UNIVERSITY of York

This is a repository copy of *Using Model Transformation to Generate Graphical Counter-Examples for the Formal Analysis of xUML Models*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/94539/

Version: Submitted Version

Proceedings Paper:

Santos, Osmar Marchi dos, Woodcock, Jim orcid.org/0000-0001-7955-2702 and Paige, Richard F. orcid.org/0000-0002-1978-9852 (2011) Using Model Transformation to Generate Graphical Counter-Examples for the Formal Analysis of xUML Models. In: Perseil, Isabelle, Breitman, Karin and Sterritt, Roy, (eds.) 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27-29 April 2011. IEEE Computer Society Press , pp. 117-126.

https://doi.org/10.1109/ICECCS.2011.19

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

2011 16th IEEE International Conference on Engineering of Complex Computer Systems

Using Model Transformation to Generate Graphical Counter-Examples for the Formal Analysis of xUML Models

Osmar M. dos Santos
Department of Computer Science
University of York
York, UK
Email: osantos@cs.york.ac.uk

Jim Woodcock
Department of Computer Science
University of York
York, UK
Email: jim@cs.york.ac.uk

Richard Paige
Department of Computer Science
University of York
York, UK
Email: richard@cs.york.ac.uk

***Abstract*—The INESS (INtegrated European Signalling System) Project, funded by the FP7 programme of the European Union, aims to provide a common, integrated, railway signalling system within Europe. INESS experts have been using the Executable UML (xUML) language to model an executable specification of the proposed system. Due to safety-critical aspects of these systems, one key idea is to formally analyse them. In this context, we have been working with other universities on different translation-based methods that enable the formal verification of xUML models. At the core of this approach is a verification framework based on model transformation technology, used to implement an automatic and transparent verification method for xUML. Since a translation-based approach is used, a key aspect to achieve transparency is the automatic generation of counter-examples for verified properties that have a false result during the analysis, in terms of the original xUML model. We describe in this paper how we achieve this using model transformation technology.**

***Keywords*-Model transformation; Executable UML; Formal verification.**

I. Introduction

UML is the *de facto* language for modelling software systems in industry. In particular, one of its profiles, Executable UML (xUML) [15], augments a subset of UML with an action language that adds enough information to enable, amongst other features, creating objects, establishing references and performing operations. From the developer's viewpoint, this has the benefit of providing means to quickly prototype the system at the modelling level, which can then have its behaviour analysed, for instance by simulation.

Currently, we are taking part in the INESS (INtegrated European Signalling System) [6] Project. INESS is an industry-focused project funded by the FP7 programme of the European Union, comprising 30 partners, including 6 railway companies. The objective is to provide a common railway signalling system that integrates existing European ones. Signalling systems are perhaps the most significant part of the railway infrastructure: they are essential for the performance and the safety of train operations. Two of the objectives of INESS are to produce a common core of validated, standardised functional requirements for future interlockings, and to provide safety-verified test tools and techniques to enable the testing and commissioning of future signalling applications.

In this context, INESS experts have been using xUML to model a specification of the proposed integrated signalling system. The idea is to use the specified xUML models to check for inconsistencies in the requirements and against core properties of the system provided by professional railway engineers. Currently, xUML models can be analysed only via simulation. Due to safety-critical requirements involved in railway signalling systems, applying formal verification to analyse the model is of vital importance.

In order to provide a formal analysis method for xUML models, model transformation technology has been utilized in our project to generate models that can be used as input to existing, state-of-the-art, formal verification tools. At the core of this approach is the definition of a verification framework, where xUML models should be automatically and transparently translated to different formal target languages. In order to achieve transparency, one key part of the framework is the automatic generation of counter-examples, obtained from properties that have a false result during the verification process, in terms of the original xUML model.

We can find several works in the literature that follow a translation-based approach for the verification of UML and xUML models [14], [19], [21], [5], [3]. Some of them have dealt with the problem of generating counter-examples that are meaningful in the original model, in particular, using UML sequence diagrams to show the counter-examples. In this work we follow the same idea, since UML sequence diagrams provide a useful and easily recognizable abstraction to the user. However, differently from previous work, we provide a framework based on model transformation technology to generate these diagrams. This means that we can easily extend our approach to other languages, which can then be used to analyse xUML models by redefining only those parts that are dependent on the target language of the translation.

