2010 Seventh International Conference on the Quality of Information and Communications Technology

Increasing Quality in Scenario Modelling with Model-Driven Development

João Pedro Santos, Ana Moreira, João Araújo, Miguel Goulão
Departamento de Informática, CITI, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal
{joao.santos, amm, ja, miguel.goulao}@di.fct.unl.pt

Abstract— **Models, with different levels of detail, share similar abstractions that can be reused by means of model-driven techniques such as transformations. For example, scenarios are a well-known technique in requirements engineering to represent behavioral flows in a software system. When using UML, scenarios are typically represented with activity models in the early stages of software development, while sequence models are used to describe more detailed object interactions as modeling progresses. This paper defines transformation rules to automate the migration from activity to sequence models. We present a case study illustrating the application of our transformation rules. Our preliminary assessment of the impact of the benefits of using these transformations points to: (i) a reduction of around 50% in the effort building sequence models, (ii) increased traceability among models, and (iii) error prevention when migrating from different scenario notations.**

Keywords – Scenario Modeling, Model Transformations, Model-Driven Engineering

I. INTRODUCTION

Scenarios [1, 2] are widely used in Requirements Engineering (RE) to represent paths of possible behavior through a use case which are investigated to elaborate requirements. A scenario is *"a straight-line sequence of (possibly numbered, typically interactive) steps taken by independent-acting (presumably intelligent) agents playing (system) roles"* [1]. Scenarios specify system and user interactions, or use cases; they ensure that stakeholders share a sufficiently wide view of the system. Scenarios are applicable to all types of systems, at any stage of the development life cycle (thus, at different levels of abstraction). Approaches using UML [3] represent scenarios through activity [4] and sequence models [5]. While activity models are mostly used in the preliminary stages of analysis and design, sequence models tend to be used later, as the design progresses, where more detailed descriptions of object interactions become necessary.

Some behavioral and structural abstractions present in activity models can be reused automatically in sequence models by means of transformations. This is the fundamental motivation of this paper: to study how information contained in activity models can be systematically used for constructing sequence models, improving the process of moving from requirements to design. By using Model-Driven Engineering (MDE) [6-8] techniques, such as defining transformations between two kinds of models, it is possible to decrease the time costs on modeling scenarios, if the transformations used are correct and applicable to any problem domain. Additionally, we use MDE to support traceability between artifacts of different models.

The remaining of this paper is organized as follows. Section 2 describes transformation rules to map activity models into sequence models, addresses the refinement of the generated models and discusses traceability support. Section 3 introduces the supporting tool to implement the transformations defined in the previous section. Section 4 illustrates the application of our approach to an existing scenario of a case study and compares the costs of modelling the scenario by hand and by refinements of the generated model. Finally, Section 5 concludes the paper and provides directions for future work.

II. MIGRATING FROM ACTIVITY TO SEQUENCE MODELS

Now we describe both the transformation rules to generate sequence models from activity models and the refinements that can be applied to the generated model. First of all, we must guarantee that the activity models are deterministic. According to [3], *"the order in which guards are evaluated is undefined and the modeler should arrange that each token only be chosen to traverse one outgoing edge, otherwise there will be race conditions among the outgoing edges"*. This means that:

- **Guards should not overlap.** For example, guards such as $x < 0$, $x = 0$, and $x > 0$ are consistent whereas guards such as $x <= 0$ and $x >= 0$ overlap thus being inconsistent as it is not clear what should happen when $x = 0$;
- **Guards on decision points must form a complete set.** For example, guards such as $x < 0$ and $x > 0$ are not complete because it is not clear what happens when $x = 0$.

The following two subsections describe how activity model elements are mapped into sequence model elements in terms of transformation rules, and what kind of refinements can be made after the generation process.

A. Generating Sequence Models

This section lists transformation rules between activity and sequence abstractions. Each sequence model element type (*Object*, *Message*, *Operator*) is grouped into a main rule, making three in total. Each rule has sub-rules specifying how properties of that element type can be derived and in what contexts this element should be generated.

Rule 1: Generating Objects in Sequence Models

978-0-7695-4241-6/10 $26.00 © 2010 IEEE
DOI 10.1109/QUATIC.2010.36

IEEE computer society

Our proposal handles four types of objects [9]: *actor object*—external person or entity that interacts with the system; *boundary or interface object*—user interface elements such as screens, reports, HTML pages, or emails; *control object*—the glue between interface objects and entity objects, implementing the logic required to manage the various objects and their interactions; *entity object*—the information processed by the system and typically found at the database level as data.

