Change Rule Execution Scheduling in Incremental Roundtrip Engineering Chain: From Model-to-Code and Back

Van Cam Pham, Ansgar Radermacher, Sébastien Gérard and Florian Noyrit
CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, P.C. 174, Gif-sur-Yvette, 91191, France

Keywords: Code Generation, Change Rules, Incremental, Model Transformation, Model Driven Engineering, EMF-IncQuery, AST.

Abstract: Model driven engineering allows many stakeholders to contribute their expertise to the system description. This practice enables agility but implies consistency maintenance issues between different system models. Incremental model transformations (IMT) are used to synchronize different artifacts contributed by the stakeholders. IMTs detect changes on the source model and execute change rules to propagate updates to the target model. However, the execution of change rules is not straightforward. A rule is only correctly executed if its precondition is satisfied at execution time. The precondition checks the availability of certain source and target elements involved in the rule. If a rule is executed when the precondition is false, either the execution is blocked or stopped. Therefore, the produced target model becomes incorrect. This paper presents two approaches to the scheduling of change rule execution in incremental model transformations. These approaches are also applied to the case of model and code synchronization and implemented in a tool named IncRoundtrip that transforms and generates code for distributed systems. We also compare the runtime execution performance of different incremental approaches with batch transformation and evaluate their correctness.

1 INTRODUCTION

The use of Model Driven Engineering (MDE) has increased in industries to gain quality and productivity (Mussbacher et al., 2014). In MDE, the process of developing a system most often relies on chaining transformations from source models at high level abstraction to target models and finally to code. Those two techniques are identified as model to model (M2M) and model to code (M2C) transformations. However, the development of complex system hardly follows a purely linear development process. Indeed, in modern developments many stakeholders contribute their expertise to the system description in an iterative way. This practice enables agility but implies consistency maintenance issues. In order to solve maintenance issues, incremental model transformations (IMT) are proposed.

IMTs consist of three phases: source model change detection, impact analysis, and change propagation (Kusel et al., 2013). The first phase detects which elements are added-to, modified-in or deleted-from the source model. The impact analysis maps these changes to change rules and the last phase executes the change rules to update the target model.

Changes are often captured by mechanisms of the modeling environment in transactions. In case many changes are detected, many associated change rules need to be executed to produce a correct target model. However, change rules cannot simply be executed in the same order as the associated modifications. The execution of a rule usually uses not only changed elements in the source model but also certain other source and target elements. In on-demand transformation mode, the changed source elements and certain other source elements used by a rule are available in the source model but in runtime mode, the source elements involved in the rule execution may not appear at the change time and added later by other model manipulations. Moreover, the target elements involved in the execution of a rule might be missing at the change time. If a rule is executed at the time the involved elements have not appeared yet in the source or the target model, either the execution is blocked or stopped, thus the produced target model is not correct. In that way, a wrong change rule execution order even leads the violation of correctness and consistency of transformations. Therefore, the transformation engine must determine which rule to be fired first.

Pham, V., Radermacher, A., Gérard, S. and Noyrit, F.
Change Rule Execution Scheduling in Incremental Roundtrip Engineering Chain: From Model-to-Code and Back.
DOI: 10.5220/0005687702250232
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 225-232
ISBN: 978-989-758-168-7

This paper presents two approaches named target-independence and target-dependence, respectively, dedicated to the scheduling of change rule execution in incremental model transformations. In these approaches, we specify that a rule is successfully executed if all its required elements are available. These required elements can be in the source or target models. These elements play the role of a precondition of a change rule. The precondition is explicitly expressed in change rules. Once the precondition is provided, it is verified at runtime before firing the associated rule. These approaches are put in practice by implementing a tool named IncRoundTrip that supports incremental JAVA round-trip engineering and deployment from UML models. We compare the runtime execution performances of the round-trip chain between the two approaches and a batch transformation.

The remainder of the paper is structured as follows: Section 2 shows a motivating example. Section 3 presents works related to incremental model transformation. Section 4 and 5 focus on our approaches and round-trip engineering from models to code, respectively. Section 6 tests the approaches by means of a proof of concept implementation of a tool named IncRoundtrip. We conclude and plan future works in Section 7.