In this paper we tackle this issue by describing the requirements and basic issues in implementing this transformation.

978-0-7695-4381-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ICECCS.2011.19

IEEE computer society

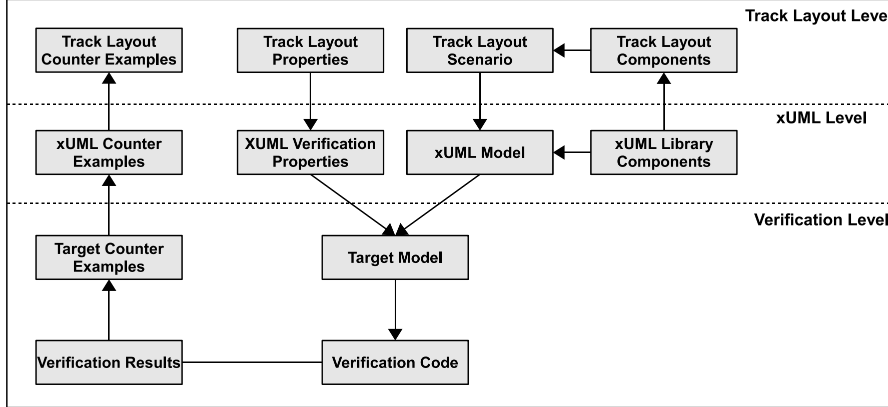

Figure 1: Illustration of the verification strategy.

We start from counter-examples in a text format, which are provided by the verification process, and automatically generate graphical ones, in terms of UML sequence diagrams. In previous work [18] we defined a translation from xUML models to the input language of the SPIN model checker [4]. Where necessary, we use that translation to exemplify the counter-example generation approach.

The paper is structured as follows. The next section provides background material on the verification strategy sought in the INESS project for xUML models and the use of xUML for modelling railway signalling systems. In Section III, we describe the transformation approach for generating graphical counter-examples in our work. Finally, Section IV presents closing remarks and future work.

II. Background

A. Verification strategy in the INESS project

The verification strategy adopted in the INESS project consists of a methodology for the formal analysis of railway signalling system models specified in xUML. Fig. 1 illustrates this methodology, where three different levels of abstraction are presented:

1) Track Layout level: A Domain-Specific Language (DSL) for describing the scenario (diagram) of railway signalling systems for verification should be used. In the project, the xUML language is used to specify different European railway signalling systems and their integration. This culminates in the definition of a set of components that can be combined in different ways. In this sense, the Track Layout level provides an abstraction, understood by railway engineers, that facilitates the definition of analysis scenarios. A component at this level provides a direct, one-to-one, mapping to an xUML component (at the xUML level). Therefore, we do not focus on the Track Layout level. Instead, we direct our efforts to the xUML level.
2) xUML level: This represents the xUML level used to model the integrated railway signalling system. An important element is the xUML Library of railway signalling components that can be put together in order to define an analysis scenario. Given a transformed xUML model of the desired Track Layout, we provide transformation rules to generate a model in the target language (used as input to a model-checking tool) integrating the model and the encoded verification property. Although we have results on the translation of xUML to PROMELA [18], we are looking at different ways to express verification properties in terms of the xUML model.
3) Verification level: This level represents the target verification model, already encoded with the desired verification properties. Once the model has been translated, the task is to generate the verification code, which is actually used in the automated analysis by the verification tool. After verification, it is necessary to translate the results back from the verification level to the xUML level, so that users can view the same abstraction level (transparency, with respect to the verification, is obtained). As already described, we use UML sequence diagrams for that.

Starting from the top level, the verification strategy should work with the definition of a Track Layout scenario. This is mapped to an xUML model. The xUML model is then translated to the input language of a formal verification tool (Target Model), being analysed (Verification Code and Results) and have its results transformed back to the abstractions found in the Track Layout (Counter Examples chain from Verification to Track Layout levels).