Interface and control objects are created by default in sequence models with the name of the activity model that represents the scenario under study. In activity models, it is common to represent access operations (*read* or *write*) to objects. These operations are represented with flows between activities and objects. We map objects found on activity models to entity objects in the sequence model. Actor objects are generated based on swimlanes representing actors on the activity model.

Rule 2: Generating Messages in Sequence Models

Each *activity* in an activity model is mapped into a message in the sequence model. Complex and, therefore, decomposable messages can then refined into a set of messages. Our approach uses sub-rules to identify the source and target object of the generated messages, i.e., which object is the *caller* and which is the *callee*. Four sub-rules are required for this purpose.

Rule 2.1: Object flows

The direction of the flow which connects an activity to an object indicates if it is a *read* or a *write* operation. For *write* operations the direction of the flow is from the activity to the object. For *read* operations the direction is reverse. A *write* operation triggers the creation of a message from the control to the entity object with the name of the activity. For a *write* operation, a return message with type *void* from the entity to the control object is created. Figure 1 depicts this rule.

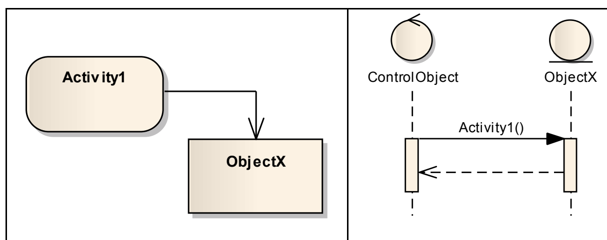

Figure 1 - Rule based on object flow denoting a write operation

Read operations require a return message with type ***not void*** is created from the control to the entity object. The name of the returned message is determined by the name of the returned object. For example, if a message *getX* is sent to an entity object, the return message to the control object is named *X*. The type of the return object is not generated and should be refined by the user. Figure 2 depicts this rule.

Rule 2.2: Message name

This is based on the names of the created messages. Some message names implicitly give information about the objects' type (interface, control, entity) the target of the message has. For example, *showMessage()* is typically sent to interface objects to display messages to the user. On the other hand, *interfaces* only receive messages from controls or actors. However, it is not common actors calling an interface's *showMessage().* The recurring pattern is that the message is sent from a control to an interface object.

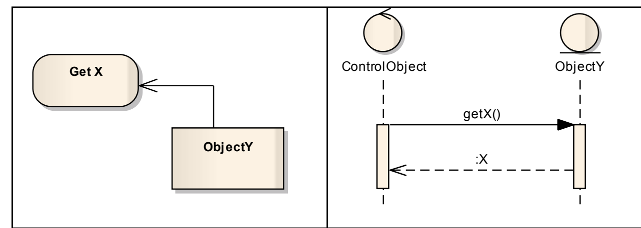

Figure 2 - Rule based on Object Flow denoting a read operation

Rule 2.3: Swimlanes

When a message is generated from an activity that is inside a swimlane representing an actor, the source object of that message is of type actor. As actors only access interfaces, the pattern is that the source and target of the message are the actor and interface objects, respectively. Figure 3 depicts this rule.

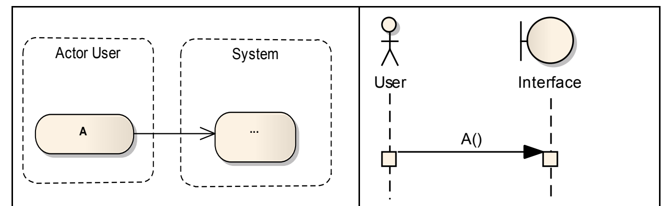

Figure 3 – Swimlanes representing actors (left); corresponding message (right)

Rule 2.4: Redirecting Messages

The main goal of interface objects is to redirect messages from actor to control objects, and vice-versa. Messages to achieve this goal are created automatically. When an actor calls an interface, the latter redirects the call to the control. Although this is not always desirable (e.g., user editing a form), in most cases it is. If this redirected message is not desired, the user needs to remove during the refinement phase. The reciprocal situation also happens, that is, when a control object makes a call to an interface object, the interface object redirects the call to the actor object.