2 MOTIVATING EXAMPLE

Our motivating example transforms a UML connector into an interaction component (Radermacher et al., 2009). This M2M transformation exists in the deployment tool Qompass Designer (Qompass, 2015), but is currently not implemented in an incremental way. As in Figure 1, the transformation consists of two rules of copying classes and properties from a UML source model and reifying UML connector to an interaction component, respectively. In a batch transformation, the transformation engine orderly visits System, client, server and eventually connector to transform them to the target model.

In runtime incremental mode, the model evolution process adds System, client, server and connector to the source model. To propagate these changes, the copying rule and the reification rule are executed three times and one, respectively. There are many ordering options to execute these rules. If the transformation engine chooses copying System first, then client, server, and finally reifying the connector, the target model is correctly produced.

If the reification rule is fired first instead, the runtime execution of the rule will not find the properties client and server in the target model. The engine

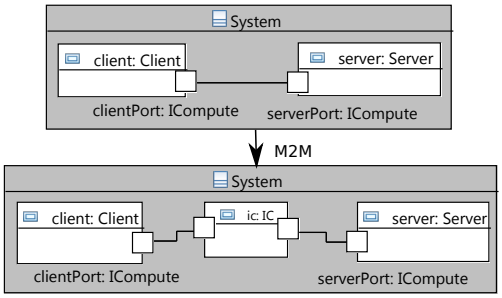

Figure 1: Transformation example from a UML Connector into an interaction component.

therefore either stops or blocks the execution of the rule. Thus the interaction component ic and the new connectors are not created in the target model. In this way, the resulting target model is not the same as that of the batch transformation.

Optionally, the transformation engine can execute change rules in the same order as the original modifications. In this example, if System, client, server and connector are orderly added to the source model, the transformation engine copies the class and the two parts before reifying the connector to produce the correct model. However, if the connector is added by model manipulations before the two parts are added, the reification rule is fired before. Therefore, the reification rule is not finished since the parts in the target model involved in the execution are missing. This order creates an incorrect target model.

3 RELATED WORK

This section presents different approaches related to incremental model synchronizations and round-trip engineering of models and code.

Triple graph grammar (TGG) (Kindler and Wagner, 2007)(Giese and Wagner, 2006)(Hermann et al., 2013) originates from graph theories. A graph transformation rule consists of two sides: a left- and a right-hand side. TGG performs local manipulations on graph models by finding a match of its left hand side graph pattern in the model and replacing it with the right hand side graph. TGG can be used for bidirectional transformations (Lauder et al., 2012) but TGG assumes that relationships between source and target model elements are bijective. Specifically, in our case, a connector is transformed in two different mappings.

QVT-R (OMG, 2008) standardized by OMG supports pre-conditions to schedule transformation rules. However, up to now, there is no implementation that supports full features of QVT-R.

Change driven model transformation (Bergmann et al., 2011) is the approach that we base on. Differently from other approaches, this approach considers changes as first-classes and part of transformation rules. In order to capture changes on a source model, uncontrolled model changes caught by mechanisms of the modeling environment in transactions are observed and evaluated. An incremental pattern matching technique is used in the evaluation to detect changes and send notifications to an event-driven mechanism. The latter then maps changes to change rules to propagate updates to the target model. Our approach focuses on the change detection and impact analysis phases which are parts of the model and code synchronization mechanism. Moreover, we support a batch transformation for the first time (the target model does not exist yet) since batch transformations are faster than IMTs in deriving the non-existing target model. Further modifications of the source model are transformed by IMTs. Traceability links created by the batch transformation are interrogated and used by IMTs.

Round-trip engineering of models and code is supported by several tools and approaches. RTET in (Nagowah et al., 2013) generates a working version of a tiered application with a JSP presentation, EJB manager classes with functions in JAVA, with an appropriate database model and Model/View/Control Web application. AndroMDA (AndroMDA, 2015) is a code generation framework that follows the Model Driven Architecture (MDA) paradigm. AndroMDA can generate deployable Java EE applications from a UML model using Hibernate, EJB, Spring and Structs frameworks. However, AndroMDA does not support reverse engineering. ArgoUML (Tigris, 2015) is an open source modeling tool that supports code generation and reverse engineering for skeleton code of C++ and JAVA. However, ArgoUML does not provide a round-trip engineering for C++/JAVA. Reverse engineering approaches in (Tonella, 2005) and (Xin and Xiaojie, 2007) do not provide the co-evolution between source code and models.