Due to the one-to-one mapping between Track Layout and the xUML levels, we are first focusing our work on providing a verification framework for xUML. Once this has been provided, we can define a DSL to specify the

verification scenarios as well as simulate the UML sequence diagrams representing counter-examples.

1) Tool support: We have modified the Papyrus UML modelling tool [16], an open-source plug-in for the Eclipse platform, to specify xUML models of railway signalling systems. We use Papyrus to show the counter-examples in terms of UML sequence diagrams described in this paper. The translation defined in our previous work [18] used other Eclipse plug-ins to implement the automatic transformation of xUML models to the input language of the SPIN model checker. In this work we also use the same Eclipse plug-ins, namely EMFText [20] and the Epsilon tool-set [8]. Since all these tools are Eclipse-based, we can use Eclipse as the underlying platform to enable the generation of a specialized tool for analysing xUML models.

2) Model transformation: Model transformation technology is used to implement every transition between levels of the verification strategy (Fig. 1). At its most basic form, model transformation consists of defining transformation rules that are executed in order to translate a model A (conforming to a given meta-model) to a new model B (which conforms to another meta-model). The meta-model defines the structure that models must conform to. By using Eclipse and associated tools (Papyrus, EMFText and Epsilon in our case), we have defined our meta-models using the Eclipse Modelling Framework (EMF). Our meta-models are composed of classes, references and basic attributes, such as string and integer variables.

The EMFText [20] tool allows the definition of text syntax for languages that conform to EMF meta-models. It enables the generation of models that are extracted from the text format following the syntax of the language. In other words, it enables text-to-model transformations by parsing the original text model.

Epsilon [8] is both a platform for task-specific model management languages and a framework for implementing *new* model management languages by exploiting the existing ones. Epsilon is currently a component of the Eclipse Generative Modeling Technologies (GMT) research incubator project. More specifically, Epsilon provides a language for direct manipulation of models (EOL) [10], and further languages for model merging (EML) [9], model comparison (ECL) [12], model-to-model transformation (ETL) [13], model validation (EVL) [11], model-to-text transformation (EGL) [17], model migration, and unit testing of model management operations (EUnit). In particular, we use Epsilon for model-to-model and model-to-text transformations.

B. xUML models of railway signalling systems

The Executable UML (xUML) language augments a subset of UML with an action language. INESS experts have been using the tool Cassandra [7], a plug-in for the UML modelling tool Artisan Studio [1], to model railway signalling systems and simulate their execution. The Cassandra tool defines its own action language. In our work, we follow Cassandra's action language, since our intention in the project is to provide experts with the possibility of formally analysing their current railway signalling system models.

The xUML models used to describe railway signalling systems in INESS are composed of class diagrams and states machines. Every class diagram has an associated state machine that describes the behaviour of the class once instantiated (the object). Some characteristics of the classes include the use of integer attributes and derived attributes, which can have a very complex behaviour. Amongst other features, the action language is used to send messages between objects, create objects and set references. To illustrate the xUML models we are dealing with, we present some parts of a very small interlocking example, which we call the Micro model, provided by INESS partners.

Fig. 2 shows the class diagram of the Micro model, which is composed of six different classes. In addition to inheritance and the use of references in the models, we also have integer attributes, like the id described in Fig. 2. In particular, a class called *application*, which does not reference any other classes, is specified to represent an initial scenario for executing the model.

State machines can have initial and normal states. Moreover, they can have concurrent regions and can execute actions when entering and exiting states. With respect to transitions, the following are possible: (i) *signal*-transitions, triggered once a signal is received; (ii) *after*-transitions, executed after a given time specified in the guard has passed; and (iii) *change*-transitions, taken once the condition of the $when$ guard becomes true. Regarding the work in this paper, we are mainly interested in the signals exchanged between objects that trigger *signal*-transitions and the execution of *after*-transitions.

