Taming Model Round-Trip Engineering

Shane Sendall and Jochen Küster

Computer Science Department
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{sse, jku}@zurich.ibm.com

Abstract: Round-trip engineering is a challenging task that will become an important enabler for many Model-Driven Software Development approaches. Model round-trip engineering involves synchronizing models and keeping them consistent, thus enabling the software engineer to freely move between different representations. This vision of complete round-trip engineering is only realized to a limited degree in tools nowadays, and it proves to be a very difficult problem to solve in general. In this paper, our goal is to clarify some of the issues in automating round-trip engineering and point out some of the high-level qualities that are desirable for round-trip engineering approaches to possess. Clarifying this domain is an important first step towards being able to systematically automate round-trip engineering of models.

1. Introduction

Many software-intensive systems are sufficiently complex that it is necessary to make use of abstraction, projection, and decomposition to understand, communicate, and/or maintain them. Software modeling practices offer a means to exploit the benefits of these techniques in coping with complexity and size, not only for capturing solutions but also the problems that are being addressed. OMG's Model-Driven Architecture (MDA) initiative [Omg03a, KWB03] proposes a three-layer architecture, where each layer contains possibly many models, and models are related within and between layers.

The use and benefit of multiple models is motivated by the principle of separation of concerns: there should be a close correlation between the kinds of models used to specify the system and the kinds of concerns of the stakeholders (technical and non-technical) and their ability to identify, express and validate their concerns with respect to one or more models. Under this scheme, most modern-day software systems are likely to require and benefit from many interrelated models.

Making use of multiple models in software development activities requires, at least, that each model is kept consistent with the others if they are interrelated. The ability to automatically maintain the consistency of multiple, changing software artifacts in software development environments/tools is commonly referred to as round-trip engineering. Changes made to one model can be propagated and reflected in another model using model transformation technologies [SK03, CH03].

In this paper, our goal is to clarify some of the issues in automating round-trip engineering and point out some of the qualities that are desirable for round-trip engineering to possess. In section 2, we present an example that illustrates a challenging situation that can arise in round-trip engineering but does not occur in either forward or reverse engineering in isolation. In section 3, we describe the problem domain of round-trip engineering and explain the different aspects of the problem that need to be addressed to automate it. In section 4, we propose a set of high-level qualities that are desirable for round-trip engineering approaches to possess. In section 5, we review the current state-of-the-art in round-trip engineering. In section 6, we summarize the paper and propose future work.

2. Round-Trip Engineering ≠ Forward + Reverse Engineering

In this section, we give some background on round-trip engineering and describe a situation that can arise in round-trip engineering but does not occur in either forward or reverse engineering in isolation.

A large number of software development environments/tools offer support for graphical models, which are typically a subset of the models possible with the Unified Modeling Language (UML) [Omg03b] (see [OD04] for a list of tools that support UML). Many of these tools provide automated support for generating programming language code from UML class diagrams and also generating UML class diagrams from programming language code (see [Xde04, Bor04, Gen04], for example). The first task is referred to as forward engineering and the second one, as reverse engineering.

More generally, forward engineering involves generating one or more software artifacts that are closer in form and level of detail to the final deployable system compared to the input artifacts, and reverse engineering involves generating one or more software artifacts that abstract certain details of, and possibly present in a different form, the input artifacts, with the goal of recovering any information lost in the forward step. Some tools also support round-trip engineering, which involves both forward and reverse engineering steps such that software artifacts, such as, programming language code and UML class diagrams, become synchronized at certain points in time.

Most development tools only offer very limited support for round-trip engineering. This is most probably a consequence of the difficulty in keeping multiple changing artifacts consistent. It is important to note that it may *not* be necessary to allow multiple artifacts to be changed over the same period, and it certainly simplifies the problem if only one artifact is allowed to change and the other ones are simply (read-only) views. For example, round-trip engineering is typically not required between programming language code and binary code because it is assumed that the binary code will not be changed (which may not always be a valid assumption[1]). In this situation, forward engineering is the compilation process, which produces the binary, and reverse engineering, which is typically used when the original source code is no longer available, produces source code similar (but usually not the same, due to information loss in the optimization activities of compilation) to the original source code.