Our IncRoundTrip implementation is an open source tool. IncRoundTrip offers not only a (partially) incremental round-trip generation of JAVA and C++ from UML but also instant incremental model synchronization.

4 INCREMENTAL MODEL TRANSFORMATION

This section describes our incremental model synchronization that is a part of the round-trip engineering chain. Our incremental synchronization of models consists of forward and backward transformations. As in the introduction section, an IMT consists of three phases: change detection, impact analysis and change propagation. Change detection is executed by a mechanism that listens to and receives elementary notifications from source models. Changes on the source model are usually not related to the target model but the changed elements are linked to the associated target elements in the impact analysis phase to check the availability of required elements and order the change rule execution. We refer this approach as *target-independence* since the changes on the source model are detected without considering the target model. The second one as *target-dependence* considers the traceability links between the changed elements of the source model and the associated elements in the target model.

4.1 Target-independence

For the first approach, as in Figure 2, a mechanism as a query engine interrogates only the source model separately from the target model. When the source model changes, the mechanism receives these changes and recognizes change pattern matches. The query engine then sends a list of pattern matches to the Analyzer which then maps these matches to corresponding change rules. A precondition is passed along with a change rule in a form similar to "*when*" of QVT-R. It, as previously mentioned, consists of one or several existing model elements required by its corresponding change rule and is examined by the Analyzer. The Analyzer chooses one rule to analyze its precondition. It retrieves target elements transformed from source ones. If one of these target elements is not available in the target model, the rule is sequentially dependent on another rule and must be fired after the other rule. The Analyzer then puts the rule in a queue and continues with other rules. If all required elements of the rule are available, the rule is fired by the Rule Executor. When the execution of a rule is finished, the Analyzer reads a rule from the queue to iterate the process. Algorithm 1 shows how the Analyzer works.

Algorithm 1 shows that the availability of all required elements must be verified before firing a rule. However, the verification of the availability of required elements does not need to be done before the activation of rules. Indeed, the verification can be placed in rule execution. Therefore, a rule can be executed immediately after an associated change is detected. When the execution of the rule needs to retrieve a target element by a retrieving function, it

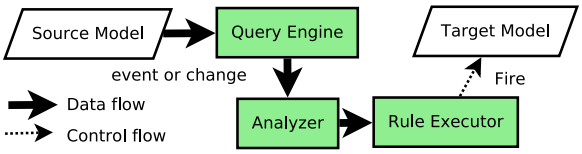

Figure 2: Target-independent change detection.

checks the availability of that element. If the element is available, it continues. Otherwise, either the rule is put into the queue or the execution waits in the retrieving function and the execution is resumed when the missing element is created in the target model. To do this, we define the retrieving function that manipulates traceability links between source and target model elements created in rule execution to get associated target elements from source elements. The retrieving function returns target elements or waits until these target elements are available. However, this option may create temporary inconsistencies in the produced model.

Algorithm 1: Change rule ordering by checking preconditions.

Precondition: Change rules (CRs) and their preconditions

```
1:  function ACTIVATE_RULES(CRs)
2:      RulesQueue ← getAllRules(CRs);
3:      while RulesQueue.notEmpty do
4:          r ← dequeue(RulesQueue)
5:          isReady ← areRequiredsAvailable(r)
6:          if isReady then
7:              execute(r)
8:          else
9:              enqueue(RulesQueue, r)
10:         end if
11:     end while
12: end function
```

In the previous example, the precondition of the copy rule is that the container of the copied element is mapped. Therefore, the container is put into the list of required elements in the precondition. The query engine gets a notification for each element (System, client, server and connector) added to the source model. Listing 1 (elements in blue are metamodel elements of UML) shows the query rule written in EMF-IncQuery (Ujhelyi et al., 2015) to recognize matches (the pattern matching is supported by several approaches as in (Fan et al., 2013) and (Bergmann et al., 2012)). A class, an attribute of a class or a connector of a class Class(element) are detected as added by Class.ownedAttribute(_clz, element), and Class.ownedConnector(_clz, element) respectively. These matches are sent to the Analyzer

```
pattern addedElement(ele) {
  Class(ele);
} or {
  Class.ownedAttribute(_clz, ele);
} or {
  Class.ownedConnector(_clz, ele);
}
```