Rule 3: Generating Sequence Model Operators

Sequence models may use several kinds of operators, such as: **ALT** (or alternative), where only one of multiple fragments satisfying the condition is executed; **PAR** (or parallel), where different fragments run in parallel; **OPT** (optional), where the fragment executes only if the condition is true; **LOOP**, where the fragment executes multiple times while the guard condition is true. Each of these fragments is generated by sub-rules.

Rule 3.1: Generating *PAR* Operators

A PAR operator is created in the sequence model when a pair of *fork-join* elements is in the activity model. The elements between *fork* and *join* are included in a *PAR* fragment.

Rule 3.2: Generating *ALT*, *OPT* and *LOOP* Operators

Decision nodes in an activity model require:

1. **Handling outgoing flows that form a cycle.** Algorithms for graphs with cycle detection mechanisms can be used to detect cycles in an activity model. Activity models can be viewed as graphs, where activities and flows between activities are seen as nodes and edges, respectively. For each cycle detected, a *LOOP* operator is created with a guard condition, respective messages and sub-operators.
2. **Handling the remaining outgoing flows.** If the number of output flows is 1, an *OPT* fragment is created with its guard condition. If the number of outgoing flows is greater than 1, a fragment *ALT* is created. Within this fragment, there should be an alternative for each outgoing flow with its guard condition. The elements inside the flow of each guard are moved to the respective fragment.

B. Refining Sequence Models

After generating the sequence model, the domain analyst must refine it. This is needed because sequence models are more fine-grained than activity models and, hence, additional information should be provided to the generated model. During this, the domain analyst should follow these typical refinements:

- **Add arguments and types (*string*, *int*, etc):** complete the message specification with arguments and their types. Specify the object type for the existing return messages.
- **Decompose a message to a set of messages:** as mentioned before, the behavior of an activity may be complex, and may be decomposed into several messages. Another solution is to decompose the complex activity using another activity model, for example, and regenerate the sequence model.
- **Add return messages:** for synchronous calls it is necessary to identify messages in which the related return message was not automatically generated. By default, the return messages are only created when *read* or *write* operations are identified.
- **Add variables:** sometimes, the result of an operation call needs to be saved in a variable, to be used later. This variable can be used, for example, as part of a guard condition.
- **Initialize guards:** in some situations, it is necessary to initialize the value of a generated guard so that it can be evaluated the first time it is used.
- **Delete undesired elements:** undesired sequence model elements can be generated by the transformation; these should be deleted by the domain analyst.

III. Tool Support

We implemented a plug-in for the Eclipse platform [10] to support the transformations described before, and used the Eclipse Modelling Framework (EMF) and UML2 plug-in for Eclipse[1]. EMF allows defining metamodels. It also has a code generation facility that helps manipulating and reading instances of metamodels. The UML2 plug-in is an EMF-based implementation of the UML2 metamodel for Eclipse and was used to access the metamodel and concrete syntax of activity models. Since sequence models are not implemented yet on the UML2 plug-in, we used EMF to specify our own metamodel for sequence models, created based on the UML2 infrastructure specification[2]. Finally, we used EMF generated Java code to read abstractions from activity models, process them, and create sequence model abstractions. One current limitation of our tool is that it does not support nested loops detection nor compound activities in activity models. The graphical user interface is very simple; the user only needs to right click in the source model (activity model) and select the option to convert into a sequence model. After this generation, the user can use the EMF environment to refine the sequence model.

To receive traceability information between activity and generated sequence model elements, a defined metamodel is used to link abstractions from the activity to sequence models (see Figure 4). A user can see how activity elements are related with sequence elements through navigation.

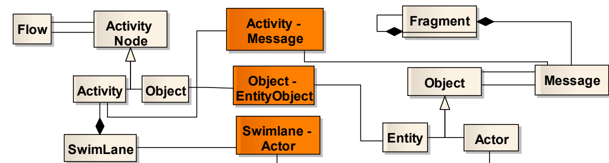

Figure 4 - Activities and Sequence Models Unified Metamodel

This metamodel is composed of activity (left) and sequence (right) model elements and then metaclasses are used to link activity to sequence abstractions (center). The central abstractions unify the concepts present on activity and sequence models and reflect the result of the transformation. The metamodel element with name Activity-Message allows preserving the connection between Activities and the sequence model messages that it generated. The element Object–Entity connects the objects found on the activity model and the generated entity object in the sequence model. Finally, the element Swimlane-Actor shows how swimlanes in the activity model were the source for the *actor* objects.