Both forward and reverse engineering steps involve transforming one or more artifacts into one or more other artifacts. This task is typically a single, isolated transformation, where any information in the target artifact is not considered, and typically a new artifact is created, possibly replacing the previous version, if one exists. In some cases, forward and reverse engineering may be optimized so that only incremental transformation is performed, i.e., only the changed modules are transformed, rather than all artifacts, e.g., incremental compilation.

In contrast, round-trip engineering requires that information in the target artifact is preserved and not "clobbered" on the return trip. This also indicates that the intention behind round-trip engineering is to *reconcile* models, rather than just transform them in a given direction. For example, in round-tripping UML class diagrams and Java, it would usually be undesirable for the names and form of the associations to be changed in the class diagram if an unrelated change is made to the Java code. Such a scenario is conceivable if only reverse engineering is applied, as associations are not

[1] As compilers have become better at producing code, there is less of a tenancy to change the binary code. However, this was not the case in the early days of compiler technology.

first-class in Java and thus would need to be generated without knowing their original property values. This kind of situation arises whenever there is not sufficient information in the source artifacts to reconstruct the target artifacts to their previous form [KW03].

To illustrate such a situation, Figure 1a shows two models that are related by a transformation, called "SquashClassHierarchyTrans", that maps the class hierarchy defined in the LHS model to a single "squashed" class in the RHS models, i.e., all of the most specialized methods and attributes of the class hierarchy are mapped into the single RHS class. Now, considering the return trip, Figure 1b.i shows that the transformation from RHS to LHS may result in any number of solutions (theoretically, an infinite number).

If reverse engineering is applied to generate the LHS model, it does not consider the existing form of the LHS class hierarchy, and as such, it would typically just use some form of default configuration or let the user decide, e.g., it creates two classes with an inheritance relationship between them. Note that the information about the class hierarchy was lost in the forward engineering step, i.e., the mapping from left-to-right. In the round-trip engineering case, it is necessary to be able to trace back the original classes: *A*, *B*, *C*, on the return trip (see Figure 1b.ii) and apply the necessary changes to these classes in accordance with the changes made to class *X*. It is also interesting to note that if *X* was deleted and replaced by a class *Y*, then the round-trip would be nothing more than a reverse engineering step, because this information would not be applicable. From hereon in, we refer to the information recorded to track the relationships established between elements as trace information.

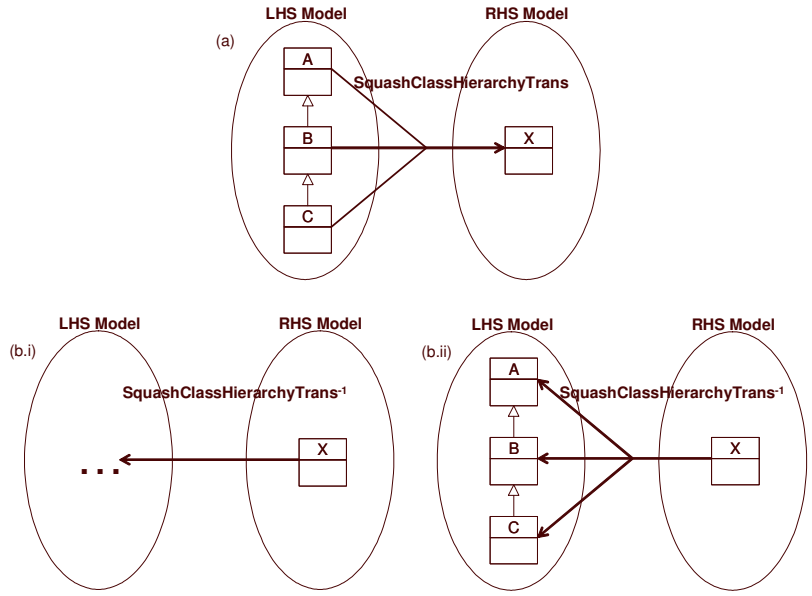

Figure 1: Mapping a class hierarchy to a single "compact" class: (a) mapping a class hierarchy to a single class; (b.i) the inverse mapping from a single class to a class hierarchy (any number of solutions are possible for LHS); (b.ii) tracing back to the class hierarchy from class in the RHS Model.

In this paper, we concentrate on round-trip engineering of models, where we (artificially) define a model to be any software development artifact that has, and

complies to, a corresponding metamodel, for example, specified using OMG's Meta Object Facility (MOF) [Omg02b]. This bias also points to our focus on model transformations that make use of metamodel definitions in accessing metadata [Sen03a, Sen03b].