Listing 1: EMF-IncQuery with the target-independent approach.

as events. If the copying rule corresponding to the added System is chosen first by the Analyzer, the latter checks that the execution of this rule does not involve other elements since System is the root element of the source model and does not have a container. The Analyzer therefore fires the rule by transferring the rule to the Rule Executor. The Analyzer then checks other rules. However, if the reifying rule is chosen before the copying rule, the Analyzer does not find the required elements System, client and server. This rule is added to the queue to be processed after these required elements are created. Similarly, the Analyzer then checks other rules and finds the copying rule for System. As a result, client and server are copied to the target model. The reifying rule is then executed with the availability of all required elements. Consequently, the produced target model becomes correct in any way of choosing rules to be analyzed.

4.2 Target-dependence

The target-dependent approach is based on the following principle: *a change is only detected if all of its required elements are available and the execution of a rule directly triggers other rules*. Required elements are verified in the change detection phase instead of the impact analysis phase. These required elements are expressed as constraints of patterns written in query languages such as EMF-IncQuery. To do it, the query engine must be able to interrogate both of the source and target models. As in Figure 3, this approach uses a traceability model between the source and target models. The traceability model refers to one or more source and target models and has explicit traces that refer to elements of the source and target models. The traceability metamodel is expressed in Figure 4. The source, traceability and target model are considered as one model. The query engine queries not only the source model but also the target and the traceability models. The principle of detecting changes is somewhat similar to that of Triple Graph Grammar. An element is detected as added to a source model if it is found in the source model but not in the traceability model. An element

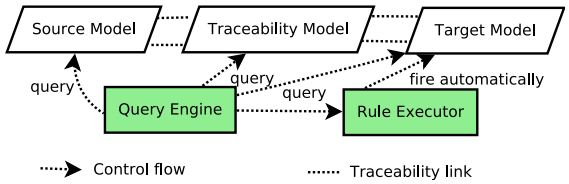

Figure 3: Target-dependent change detection.

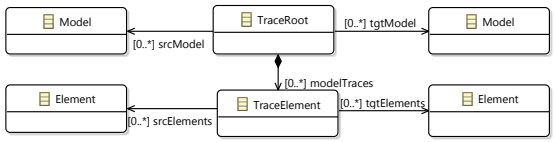

Figure 4: Traceability metamodel.

is considered as removed from the source model if either it is found in the traceability model but not in the source model or there are target elements in the traceability model but no corresponding source elements. When the source model changes, the pattern matcher of the query engine utilizes these links to collect all required elements of the rule. The rules are hence executed without referencing or checking any conditions.

In the previous example, several query rules are written as in Listing 2. "*neg*" is a negative application condition in graph pattern matching. neg find traceElement(ele, _tgt) queries the traceability model to detect that the element ele is not mapped or transformed to the target model yet. When the four elements from the example are added, only System is detected as added since the list of required elements of the rule is empty (System is the root element). After the execution of copying System is finished, client and server are detected as added since the container System of these parts as the precondition of the copying rules corresponding to these elements is available in the target model. These rules can be simultaneously executed. The finishing of copying the three previous elements prompts the query engine to detect the added connector. The Rule Executor fires then the reifying rule to finish the propagation.

4.3 Discussion

The target-independent approach has the advantage of simplicity and is useful for in-place transformations, since the source model is (trivially) always up to date and therefore required elements are always available. Nevertheless, it implicitly wastes time in the impact analysis within the Analyzer since many iterations of verifying required elements for every rule are needed before the activation of rules. These iterations even increase the execution time if many change rules need

```
pattern traceElement(src, tgt) {
 TraceRoot.modelTraces(_root, tr);
 TraceElement.srcElements(tr, src);
 TraceElement.tgtElements(tr, tgt);
}
pattern unmappedElement(ele) {
 Class(element);
 neg find traceElement(ele, _tgt);
} or {
 Class.ownedAttribute(clz, ele);
 find traceElement(clz, _tgtClz);
 neg find traceElement(ele, _tgt);
} or {
 Class.ownedConnector(ele);
 find traceElement(clz, _tgtClz);
 neg find traceElement(ele, _tgt);
 Class.ownedAttribute[0](clz, cli);
 Class.ownedAttribute[1](clz, srv);
 find traceElement(cli, _);
 find traceElement(srv, _);
}
```

Listing 2: EMF-IncQuery with the target-dependent approach.

to be fired (see Section 6 for the execution time comparison).

