Implementation

In order to give answers to the questions raised in the previous section, we extended the Reuseware Composition Framework with support for change backpropagation and conflict resolution. First, to propagate manual changes to source fragments, additional information must be gathered during the composition process. Before a fragment is composed into another one, each of its elements is copied by Reuseware. This copying step was extended such that each copied element in a composite is linked to its respective original. Since Reuseware and the modelling tools it works with are based on the EMF, this was technically realized by attaching additional information to the elements through a mechanism called EMF Adapters.

The second functionality is observing the manual editing of composed models. These edits are done in EMF-based (graphical or textual) model editors. Thus, EMF Adapters, which can also serve as listeners, are used to notice changes during editing. When the developers intent of a particular change is unclear, a so-called `ConflictResolver` is consulted. We added an extension point to Reuseware at which `ConflictResolver`s can be registered and implemented one—the `InteractiveConflictResolver`. With the `InteractiveConflictResolver` we tackle the two questions raised in Sect. 3.3 in a generic way. In the following we show how it clarifies the developer intent at different positions (Question 1) and how Reuseware reacts upon the intent (Question 2).

Identifying the source fragment. Remember the three change possibilities discused in Sect. 3.2: insertions, deletions and updates. The first conflict that requires developer interaction occurs only in the case of insertions and deletions. The gap edit problem (Sect. 3.3) requires the developer to specify to which source fragment the change should be backpropagated. By analysing the change, the tool assembles a list of possible sources to only present valid options to the developer. Figure 5 shows the corresponding dialogue presented by the `InteractiveConflictResolver`.

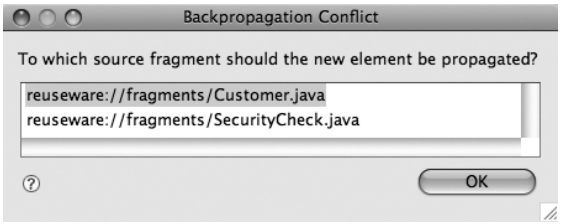

Fig. 5. Gap edit conflict triggered by a change in a composed model

Propagating the change. The next conflict occurs for any type of change, when the source fragment is used more than once either in the same or in other compositions. In general, there are three options to resolve this issue: (1) discard the modification completely, (2) propagate the modification and accept all implied changes or (3) create or copy source fragments to include the changes and adapt the composition program accordingly. The first option is the exit for the developer if he realises at this point that the change was wrong. It is simply resolved by reverting the change. With the second option, the developer confirms that the change was made with the intent to propagate it forward to all uses of the corresponding source fragment. It is translated into action by changing the source fragment. The developer should choose the third option if his intention was to modify exactly the changed element only and not to forward propagate modifications elsewhere. This choice triggers another developer interaction, in which the developer has to define a name for a copy of the corresponding source fragment. The tool then copies the fragment, stores it in the repository under the new name, and changes the current composition program to use the copy. The third option might be the only choice in cases where the fragment in question is reused from a library and all the implications of changing such a fragment can not be detected by the system or the developer.

Review forward propagation result. After the developer decided for option two or three above, he gets the chance to review all the implications of his decision. This is in particular important when forward propagation to all occurrences was chosen. However, any change in a source fragment can have unpredicted implications when it changes the fragment's composition interface. To support the developer, the tool composes a preview of all affected composites by temporarily changing the source fragment. It also provides a compare view (utilising the EMF compare tool[3]) where the developer can directly compare the recomposed with his manual modified model.

Final confirmation and actions. If the developer thinks that the chosen propagation method will not affect any composition in a negative way, he can confirm his decision and the temporary change of source fragments will be persistent. He can also choose to discard his initial change at this point which will roll-back the temporarily made modifications. If the backpropagation was

[3] http://www.eclipse.org/modeling/emft/?project=compare

accepted despite of errors that may have appeared in composition programs, the developer has to do additional adjustment there. Luckily, Reuseware reports all such errors directly in its graphical composition program editor, where the problems can be reviewed and resolved.

To summarise, the implemented `InteractiveConflictResolver` allows to easily assess the implications of a modification and choose an option that reflects the intent of the modification made. For dealing with changes that can have more than one meaning this is the best one can do. For specific applications or projects, default strategies (e.g., always copy fragments) could replace parts of the GUI through an alternative `ConflictResolver`.