For the sake of restricting the scope of this paper and our work, we do not address the issues that surround concurrent updating of models by assuming that each change is only made to a model if it is in a stable state. Furthermore, we do not address performance issues in propagating and applying highly frequent changes. As such, both these areas (which have been well-studied in the domain of information systems) are out of the scope of this paper.

3. The Problem Domain of Model Round-Trip Engineering

In this section, we briefly describe the problem domain of model round-trip engineering and explain some of the issues involved in composing round-trip engineering activities.

Given a set of models (possibly at varying levels of abstraction), round-trip engineering is a technique that allows the software developer to move freely between these models and change them as seen fit, where changes made to each one are reflected in the other ones, keeping each one consistent with the others.

Round-trip engineering can be divided into three steps:

1) deciding whether a model under consideration has been changed,
2) deciding whether the changes cause any inconsistencies with the other models, and
3) once inconsistencies have been detected, updating the other models so that they become consistent again.

Typically, the first step is trivially realized in modern day tools, e.g., the Eclipse Modeling Framework (EMF) [BSM+03] provides direct support for notifying listeners when changes are made to EMF models. The triggering of the synchronization activity may be performed loosely or strictly. *Strict* synchronization requires all changes to models to be taken into account immediately (or in the next stable state). *Loose* synchronization makes no statement on when synchronization should occur. Loose synchronization is triggered by an external stimulus, e.g., user, timer, etc.

Step 2 requires a definition of consistency and a means to evaluate whether it is satisfied or not by the model states. This point is further elaborated in section 3.1. In considering step 3, the challenge is to find an appropriate model transformation approach for making inconsistent models consistent again. This point is elaborated in section 3.2.

3.1. Deciding When Models Are Inconsistent

To be able to decide whether a set of models is consistent or not, it is necessary to first understand what makes the models consistent and inconsistent with respect to each other. This requires an understanding of the semantics of each model and a definition of the relationships between them. Theoretically, the relationships between the elements of each model can be formally understood if the semantics of each model are translated into a common semantic domain, where equivalences/refinements can be mathematically verified or reasoned about [EK+01, FEL+98]. However, in practice, this step is usually performed informally, which is

probably due to the significant overhead of performing such a task formally and a lack of tool support. In both cases, it is necessary to clarify in which states models are consistent and in which states they are inconsistent. We refer to this (informal or formal) clarification as a consistency definition. This definition is the basis for a function that can be used to decide whether the models are consistent or not in a given (stable) system state.

Informally clarifying the consistency relationships between models can be performed in a number of ways and can also depend on the intention of the models' architect(s). For example, in some cases, models may simply contain a subset of common information, e.g., a UML interaction diagram contains message instances that correspond to the method names of the respective classes in a UML class diagram. In other cases, a correspondence has a less direct equivalence, e.g., the presence of an account in one model means there must be three corresponding contracts in another model. Furthermore, in other cases, it may be easier to define consistency in terms of negative conditions, e.g., the presence of a loan in one model means there must be no checking accounts in another model.

3.2. Updating Models So That They Are Consistent

Once inconsistencies are detected, the necessary changes to the models need to be made to make them consistent again, if possible. This problem has been well studied in the field of software engineering and information systems and is often referred to as inconsistency management [SZ01]. In general, since elements can be added, modified, and deleted in various models, different strategies can exist for reconciling models, according to the consistency definition; such strategies are also known as inconsistency handling policies.

The main challenge that is faced in reconciling models (i.e., making them consistent) is to ensure that all possible situations that could arise have been covered and that the effect of updating models in each case leads to a stable and consistent state. To illustrate this problem, Figure 2 revisits the *SquashClassHierarchyTrans* transformation, which is used as the context for round-tripping between a class hierarchy and a single compact class.