Fig. 3 shows an example scenario for the Micro model specified in the xUML action language. From lines 1 to 7, the objects (T1, T2, T3, S1, P1, R1 and R2) representing the tracks, point, signal and routes of the scenario are instantiated. From lines 8 to 15, the references for the objects are set. For example, at lines 8 and 9, the reference tracks of the route object R1 is set to the track objects T1 and T3, respectively. Note that the railway signalling system scenarios defined in the project are finite. That is, once an initial scenario for the analysis is defined, dynamic creation of objects is not allowed. In Fig. 4, the same scenario defined in Fig. 3 is depicted using a Track Layout diagram. The Track Layout is a closer abstraction for railway engineers. As already described, it effectively has an one-to-one correspondence to the xUML model.

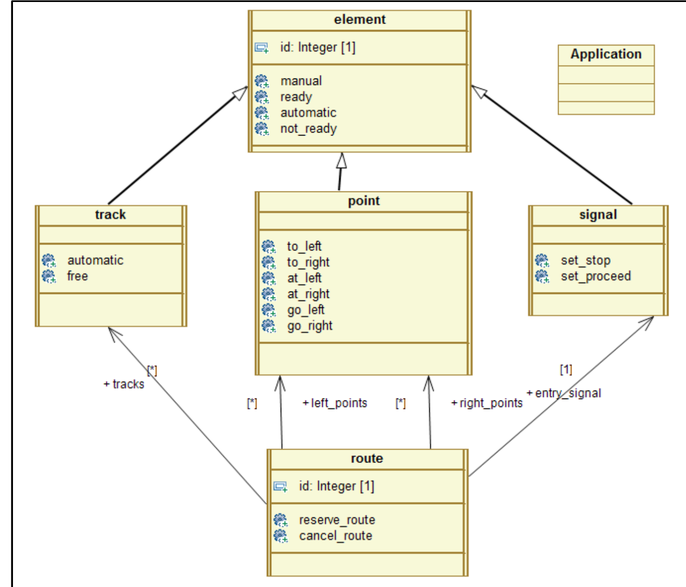

Figure 2: Micro model - class diagrams.

```
1 create T1 from track by track;
2 create T2 from track by track;
3 create T3 from track by track;
4 create S1 from signal by signal;
5 create P1 from point by point;
6 create R1 from route by route;
7 create R2 from route by route;
8 link R1 via route with T1 via tracks;
9 link R1 via route with T3 via tracks;
10 link R1 via route with P1 via left_points;
11 link R1 via route with S1 via entry_signal;
12 link R2 via route with T1 via tracks;
13 link R2 via route with T2 via tracks;
14 link R2 via route with P1 via right_points;
15 link R2 via route with S1 via entry_signal;
```

Figure 3: Micro model - scenario in the xUML action grammar.

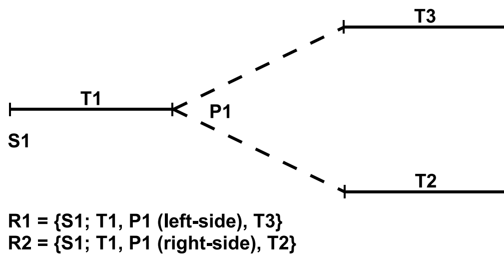

Figure 4: Micro model - example of a possible Track Layout for the scenario.

III. Generation of counter-examples

By using a translation-based approach for the formal analysis of xUML models, we focus on providing UML sequence diagrams to represent counter-examples of verification properties that have a false result during the verification process (of the translated model). UML sequence diagrams provide a simple and effective way to represent the interactions happening on the execution of the system. In order to show the execution of the system using a sequence diagram, we represent each object composing the verification scenario and describe both the exchange of signals between them as well as the triggering of *after*-transitions.

In this section we present a strategy, based on a chain of model transformations, which enables the automatic and transparent generation of such UML sequence diagrams. We start by presenting this chain in Fig. 5. This chain of transformations is implemented inside the Eclipse framework and, once started, works automatically via the execution of a transformation script. According to Fig. 5, the transformation strategy is composed of four steps that represent the generation of the:

1) Counter-example model: includes the parsing of the text file provided by the verification tool. The model obtained maintains the execution order of the parsed counter-example text file.

2) Trace-sequence model: the generation of this model explicitly introduces execution steps for each action that should be present in the UML sequence diagram files.

3) Graphical trace-sequence model: graphical information, related to the generation of the UML sequence diagram, is added.