IV. Application to a Case Study

This section uses the case study *Mobile Media* [11] to illustrate the proposed approach. *Mobile Media* is a software system for mobile devices such as mobile phones, which manipulates photo, music and video on mobile devices. The user can manipulate data, such as adding and deleting media, configure a media file as a favorite, add or delete media albums. The user can also access the data on the device. The user can list albums, media, view the favorites media or eventually play a media file (play a video, see a photo or hear a sound). Finally, the user can share the media data with other mobile media users, by sending messages. These messages can be sent via an SMS or Email protocol. Due to space restrictions, we only present one of the scenarios - *send media via SMS* - of this case study.

[1] www.eclipse.org/uml2/

[2] http://www.omg.org/technology/documents/formal/uml.htm

A. Activity Model for "Send Media via SMS"

In *Send Media via SMS*, the user starts by selecting the Send Media via SMS option, then the system asks for media to send. The user selects the media to send in the message. Then, he specifies the target number of the message. This information is enough to send the message to the target mobile device (activity Send Message). If the message is sent without errors, it is saved locally in the Mobile Media system and "Message Sent Successfully" is shown to the user. Figure 5 depicts this scenario represented through an activity model.

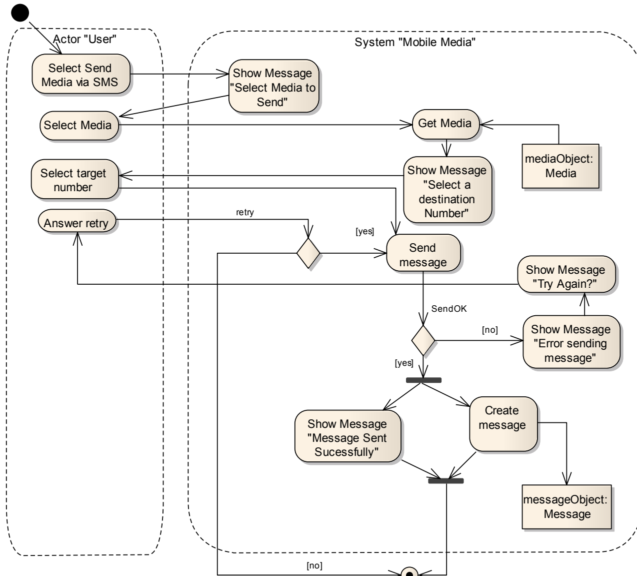

Figure 5 - Activity model for Send Media via SMS

B. Generation of Sequence Model for "Send Media via SMS"

By applying the rules discussed in Section II, we generate the sequence model depicted in Figure 6 for the scenario Send Media via SMS.

Table I shows how each numbered element was created and which transformation rule was used.

C. Refining the Sequence Model for "Send Media via SMS"

After the generation of the candidate sequence model, some refinements can be done to obtain a more complete sequence model. The following points show some of the possible refinements for this example:

- The message *selectMedia()* can be completed with an argument of type *String*, denoting the path of the selected media.
- The message *selectTargetNumber()* can be completed with an argument of type *integer*, denoting the destination number of the message.
- The message *sendMessage()* can be completed with two arguments: the path of the selected media and the destination number of the message. The return of that message should also be assigned to a variable *sendError* which will be evaluated on the *LOOP* operator.
- The variable *retry* of the loop fragment must be initialized to be evaluated on the first iteration of the loop. In this case, the value should be *retry = yes* in order to execute the loop the first time. The *answerRetry()* return value should also be assigned to the *retry* variable.

Figure 7 illustrates a refined version of the sequence model.