Figure 2a shows the initial state (before reconciliation is attempted): the LHS model has 3 classes, each with one attribute, and the RHS model is empty. Figure 2b shows the desired result of reconciling the two models. Note that the strategy of reconciliation used here implies that adding new elements is preferred over deleting existing elements for reconciling the models. In this case, it makes no sense to delete elements that have just been created to find a consistent state, even if two empty models are nevertheless consistent. Figure 2c shows the result of the reconciliation activity, which was triggered by the deletion of the attribute *b* from the RHS model. It also highlights how reconciliation makes use of the trace information to ensure that no information is "clobbered" on the return trip. Figure 2d shows the result of the reconciliation activity, which was triggered by the addition of the attribute *d* to the RHS model. In this case, no assistance can be derived from the trace information, so the attribute *d* is added to the root class of the hierarchy. We suppose that this type of decision is usually based on some default behavior or user input. Note that the decision of where to place it has no impact on the consistency of the two models. In other words, it would have been equally valid to place the attribute in class *B* or *C* and the models would still be consistent. Finally, Figure 2e shows the result of the reconciliation activity, which was triggered by the addition of the attribute *e* to the LHS model, where the new attribute *e* has been added to class *C*.

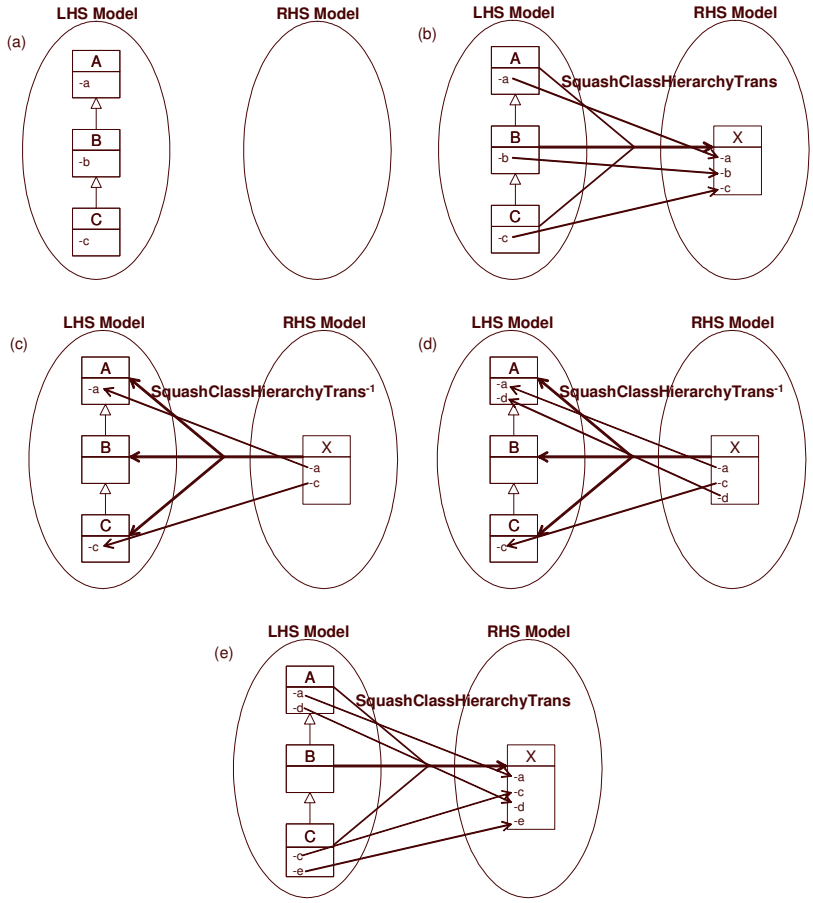

Figure 2: Mapping a class hierarchy to a single "compact" class.
(a) shows the initial state of LHS and RHS models (before they are reconciled);
(b) shows the result of reconciling the LHS model to an empty RHS model;
(c) shows the result of reconciling the models after attribute *b* has been removed from RHS model;
(d) shows the result of reconciling the models after adding attribute *d* to RHS model;
(e) shows the result of reconciling the models after adding attribute *e* to LHS model.

3.3. Composing Round-Trip Engineering Activities

Round-trip engineering activities may not be isolated; different round-trip engineering activities may overlap in domains. In fact, it is likely that given a set of models that need to be kept consistent, it is easier to establish multiple round-trip engineering activities between subsets of the models, rather than a single round-trip engineering activity over all models. For instance, it may be easier to express, and more intuitive to understand, bi-directional relationships, compared to multi-directional ones, i.e., with a cardinality greater than two. This notion of composition requires that round-

trip engineering activities that share domains do not have conflicting agendas. In other words, it is necessary to ensure that the combined constraints on each shared domain are satisfiable. In some cases, constraints on shared domains may not be contradictory per se, but they may nevertheless lead to unexpected or undesired behavior.