5 Example Applications

In this section we use two example applications to show how the extended Reuseware is used to synchronise modifications and to resolve conflicts. To stress that this can be done for arbitrary languages, both textual and graphical, we present one application on using Java in Sect. 5.1 and a second one using UML Activity Diagrams in Sect. 5.2.

5.1 Fixing Bugs in Composed Java Programs

In previous works we successfully used ISC to compose programs and in particular to introduce new language features (e.g., aspect-orientation or modularization [2, 3]). The research in this area showed, that fragment composition is powerful, but also revealed, that it can interfere with other development activities. This is because composition is forward-oriented, but composed fragments must usually undergo further validation (e.g., unit tests). Consequently, if errors are found, they must be fixed—ideally right where they appear—in the composed fragment. To avoid inconsistencies these fixes must be propagated to the original source fragments.

To show that the presented work can serve as a solution to this problem, we used Reuseware to compose a core application with additional code that implements security checks. To perform these checks, methods that provide access to sensitive data are augmented with a static call to class `SecurityManager` that throws an exception if the current thread does not have the needed privileges. Figure 6 shows two fragments A and B and the composition program we used to generate the composition result (Note: fields were omitted).

In this example the statements of every method were selected to be variation points. Thus, each method can be connected to another fragment in a composition program. Consequently, the fragment B is inserted into the two methods as shown in the composite in Fig. 6.

Now consider a developer executes a unit test on the composed fragment. Assume, the test fails because the method `getSSN()` returns the social security number even though the calling thread does not have the required privilege `AUTH`. Therefore the test expects that an exception is thrown. Subsequent debugging

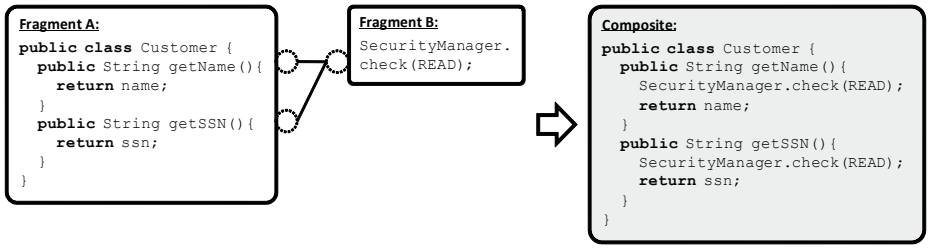

Fig. 6. Initial Composition of Java Fragments

reveals the error—there is no check for this privilege. Naturally, the developer wants to fix the bug right away and then run the unit test again to verify that the modification indeed removes the undesired behaviour. Therefore he adds a second statement `SecurityManager.check(AUTH)` to make sure the exception expected by the test is thrown.

Now the test succeeds, but the fix must be propagated to the source fragments to restore a consistent state. In this case the gap edit problem is raised. The new statement is inserted "between" the two fragments. It could be added to fragment A as well as fragment B. The developer decides to add the new statement to fragment B, because a security check does not belong to the application core. But, fragment B occurs twice and a second decision is needed. Either all occurrences must be changed or only one. Retrieving a customers name does not need the privilege `AUTH`, so he picks the latter. Thus, fragment B is copied and the composition program is automatically changed as depicted in Fig. 7.

5.2 Business Process Modelling with UML Activity Diagrams

Processes supported by software systems can be described by behaviour modelling using UML Activity Diagrams. Often, general processes (e.g., a process for ordering goods in a shopping system) can be defined once and specialised for a concrete system. Such specialisations can be performed by invasive composition with Reuseware, as demonstrated in [3]. Each *fork*, *join* or *merge* node of an activity is used as reference point and each *initial* or *final* node as variation point.