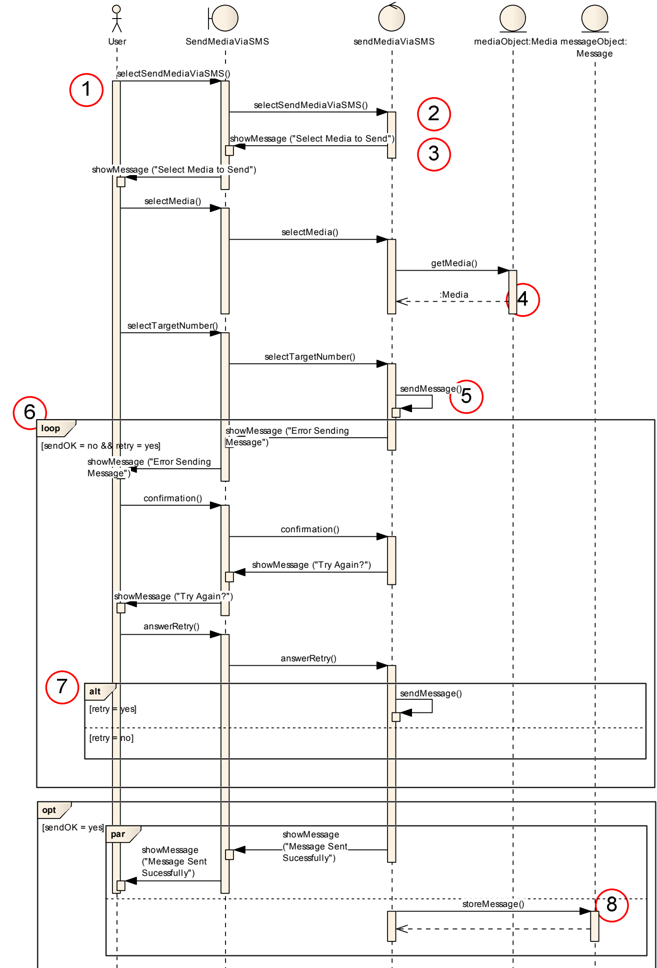

Figure 6 - Sequence model for the scenario Send Media Via SMS

TABLE I. RULES APPLIED FOR SEQUENCE MODEL GENERATION

Nr.	Rule Applied
1	**Rule 2.2.** This message was created from the activity *Select Send Media Via SMS*. The source of the message is the actor object, since it was the first generated message.
2	**Rule 2.4.** This message was created using the rule that redirects a message from the *actor* object to the *control* object.
3	**Rule 2.3.** This message has *interface* as the target object since the message name fits with the pattern *ShowMessage*
4	**Rule 2.1.** This message was created with name *Media* since the last created message denotes a read operation.
5	**Rule 2.2.** The name of this message was derived from an activity with the same name. This message has the *control* object as source and target, since no other rules were

	applicable in this situation.
6	**Rule 3.2.** This fragment was created as a loop was detected in a decision node with an outgoing flow with guard *[sendOK = no]*. Since the cycle includes also the outgoing flow with guard *[yes]* both conditions must be true to enter the loop fragment.
7	**Rule 3.2.** This fragment was created since a decision node with two outgoing flows and no loops were detected on the activity model.
8	**Rule 2.1.** This message was created with type *void* since the previously created message denotes a write operation.

D. Discussion

We have shown the application of a scenario of the Mobile Media case study with positive results. More scenarios have been developed and sequence models generated successfully. Statistical tests involving a total of 11 scenarios to quantitatively evaluate the gain were also applied (not shown here due to lack of space) with positive results.

In order to compare costs between creating a sequence model from scratch or use our approach and refine the generated model, we can associate each sequence model element action with a cost. If we consider that actions made in a sequence model such as (i) removal of any kind of element; (ii) insertion of a variable/argument name; (iii) insertion of a variable/argument type; (iv) insertion of an operator (PAR, ALT, etc) and respective guard conditions; (v) insertion of an object and its name; (vi) insertion of a message and the corresponding procedure call name (if necessary), have one unit of time cost (to simplify, we considered all types of action as having the same cost), differences in time costs can be calculated.

If we compare the time cost to create the sequence model presented previously from scratch (72 additions) and the cost needed for refinement over the generated model (30 additions + 2 removals) we can conclude that the cost has decreased from 72 to 32 units of time cost, a value that shows a significant improvement.

There is also a limitation with our approach, regarding reuse of refinements performed by the user when the sequence model is re-generated. The refinements done previously are currently lost and must be redone by the domain analyst. We are currently working to support reuse of refinement information as a future step.

It is also important to point that our approach is dependent on the quality of the activity diagrams, so poor activity diagrams will lead to poor sequence diagrams and more effort needed in the refinement stage.