To illustrate such a situation, Figure 3 revisits the earlier *SquashClassHierarchyTrans* example of round-trip engineering and introduces another round-trip engineering activity (relating *Model2* and *Model3*), where the two activities have *Model2* in common. Consistency between *Model2* and *Model3* is defined in terms of a one-to-one correspondence between attributes of classes in *Model2* and attributes in a class composition hierarchy in *Model3*. In addition, a constraint is defined that asserts that any class in *Model3* cannot contain a public attribute (public visibility is diagrammatically shown with a prefixed '+'; '-' means private visibility). This constraint poses a problem for the scenario given in Figure 3, which we now describe.

Figure 3 shows the result of public attribute *e* being added to *Model1* and the update made by round-trip activity 1 to reconcile *Model2* to *Model1*. Up until this point, all attributes have private visibility. A problem arises, however, for round-trip activity 2 because it cannot reconcile *Model3* to *Model2* by simply adding the attribute to one of the classes *Model3*, due to the constraint stated earlier. Hence, round-trip activity 2 would need to delete attribute *e* from class *X* in *Model2* to make the models consistent again. So if round-trip activity 2 does delete public attribute *e* from class *X*, the question arises as to how round-trip activity 1 will react. Round-trip activity 1 has two possible paths, either it can reconcile *Model1* and *Model2* by deleting attribute *e* from *Model1*, or it can add attribute *e* back to *Model2*. The first scenario will lead to a stable state, but probably a confused developer, because after s/he adds attribute *e* to *Model1*, it will be subsequently removed (hence the confusion for the developer). The second scenario may lead to an infinitely changing state where round-trip activity 1 and 2 play against each other by continually adding and removing attribute *e* to *Model2*—the state will only become stable if attribute *e* is removed from *Model1*.

In general, it is important to note that inconsistencies between models may be necessary, if a number of intermediate steps may be required to reach the desired, consistent state. In such cases, the boundary of a stable state needs to be defined, together with a loose synchronization policy.

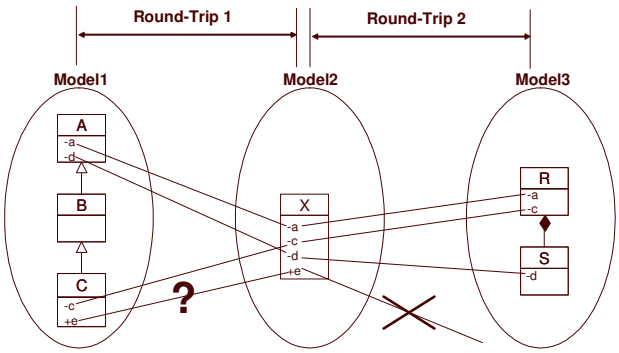

Figure 3: Two round-trip engineering activities that have a common domain (Model2): it shows the result of reconciliation according to round-trip activity 1, after an attribute was added to Model1, which breaks the consistency relationship between Model2 and Model3 according to round-trip activity 2.

In the context of UML with all its nine diagram types, it would seem that there is quite some scope for round-trip engineering between the different diagram types. However, because UML is defined by a single metamodel, the problem of keeping each view consistent with all the others can be moved to the problem of round-tripping each view to an amalgamated model, which is an instance of the whole UML metamodel. This situation is illustrated in Figure 4, where each view (e.g., Class $View_1$) takes its diagram type's perspective of the current system, and the amalgamated system model is the composite of all the views. Having the amalgamated model also means that there need only be a single round-trip engineering relationship with other artifacts, e.g., programming language code. As an aside, the internal consistency of the amalgamated model will still need to be checked and reconciled to be well-formed, which in itself is a very complicated activity; this is represented by the validation arrow in Figure 4. The validation activity would preferably allow inconsistent intermediate states, until a final consistent state is reached, at which time it would resolve the inconsistencies.