To improve execution time of the target-independent approach, the Analyzer can set a priority for each rule. A higher value would be assigned to the priority of the copying rule than that of the reifying rule. The engine can activate the rule with the highest priority without multiple iterations and the performance would be better. However, the priority assigning approach is not feasible since the priority settings of rules are manually written by transformation writers. Moreover, in the example above, System and its attributes are transformed by three instances of the same rule (the copying rule). If the Analyzer chooses an instance associated with one of the attributes to transform first, the execution of the rule instance would not find System in the target model. To resolve this, a dynamic priority scheduling that checks all rules to execute is necessary. In fact, this dynamic scheduling is similar to the above target-independence approach that checks all required elements of a rule to be executed.

In the target-dependent approach, two independent rules with required elements can be detected and executed in parallel mode. The impact analysis and firing of rules are effortless but the pattern matching is more complicated. The matcher does not only evaluate the source model (as the target-independent approach does) but also the target and traceability model.

By comparing the two approaches, there is a trade-off between the change detection and impact analysis.

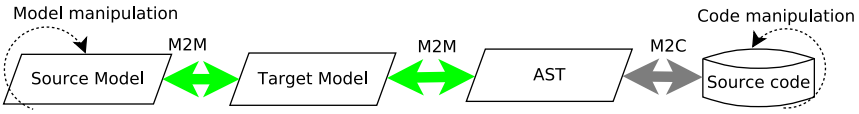

Figure 5: Incremental round-trip engineering chain with ASTs.

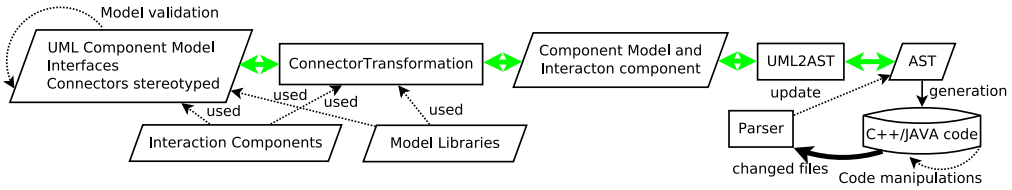

Figure 6: Model transformation chain in Qompass Designer.

If a smart pattern matching technique (such as EMF-IncQuery in the Eclipse Modeling Framework) is supported by the development environment, the target-dependent would be the choice. Contrarily, the target-independent would be more suitable for tools with poor graph techniques support. In the implementation, we compare the runtime execution performance of both approaches.

5 TARGET MODELS AND CODE SYNCHRONIZATION

This section presents a synchronization of code and target models derived from the above transformations. In order to reuse the transformation techniques discussed above, the models should be transformed into an intermediate representation that reflects code. We transform the models into an Abstract Syntax Tree (AST). Code is generated from the AST by an unparsing process. The synchronization from code to ASTs is a parsing process supported by many development environments. By this way, model synchronization from target models to AST is needed (as shown in Figure 5). Using an intermediate representation as ASTs increases the modularity of the chain.

In the model evolution process, the incremental transformation updates the corresponding AST nodes from the changed model elements. The updated AST nodes are used to deduce changed compilation units that define a set of modeling elements used for modeling source files. The code generator then uses templates, which define the generation of AST compilation units to code, to re-generate the files associated with the changed compilation units. In fact, the number of compilation units deduced from the updated AST nodes might be high, as shown in the renaming case below. In order to reduce it, we categorize different cases in which a compilation unit should be re-generated to its corresponding source file as followings:

- If a model element is added, only its containing file on the code level is generated; if one is updated, two cases are distinguished: renaming and changing the element content.
- If an element is renamed, its corresponding compilation unit and all other compilation units that refer to (or use) it need to be re-generated since references in code are name-based.
- If the content of an element has changed, only re-generate its compilation unit; if one is deleted, delete its compilation units and re-generate the compilation units that refer to this element.