V. RELATED WORK

Each UML model represents a particular aspect of a software system from a particular viewpoint. However, overlapping between the different models exists. This overlap can be used in the form of semi-automatic transformations between notations, to reduce the design time and help maintaining consistency between different models. The authors of [12-14] define transformations between different models or viewpoints in UML or other languages, aiming at automating part of the modeling process.

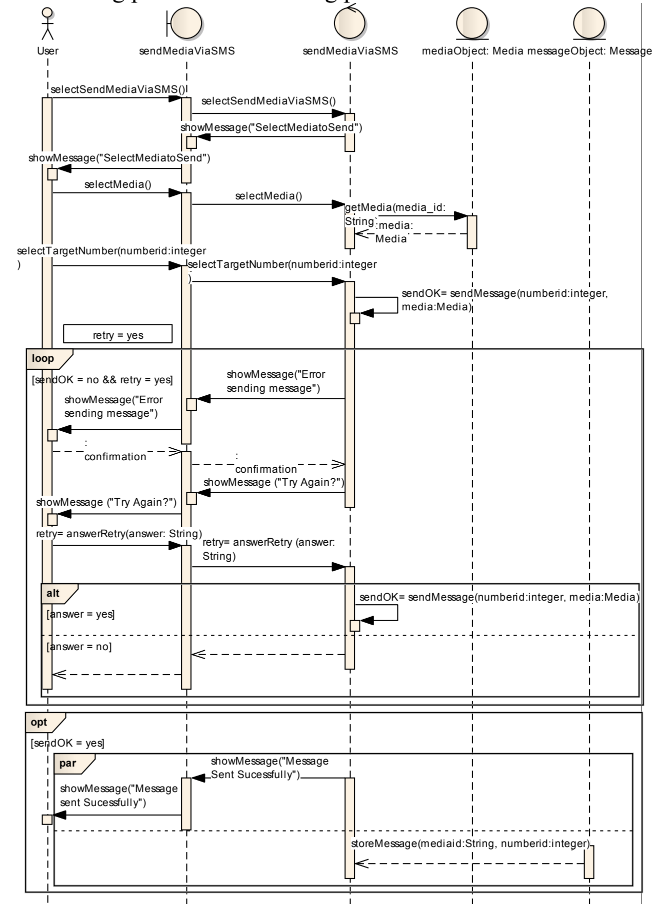

Figure 7 - Refined sequence model for the scenario Send Media Via SMS

When defining these transformations, new problems arise, related to differences in expressivity between models. Sometimes, these differences are addressed by extending the original notation of a model to enrich its semantic information and, thus, facilitate the definition of the transformation rules [13]. Other works [12, 14] address those differences at the transformation level, maintaining the original notation of the models. In this case, the transformations are difficult to express if the models have poor overlap between them. This section briefly presents and comments some scenario-based related work, taking these considerations in mind. We will see that the work we present in this paper complements and completes the existing work.

Whittle and Schumann [12] present an algorithm to automatically generate UML statecharts from a collection of scenarios represented using UML sequence models. In this work, they address several issues, such as detecting conflicts arising from the merging of independently developed sequence models and find behavioural similarities between different sequence models. They do this at the algorithm or transformation level.

There are also works that extend the original UML2 notation to enrich the semantic information needed for a

transformation. An example is the work presented in [13], where they extend the UML2 Activity Model with process goals and performance measures to make them conceptually visible. They also provide transformation rules to BPEL (Business Process Execution Language) to make the measures available for execution and monitoring. In this work, the additional notation defined, for the activity models, allow both models being semantically closer, which made the definition of the transformation rules easier.

Gutiérrez et al [14] propose to generate automatically, through model transformations, an activity model representing the use case scenario from a textual template. In this work, we observed that the semantic inherent to the abstractions present on the template (if-then-else, requirements numeration indicating parallelism) and on the activity model were very close, which resulted in a relatively trivial set of transformation rules.

Petriu and Sun proposed a way to generate activity models from sequence models [15] in a reverse engineering approach, where the source model is more fine-grained than the generated model. This is useful when handling legacy systems. However, in a context where we first model the system at higher levels of abstraction and then progressively move towards a more fine-grained models, the solution proposed in [15] does not help. The work proposed by Dijkman and Joosten [16] is also another example of transforming fine-grained models (business models) into coarse grained models (use case models).