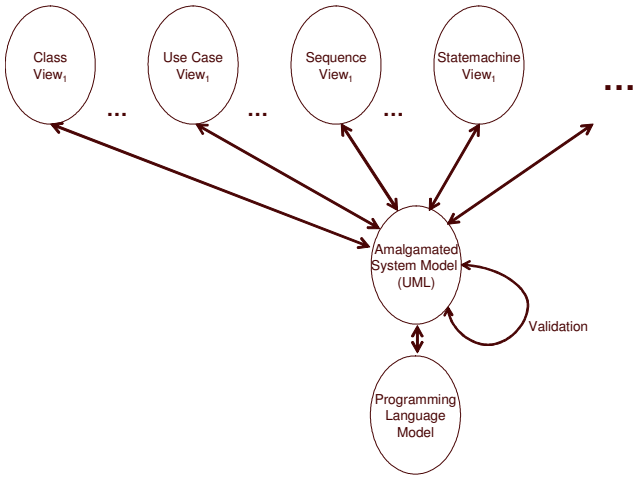

Figure 4: Round-trip engineering UML models.

4. Desirable Qualities of Round-Trip Engineering Approaches

In this section, we present and motivate a number of qualities that we believe are desirable for model round-trip engineering approaches to posses, drawing some conclusions from earlier sections. We also discuss each point, highlighting some approaches/techniques that could be used to address them.

4.1. Ability to Manage Trace Information

In section 2, we motivated the need for keeping trace information in round-trip engineering, and highlighted that the ability to trace back to existing model elements is one of its main differences to forward and reverse engineering (see Figure 1). Trace

information is needed if information is lost in the transformation from one model to the other. In other words, trace information is needed whenever the relationship between elements in the respective models is not bijective (i.e., not one-to-one and onto). As such, obtaining and managing trace information could be seen as a requirement for round-trip engineering, and not just a desirable quality.

Two approaches are commonly used in capturing trace information: 1) augment the models with the additional information needed for the return trip using auxiliary features [LB03], such as, comments; or 2) store the trace information externally so that the inverse transformation can use it on the return trip (called a reconstruction model in [Kue04a]). Option 1 is a common approach in round-tripping UML class diagrams to programming language code, where information about associations, etc. are stored in code comments. One problem that arises with using code comments is that as soon as the comments are changed/removed, the trace information becomes corrupted and the return trip can no longer make use of it. Also, adding extra information to artifacts can make them less aesthetically pleasing.

Option 2 remedies both these problems, so long as the trace information is kept up-to-date with any changes made to the models; this is because the trace information cannot be indirectly changed, i.e., by changing the artifacts, and the artifacts do not contain the trace information themselves, respectively. However, one problem that can still arise with both options is that different tools could define their own proprietary trace structures and hence interoperability between tools would not be possible. OMG's emerging MOF 2.0 standard will include a Query/View/Transformation (QVT) specification [Omg02a, QMG04]. This upcoming specification is moving towards standardizing trace information for model transformation, which is an important step forward for tools that will support round-trip engineering.

4.2. Intuitive and Concise Approach

When specifying round-trip engineering activities, there are a number of capabilities that must be taken into consideration in judging the intuitiveness and conciseness of the approach:

- the ability to precisely define the meaning of consistency between models;
- the ability to correctly decide when models become inconsistent;
- the ability to correctly reconcile models, according to the consistency definition; and
- the ability to connect the above three abilities in a seamless and useable way.

Ideally, a single specification could be used for all three tasks, i.e., consistency definition, deciding consistency, and model reconciliation. However, care needs to be taken that the three different usage requirements do not result in idiosyncrasies of the language, which could lead to difficulties in comprehension and expression for users.

With the task of model reconciliation, it is important that each case of model evolution can be clearly specified and understood in terms of the way that the reconciliation activity will react (see section 3). For example, how should the reconciliation activity react to the creation of new elements, modification and deletion of existing elements that are under trace relationships and not, etc.

4.3. Assistance When Multiple Solutions Are Possible

In Figure 2d, we saw an example of the reconciliation activity making a choice between multiple possibilities. In this case, a number of solutions are possible to correctly reconcile the models, but only one may be taken. One approach would be to let the reconciliation activity make a random choice. However, such an approach usually makes it more difficult for users to understand how the reconciliation activity will react in a given situation and it can make testing unnecessarily difficult. Also, in many cases, there may be one solution that is preferred over the others. As such, it should be possible to define a policy that is used to decide which choice should be taken when many solutions are possible. Such a policy in its simplest form may be nothing more than a default value or default configuration, or soliciting input from the user.