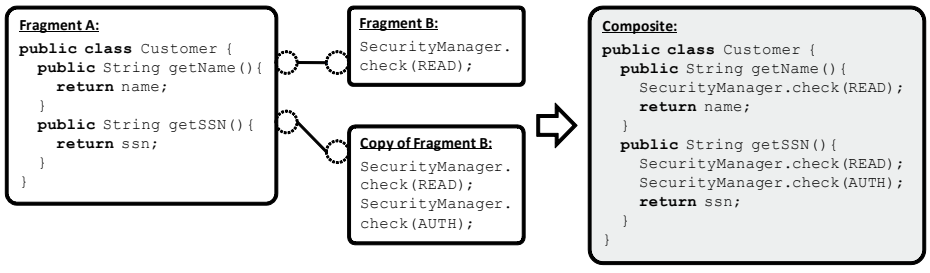

Fig. 7. Modified Composition of Java Fragments

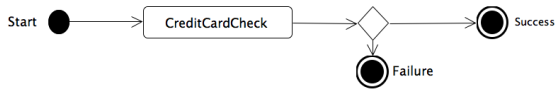

Fig. 8. The original *CreditCard.uml* fragment

Each reference and variation point is mapped to a port with the same name as the node in upper-case. This way, activities can be woven into each other by effectively replacing *initial* with *fork* and *final* with *join* (or *merge*) nodes.

In the scenario described in [3] an activity for a payment specific consistency check (*CreditCard.uml* shown in Fig. 8) is added to a generic ordering process through the composition program of Fig. 9a. The composed model (cf. Fig. 10) can then be used to perform model-based performance simulation of the composed system as done in [9]. In this setting, results from the simulation are fed back into the activity diagram to suggest process improvements. Thus, it is an analysis that can only be performed on the composed model.

In the example, the simulation might report that the `CreditCardCheck` action is indeed a bottle neck delaying the whole activity since it involves communicating with external systems. The process developer might come up with the solution to not wait for the `CreditCardCheck` before continuing with order processing, but rather let it continue in parallel to other actions and cancel the whole activity later in the case the `CreditCardCheck` fails. This can be expressed, as shown in Fig. 11, by removing `Join1`, introducing a new join node `Join2` and altering the outgoing transition of `CreditCardCheck` to point at `Join2`.

Since the new element `Join2` is connected with elements originating from both *CreditCard.uml* and *OrderProcessing.uml*, the gap edit problem applies. Furthermore, the composition requires adjustments, since `Join1` and `Join2`, being join nodes, both influence the composition interface.

When the developer performs the change, the synchronization mechanism informs him about a conflict. After he decided that the new element should be added to *OrderProcessing.uml* (since it is part of the main ordering process flow), the tooling lets him review the impact of his decision. By reviewing the recomposed model in the compare view, he realises that his change was not preserved after recomposition—the altered transition is now missing.

The developer discovers that the composition program is now erroneous. Reuseware reports erorrs directly in its graphical composition program editor

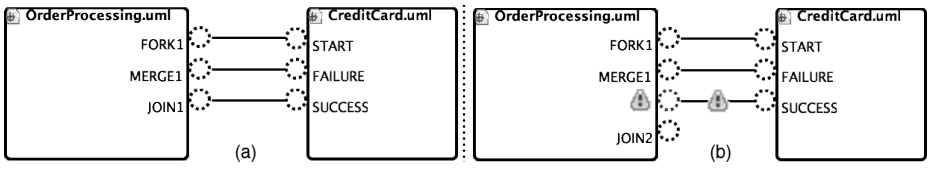

Fig. 9. Composition program (a) before and (b) after modification

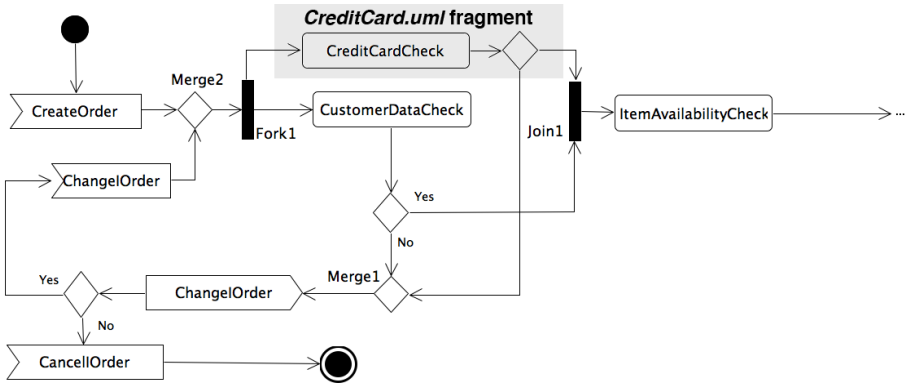

Fig. 10. Composition of the *OrderProcessing.uml* and *CreditCard.uml* fragments

(cf. Fig. 9). In this case, the tool reports, as shown in Fig. 9b, that the port `JOIN1` is missing and the link between `JOIN1` and `SUCCESS` is therefore invalid. It also updates the interface of the *OrderProcessing.uml* fragment in the composition program, making `JOIN2` appear as additional port. The developer can now adjust the composition program by linking `SUCCESS` with `JOIN2`.