In fact, if a compilation unit refers to a deleted element, it will require changes since re-generation will not help resolve code errors (i.e., an attribute of a class typed with a deleted class). Thus, developers need to explicitly make changes to fix core errors either on the model level (re-type the attribute or delete the attribute) or the code level.

In the backward direction of the round-trip, when a code file is changed by developer operations such as adding or renaming classes, functions/methods or attributes, the file is parsed into an AST by an update of existing ASTs. By updating ASTs, the backward incremental transformation automatically propagates changes from code to the target models and eventually to the source models. We implemented both of the IMT approaches as presented above to propagate changes from code to models in the backward direction.

6 PROOF OF CONCEPT

As previously mentioned, Qompass Designer currently executes the connector transformation among other transformations in a batch mode. Qompass Designer also supports the deployment and generation

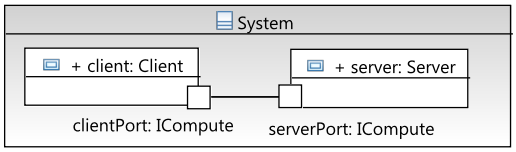

Figure 7: Application model for evaluation.

of C++ code for embedded distributed systems from UML models. In order to do so, it needs the information about which interaction components (IC) should be used and this information is provided by a stereotype from the Flex-eware compoment model FCM (Jan et al., 2012). To put our incremental approaches in pratice, we implement a tool named IncRoundTrip based on the existing batch transformation in Qompass Designer. The tool transforms a source UML model that contains FCM connectors to a target model containing ICs. The target model is then transformed into ASTs and eventually JAVA code. The implementation of an incremental C++ generator is in progress. The incremental chain is shown in Figure 6. To detect changes, we utilize the incremental graph pattern matching language EMF-IncQuery.

A UML component model firstly consists of components, interfaces, connectors and stereotypes applied to components used in the ConnectorTransformation phase. The component model can also import other UML model libraries that contain different data types or ICs (in model libraries as well). The semantics of model libraries is similar to source code libraries in traditional software development. Therefore, ICs that are used in the transformation can be defined in other model libraries independently from the application model. The ConnectorTransformation principally transforms UML connectors into ICs. The transformation is not trivial since ICs are adapted to the types used by a connection (types of connected ports). The details of the ConnectorTransformation are shown in (Radermacher et al., 2009) but are not in the scope of this paper. After transformation, code is generated from components to object-oriented code. The synchronization of ASTs and code is independent from model transformations.

In order to test the IMTs in practice, we implemented a batch transformation and the incremental approaches, executed them and compared the resulting models and code in all cases. The resulting models and code are the same. We also evaluated the performance of these implementations. We executed the chain on models with the number of nodes ranging from 2 to 20 nodes and the number of connectors with different interfaces from 1 to 10. A small application model is shown in Figure 7. We transform ICs for distributed applications that use socket connections

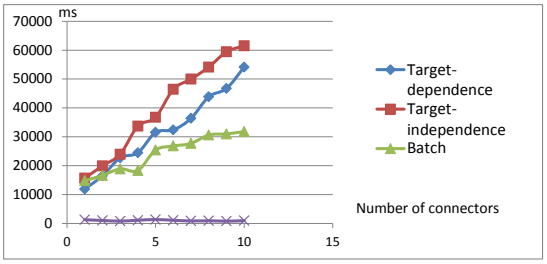

Figure 8: Performance comparison between the incremental transformation approaches and batch transformation.

to communicate. The number of generated classes ranges from 8 to 71. The performance comparison of the two approaches and the batch transformation is shown in Figure 8. As noted in the above sections, the target-dependent approach runs faster than the target-independent one. Moreover, the batch transformation outperforms the two others for the first time since the batch does not need to detect and analyze the impact of changes on the target model. The communication between components is established by the connector that connects through ports typing interfaces. After the first transformation, we add 5 UML classes (each one has 5 operations and 5 attributes), the transformation time in the target-dependent approach dramatically decreases as the Incremental line in Figure 8. The time for processing (transforming to the target model and eventually code) these 5 classes is 1s.