4.4. Assistance with Detecting Conflicts between Round-Trip Engineering Activities

As we described in section 3, it may be more straightforward to break up the round-trip engineering of a set of models into a set of round-trip engineering activities between subsets of the models in the original set. Figure 3 however illustrated a problem that can arise when composing round-trip engineering activities. The problem comes about because different reconciliation activities that share domains can have conflicting agendas. Thus, it would be valuable to be able to detect such conflicts so that they can be found before the system is deployed. The ability to decide if there are conflicts between the agendas of reconciliation activities will depend on the language used to express the agendas. In many cases, it may not be decidable, requiring some form of heuristic-driven approach instead.

5. Related Work

In this section, we survey some of the work that is related to round-trip engineering and inconsistency management.

As round-trip engineering integrates technology from different areas of software engineering, there exists a large amount of related work. As already mentioned, inconsistency management [SZ03] is a technique for making models consistent again. Inconsistency management has been studied for viewpoint-oriented approaches within software engineering, giving rise to a number of different approaches. In particular, approaches studying consistency management for UML models (see e. g. [Kue04b] and [EK+01]) can be seen as one foundation for a solid definition of round-trip engineering. We already pointed out that a basis of round-trip engineering is a clear definition of required consistency between the models which is usually one purpose of consistency management. Concerning UML models, this task is complicated by the absence of a formal semantics and a common development process. The methodology presented in [EK+01] and elaborated in [Kue04b] can be applied to define consistency for a given set of UML models. On the basis of that, round-trip engineering can then provide the necessary technology for enabling the modeler to move freely between different models.

Another set of related techniques to round-trip engineering can be seen in approaches that deal with forward engineering, e.g., generating Java programs from UML models. Whereas current tool support is mainly restricted to structural models,

there exist also approaches for including behavioral models. Engels et al. describe an approach for generating Java code from UML collaboration diagrams. Already there, it could be seen that complete forward engineering is a challenging problem, mainly due to the same reasons we already discussed above (lack of consistency definition and development process).

Concerning round-trip engineering in particular, there are several existing approaches. Assmann [Ass03] introduces mathematical definitions for round-trip engineering and requires for a so-called automatic round-trip engineering system, where the inverse function can be computed automatically. The Fujaba System [NNW+00] supports round-tripping for class diagrams and realizes this by extracting a syntax graph from the Java program. This syntax graph is then annotated for the generation of associations in class diagrams and also control flows in so-called story diagrams. This system is a good example of how round-tripping is currently realized in case tools and why it also involves a certain overhead: Without a technology to specify model transformations at a higher level of abstraction, the realization of round-trip engineering depends on encoding many different transformations directly on the syntax tree.

The CODEX system [LB03] aims at keeping a model consistent with a set of views on the model. Transformations from the model to the views must be specified manually (by graph transformation rules) but the CODEX system then automatically computes the required inverse transformations. One key idea in the system is to introduce so-called tags to remember where transformation rules have been applied and then use these tags for the inverse transformation rules. One restriction of the CODEX approach is that models can only be updated in a syntax-directed way which is induced by the transformation rules.

6. Summary and Future Work

Model round-trip engineering will be a key factor for many next generation Model-Driven Software Development approaches, because it will enable modelers to move freely between different representations of the system under discussion. In this paper, we have clarified a number of the issues that need to be addressed in automating round-trip engineering. We first explained that there is a difference between round-trip engineering, on one side, and forward and reverse engineering, on the other side, pointing out that round-trip engineering is even more challenging. We then clarified that round-trip engineering must be based on a clear definition of consistency of models, which is used to locate potential inconsistencies. Once these have been detected, round-trip engineering could make use of different model reconciliation strategies to make the model consistent. We further addressed composing round-trip engineering activities, highlighting the problem of conflicting agendas.

On the basis of this discussion, we proposed a number of qualities that we believe are desirable for model round-trip engineering approaches to posses, and we suggested, in some cases, possible directions toward solutions. Clarifying this domain is an important first step towards being able to systematically automate round-trip engineering of models, at least in specific cases.

As part of future work, we are interested in better understanding and enumerating different reconciliation strategies. We also plan on looking into improving current model transformation approaches for expressing model reconciliation. Furthermore, we intend to investigate how we could integrate the tasks of defining consistency,

decide consistency, and resolve consistency into a single homogenous, but accessible approach.