This manual alteration of the composition program in obvious cases like this could possibly be automated as well. However, it is important to stress that, whatever complex changes the developer does on a composed model, failing to reexecute the composition will be immediately reported. The developer can then resolve the issue by investigating the errors reported by the composition engine.

6 Related Work

The term Round-trip Engineering (RTE) was first mentioned in the context of MDSD in [10] and besides a description of arising issues, general qualities that are desirable were given. Our approach offers these qualities, namely the ability to manage trace information, the intuitiveness and conciseness, as well as understanding the intention of users and assistance to detect conflicts. Furthermore, we provide assistance to resolve conflicts. A more formal description of Automatic Round-trip Engineering was given in [11]. However, as shown in this paper, the domain transformations used in ISC (i.e., the composition operators) are not

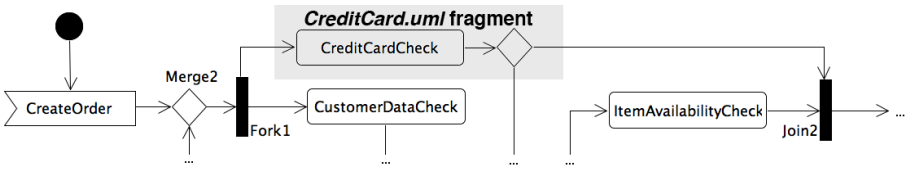

Fig. 11. Manually modified composed model

always invertible. This prevents us from completely automating our Round-trip support as described in [11].

Not necessarily based on models, but in the field of RTE, the AOP community investigated how the results of invasive aspect weaving can be analysed [12] and changed [8]. In contrast to ISC the first part is already problematic for some AOP implementations, that do not weave source, but binary code. Those that compose source fragments often do so on a textual basis. As a consequence, the composition programs are not strongly typed. Rather strings are inserted into other strings. Thus, detecting broken composition programs, as we did in Sect. 5.2 is not possible. In addition we process graphs rather than trees. References are therefore safely preserved during composition and backpropagation. In contrast, tree-based approaches (e.g., [8]) must use identifiers which can only be composed safely if they are unique. However, unique identifiers are hard to choose if the set of compositions a fragment is used in is not fixed.

As Reuseware operates on models, compositions can also be considered as model transformation. Hence, bidirectional transformations [13] are directly related to this work. However, mapping compositions to bidirectional rules imposes some problems, which is why we choose a different approach. To illustrate this problems, we will compare our work with Triple Graph Grammars (TGGs) [14], which are the most sophisticated approach in this area.

TGGs can be used to link models with rules that can be executed both forward and backward. Mapping composition programs to such rules is therefore a possibility. However, it is not clear whether copying fragments, which is necessary in ISC, can be realised with TGG rules. Moreover, TGGs synchronize model elements only. Propagating changes to attributes, which we consider a frequent activity for editing graphical models, needs extensions to the classical TGG formalism. Furthermore, TGGs can propagate the addition and deletion of model elements, but do not distinguish between fragments (i.e., parts of the models). As discussed earlier, it is important to detect where changes are propagated to. If one cannot detect the propagation of a modification to another occurrence of a fragment or to another composite, changes are always propagated. Fine-grained decisions about the intent of a change, as we allow them, are not possible.

7 Conclusions and Future Work

Composing artefacts using ISC is a powerful approach for reusing parts of software. This paper showed how to extend ISC with Round-trip support. Results of compositions can be modified and synchronized with their source fragments. This promotes ISC from a forward directed method to a bidirectional composition technique. Admittedly, the synchronization or propagation process cannot always be performed automatically. In some cases the intent of the developer must be known to correctly transform the changes made to a composite to source fragments or the composition program. This is not a drawback of our approach, but merely bound to the specific composition technique employed by ISC.