7 CONCLUSION

In the paper, we discussed two approaches of change rules scheduling in incremental model transformations. One approach iterates to check the availability of required elements of rules and the other one uses triggers and query languages. In the later, a change is only detected if its required elements are available. We applied the approaches to a specific round-trip engineering model transformation and code generation tool named IncRoundtrip in model- and component-based development. In the implementation, we detect changes by the incrementality of the EMF-IncQuery query language. We compared the runtime execution performance of both approaches and the batch transformation. The target-dependent approach outperforms the other one.

Models and code are synchronized in both directions. We used ASTs as intermediate models to reuse model transformation techniques. We synchronized the models and ASTs. The synchronization of ASTs and code are executed by the help of parsers.

For the moment, the model synchronization is done by two uni-directional transformations that are difficult to maintain and the synchronization from code to models is not fully implemented. For future works, we will apply the scheduling algorithms to the definition of a transformation language. Queries written in this language are then generated to EMF-IncQuery and Xtend code to provide fully automatically runtime incremental model synchronization. We will also complete the round-trip chain for C++ and JAVA.

ACKNOWLEDGEMENTS

The work presented in this paper is supported by the European project SafeAdapt, grant agreement No. 608945, see http://www.SafeAdapt.eu. The project deals with adaptive system with additional safety and real time constraints. The adaptation and safety aspects are stored in different models in order to achieve a separation of concerns. These models need to be synchronized.

REFERENCES

AndroMDA (2015). http://www.andromda.org/ [Online; accessed 01-Sept-2015].

Bergmann, G., Rth, I., Szab, T., Torrini, P., and Varr, D. (2012). Incremental pattern matching for the efficient computation of transitive closure. *Lecture Notes in Computer Science*, 7562 LNCS:386400.

Bergmann, G., Rth, I., Varr, G., and Varr, D. (2011). *Change-driven model transformations*, volume 11.

Fan, W., Wang, X., and Wu, Y. (2013). *Incremental graph pattern matching*, volume 38.

Giese, H. and Wagner, R. (2006). *Incremental Model Synchronization with Triple Graph Grammars*, page 543557.

Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., and Engel, T. (2013). Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software and Systems Modeling*, page 129.

Jan, M., Jouvray, C., Kordon, F., Kung, A., Lalande, J., Loiret, F., Navas, J., Pautet, L., Pulou, J., Radermacher, A., and Seinturier, L. (2012). Flex-eWare: A flexible model driven solution for designing and implementing embedded distributed systems. *Software - Practice and Experience*, 42:1467–1494.

Kindler, E. and Wagner, R. (2007). Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *Techn. Ber. tr-ri-07-284, University of.*

Kusel, A., Etzlstorfer, J., and Kapsammer, E. (2013). A survey on incremental model transformation approaches. *ME 2013* Models and Evolution Workshop, at MODELS'13.

Lauder, M., Anjorin, A., Varr, G., and Sch, A. (2012). Efficient model synchronization with precedence triple graph grammars.

Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-m., Cheng, B. H. C., Collet, P., Combemale, B., France, R. B., Heldal, R., Hill, J., Kienzle, J., and Schöttle, M. (2014). The Relevance of Model-Driven Engineering Thirty Years from Now. *ACM/IEEE 17th MODELS*, pages 183–200.

Nagowah, L., Goolfee, Z., and Bergue, C. (2013). RTET - A round trip engineering tool. In *ICoICT 2013*, pages 381–387.

OMG (2008). Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification.

Qompass (2015). https://wiki.eclipse.org/Papyrus_Qompass. [Online; accessed 01-Sept-2015].

Radermacher, A., Cuccuru, A., Gerard, S., and Terrier, F. (2009). Generating execution infrastructures for component-oriented specifications with a model driven toolchain: A case study for marte's gcm and real-time annotations. *SIGPLAN Not.*, 45(2):127–136.

Tigris (2015). Tigris. http://argouml.tigris.org/. [Online; accessed 01-Sept-2015].

Tonella, P. (2005). Reverse engineering of object oriented code. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*

Ujhelyi, Z., Bergmann, G., Hegedüs, A., Horváth, A., Izsó, B., Ráth, I., Szatmári, Z., and Varró, D. (2015). EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98(2015):80–99.

Xin, W. and Xiaojie, Y. (2007). Towards an AST-based approach to reverse engineering. In *Canadian Conference on Electrical and Computer Engineering*, pages 422–425.