In our approach, we have not extended the activity and sequence models standard notation; we concentrate our effort on the definition of transformation rules to facilitate the semi-automatic generation of sequence models from activity models. Both models have different levels of granularity, representing different viewpoints, which makes the definition of transformation rules more difficult. However, since some information between them overlaps, such as, for example, conditional behaviour or concurrency, it is possible to automate part of the process using model transformations.

VI. Conclusions and Future Work

Modelling scenarios with activity and sequence models of a system can be semi-automated by using transformation techniques, a key concept in MDE. By using transformations, it is possible to reuse information which was directly mapped from one model to another. This frees the burden of the domain analyst from creating similar abstractions which can be automatically generated and also avoids modelling errors, concentrating the effort on the refinement stage of generated artefacts. Transformation rules were defined to generate sequence models artefacts from activity models artefacts. Our transformational rules support the automation of the creation of objects, messages and operators for sequence models from the information contained in activity models.

Our initial validation effort, through the case study described in section IV provided encouraging feedback concerning the desired effort reduction. Indeed, the number of edits required for building a sequence model from the activity model decreased by around 55%, when using our semi-automatic transformation approach. The advantages, from a quality point of view, include: (i) a reduction in the effort building the sequence model, (ii) increased traceability among models (through the semi-automatic translation rules), (iii) error prevention when migrating from different scenarios notations, and (iv) support for reuse of sequence models design best practices, thus providing a good stepping stone for high quality scenario modelling.

For future work, we plan to fully implement the transformation rules described in this paper. Currently, we have an initial version3, which is implemented using Eclipse [10], EMF and UML2 plug-in. We also plan to apply our approach in projects where real case studies are available in order to further validate the claim about time costs improvement provided by our approach. Finally, we plan to extend our approach to support reutilization of refinement information.

References

[1] I. Alexander and N. Maiden, *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*: John Wiley, 2004.

[2] M. Karen and H. Karan, *User-centered requirements: the scenario-based engineering process*: Lawrence Erlbaum Associates, Inc., 1997.

[3] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 1st ed. Reading, MA, USA: Addison-Wesley, 1998.

[4] M. Alférez, U. Kulesza, A. Sousa, J. Santos, A. Moreira, J. Araújo, and V. Amaral, "A Model-Driven Approach for Software Product Lines Requirements Engineering," in *20th International Conference on Software Engineering and Knowledge Engineering*, San Francisco Bay, USA, 2008, pp. 779-784.

[5] W. Hongyuan, Z. Ke, F. Tie, C. Haiyan, and Z. Yinshi, "Synthesizing Statecharts Through Sequence Diagrams Analysis," in *Conference on Software Engineering and Applications*, 2004, pp. 617-622.

[6] M. Volter and T. Stahl, *Model-Driven Software Development*. Glasgow, UK: Wiley, 2006.

[7] S. Beydeda, M. Book, and V. Gruhn, *Model-Driven Software Development*. Berlin, Germany: Springer, 2005.

[8] S. J. Mellor, *MDA Distilled Principles of Model-Driven Architecture*. Boston, MA, USA: Addison-Wesley, 2004.

[9] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*: Addison Wesley, 1992.

[10] D. Carlson, *Eclipse Distilled*: Addison-Wesley Professional, 2005.

[11] T. Young, "Using AspectJ to Build a Software Product Line for Mobile Devices - www.cs.ubc.ca/grads/resources/thesis/Nov05/Trevor_Young.pdf," University of Waterloo, 2005.

[12] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios," in *International Conference on Software Engineering*, Limerick, Ireland, 2000, pp. 314-323.

[13] B. Korherr and B. List, "Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL *," in *25th International Conference on Conceptual Modeling*, pp. 7-18.

[14] J. J. Gutiérrez, C. Nebut, M. J. Escalona, M. Mejías, and I. M. Ramos, "Visualization of Use Cases through Automatically Generated Activity Diagrams," in *Model Driven Engineering Languages and Systems*, Toulouse, France, 2008, pp. 83-96.

[15] D. C. Petriu and Y. Sun, "Consistent behaviour representation in activity and sequence diagrams," in *Third International Conference on the Unified Modeling Language*, 2000, pp. 359-368.

[16] R. Dijkman and S. Joosten, "Deriving use case diagrams from business process models," *CTIT Technical Reports Series,* vol. 08, 2002.

[3] http://ample.di.fct.unl.pt/VML_4_RE/ActivityToSequenceModels.zip