The approach was implemented as an extension to the Reuseware Composition Framework to show its practical applicability. Furthermore, to ease the process of determining the intent of modifications, different possibilities to show the implications of a change have been investigated and presented.

Still, important questions remain open. First, only basic composition operations where studied in detail in this paper. If more complex operations are involved in composition, the round-trip is less automatic. This could be the case when a modified element was part of different ports on the composition interface or was extended with a number of elements during several composition steps. In such cases, the composition programs breaks and requires manual adjustment. As mentioned, the good thing is that conflicts are always immediately visible due to the composition engine's error reporting. However, we also showed that composition programs can be automatically corrected to a certain degree (cf. Sect. 5.1). In the future, we will study which other automatic adjustments to the composition programs are possible (e.g., the one indicated in Sect. 5.2).

Second, all changes were propagated individually. Our evaluation turned out that this is not feasible in some scenarios. Asking developers after each keystroke or mouse click about their intention and whether they want to put the new character or arrow into one fragment or the other is not appropriate. Rather, developers should be able to make a set of changes and then trigger the synchronization. However, this may result in more complex conflicts, because modifications within one change set may interfere. Analysing such conflicts is therefore also subject to future work.

Third, the intention of changes (and in particular the fact that it might not be known) prevents us from fully automating the Round-trip support. Nonetheless, for some composition applications or languages there might be reasonable default strategies. For example, when ISC is used to introduce aspect-orientation to a language, edits in an advice may always be propagated to all places where the advice is inserted. When fragments have proven stable and general reusable over time they can be published in fragment libraries to be reused in different systems. Such maturity information about fragments can be taken into account for automatic decision for the gap edit problem (e.g., always modify the less mature fragment) and for deciding between modification and replacement by copy (e.g., always copy and never modify fragments from public libraries). This functionality could be added by annotating fragments in Reuseware with a maturity status. Furthermore, composition programs could hold information about the propagation strategy to use. This way, developers can explicitly state their intent for individual composition steps.

References

1. Aßmann, U.: Invasive Software Composition. 1 edn. Springer Verlag (April 2003)
2. Henriksson, J., Heidenreich, F., Johannes, J., Zschaler, S., Aßmann, U.: Extending Grammars and Metamodels for Reuse: The Reuseware Approach. IET Software **2**(3) (2008) 165–184

3. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Language-Independent Model Modularisation. Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE (2009)
4. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, Reading, MA, USA (1993)
5. Henriksson, J.: A Lightweight Framework for Universal Fragment Composition—with an application in the Semantic Web. PhD thesis, Technische Universität Dresden (January 2009)
6. Object Management Group: MOF 2.0 core specification. OMG Document (January 2006) URL http://www.omg.org/spec/MOF/2.0.
7. Object Management Group: Unified Modeling Language: Superstructure Version 2.1.2. Final Adopted Specification formal/2007-11-02 (2007)
8. Chalabine, M., Kessler, C.: A Formal Framework for Automated Round-Trip Software Engineering in Static Aspect Weaving and Transformations. In: Proceedings of 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, USA (2007) 137–146
9. Fritzsche, M., Johannes, J.: Putting Performance Engineering into Model-Driven Engineering: Model-Driven Performance Engineering. In: MoDELS'2005 Satellite Events: Revised Selected Papers, Lecture Notes in Computer Science 5002, Springer (2007)
10. Sendall, S., Küster, J.M.: Taming Model Round-Trip Engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development, Vancouver, Canada (2004)
11. Aßmann, U.: Automatic Roundtrip Engineering. Electronic Notes in Theoretical Computer Science **82**(5) (2003)
12. Eaddy, M., Aho, A., Hu, W., McDonald, P., Burger, J.: Debugging Aspect-Enabled Programs. In: Proceedings of the 6th International Symposium on Software Composition (SC2007). (2007) 209–225
13. Stevens, P.: Towards an Algebraic Theory of Bidirectional Transformations. In: Proceedings of 4th International Conference on Graph Transformations (ICGT2008), Leicester, United Kingdom. (2008) 1–17
14. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching, Germany, Springer (1994)