Acknowledgements

The authors would like to thank Rainer Hauser and, in particularly, James Rumbaugh for their very helpful comments and suggestions for this paper.

References

[Ass03] U. Assmann; "Automatic Roundtrip Engineering". ENTCS 82, No. 5, 2003.

[Bor04] Borland; "Together Control Center", 2004. http://www.borland.com/together/controlcenter/index.html

[BSM+03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose; "Eclipse Modeling Framework". Addison-Wesley Professional, 2003.

[CH03] K. Czarnecki, S. Helsen; "Classification of Model Transformation Approaches". Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, USA, 2003.

[EK+01] G. Engels, J. Kuester, L. Groenewegen, R. Heckel; "A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models". V. Gruhn (ed.): Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), ACM Press, Vienna, Austria, September 2001, pp.186-195.

[FEL+98] R. France, A. Evans, K. Lano, B. Rumpe; "Developing the UML as a formal modeling notation". Computer Standards and Interfaces: Special Issues on Formal Development Techniques, 1998.

[Gen04] Gentleware; "Poseidon Professional Edition", 2004. http://www.gentleware.com/products/descriptions/pe.php4

[Kue04a] J. Kuester; "Towards Inconsistency Handling of Object-Oriented Behavioral Models". Proceedings International Workshop on Graph Transformation (GT-VMT'04), Barcelona, Spain, March 2004.

[Kue04b] J. Kuester; "Consistency Management of Object-Oriented Behavioral Models". PhD Thesis, University of Paderborn, March 2004.

[KW03] A. Kleppe, J. Warmer; "Do MDA Transformations Preserve Meaning? An investigation into preserving semantics". MDA Workshop, York, UK, November 2003.

[KWB03] A. Kleppe, J. Warmer, W. Bast; "MDA Explained: The Model Driven Architecture–Practice and Promise". Addison-Wesley, 2003.

[LB03] H. Larrson, K. Burbeck; "CODEX – An Automatic Model View Controller Engineering System". Proceedings of Workshop Model Driven Architecture, Foundations and Applications, CTIT Technical Report TR-CTIT—03-27, University of Twente, 2003.

[NNW+00] U. Nickel, J. Niere, J. Wadsack, A. Zündorf; "Roundtrip Engineering with FUJABA". Proceedings of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany (J. Ebert, B. Kullbach, and F. Lehner, eds.), Fachberichte Informatik, Universität Koblenz-Landau, August 2000.

[OD04] Objects by Design; "UML Modeling Tools", 2004. http://www.objectsbydesign.com/tools/umltools_byCompany.html

[Omg02a] Object Management Group; "Request for Proposal: MOF 2.0 Query / Views / Transformations", 2002. http://www.omg.org/docs/ad/02-04-10.pdf

[Omg02b] Object Management Group; "Meta-Object Facility Specification". Version 1.4. http://www.omg.org/cgi-bin/doc?formal/2002-04-03

[Omg03a] Object Management Group; "MDA Guide Version 1.0.1". 2003.

[Omg03b] Object Management Group; Unified Modeling Language Superstructure Specification, version 2.0. Final Adopted Specification ptc/03-08-02, 2003.

[QMG04] QVT Merge Group; Revised submission for MOF 2.0 query / views / transformations RFP. OMG document ad/04-04-01, 2004.

[Sen03a] S. Sendall; "Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance?" OOPSLA '03 Workshop "Generative techniques in the context of MDA", 2003.

[Sen03b] S. Sendall"; "Source Element Selection in Model Transformation". UML '03 Workshop in Software Model Engineering (WiSME). Proceedings published in Technical Report, University of Bremen, 2003.

[SK03] S. Sendall, W. Kozaczynski; "Model Transformation - the Heart and Soul of Model-Driven Software Development". IEEE Software, vol. 20, no. 5, September/October 2003, pp. 42-45.

[SZ01] G. Spanoudakis, A. Zisman. "Inconsistency Management in Software Engineering: Survey and Open Research Issues". Handbook of Software Engineering and Knowledge Engineering, (eds.) S.K. Chang, World Scientific Publishing Co., 2001.

[Xde04] IBM Rational; "Rational Rose XDE Developer", 2004. http://www-306.ibm.com/software/awdtools/developer/rosexde/