4) UML files: the output of the whole process includes two UML files. The first represents the elements of

the model, i.e., the objects and the visible actions. The second provides graphical information to correctly display the elements in the diagram.

However, for the strategy to work, we need certain requirements over the input file to hold. Next, we present the basic requirements for this approach to work. Then, in the other subsections we look in more detail at each one of the steps that lead to the models and files described. Where necessary, we provide examples by using our previous translation to PROMELA presented in [18].

A. Basic requirements for the input file

For the approach to work, three basic requirements over the input file have to hold. The first requirement is that the verification tool uses a text format to represent the counter-example file. It is composed of a sequence of statements leading from the initial state of the model to a state where the verification property does not hold any more.

The second requirement is that the ordering for the sequence of statements inside the counter-example file is maintained. This means that the first statements in the file are related to the first execution steps; similarly, the last statements in the file represent the last steps in the execution of the model.

The third requirement is that certain patterns about the execution of the model should be located in the text counter-example file provided by the verification tool. This is related to the gathering of essential information needed to construct a meaningful representation of the model's execution.

Indeed, we required that at least four events that can occur in the execution of the model are captured in the counter-example file. These events are shown in Fig. 6. Besides requiring that the occurrence of an *after*-transition should be detected, we need to know explicitly when a message is being sent (and to which object) as well as when that message is received (a *signal*-transition takes place). This is the minimal information that we need to support the automatic transformation. Note that in the xUML models we are dealing with, the models can generate messages to the environment; therefore, we have to tackle that case. Other xUML models might not have to deal with such case.

a) Execution of an *after*-transition
b) Message sent to an object
c) Message received by an object
d) Message sent to the environment

Figure 6: Basic events required to appear in the counter-example file.

In the implementation we have defined for the models generated by the SPIN model checker, these three requirements hold. The first two requirements are true for the output provided by a large portion of formal verification tools, especially model checkers, found in the literature. Regarding the last one, SPIN allows the user to print information of the model's execution using a *printf* statement. We used that feature of SPIN to explicitly specify the events we are interested in, therefore instrumenting the resulting counter-example file with the required events.

B. Counter-example model

Once a counter-example text file with the requirements described previously is generated by the verification tool, the EMFText tool is invoked in order to generate the counter-example model. The transformation triggered by this tool is a text-to-model transformation, where the elements of the input file are parsed and the events of interest are selected.

For this transformation to work, we have to define a meta-model with the information we want to gather in the input text file and how the tool should find this information in the text file (the syntax used). The meta-model used in the definition of the counter-example model is independent of the language used by the model checking tool, being described in Fig. 7.

As depicted in Fig. 7, the main class of the meta-model (*CounterExampleSequence*) contains a list of the actions representing each one of the events described previously (in Fig. 6). This list of actions is ordered by the execution appearance in the counter-example and is represented by four different classes: (i) *AfterTransition*; (ii) *ReceiveMessage*; (iii) *SendMessage*; (iv) *SendToEnvironment*. Moreover, all these classes inherit different abstract classes: *HasReceiverObject*, *HasMessage* and *HasSenderObject*. These are used to specify if the class contains certain fields (used in the next transformations). For instance a class inheriting *HasMessage* will have a *message* field, which describes the name of the message being sent/received.

Differently from the meta-model, the syntax detection is completely language dependent. This happens because each model checking tool might represent the information differently, according to its output language. As presented previously, in our translation to SPIN, we instrumented the model's execution with the desired events via *printf* statements. The syntax rules defined, in terms of the EMFText tool, are illustrated in Fig. 8.

A rule in EMFText always relates a specific class in the meta-model (left-side) to its attributes and references (right-side). Following Fig. 8, the class *CounterExampleSequence* contains several actions. Each action is obtained by deriving the *AfterTransition*, *ReceiveMessage*, *SendMessage*, and *SendToEnvironment* classes. The fields *object*, *receiverObject*, *senderObject* and *message* are all strings extracted from the text file and inserted in the correct field of the counter-example model. The elements presented inside the double quotes are keywords that instrumented the SPIN model, and that have to be present in the text file for the correct parsing.

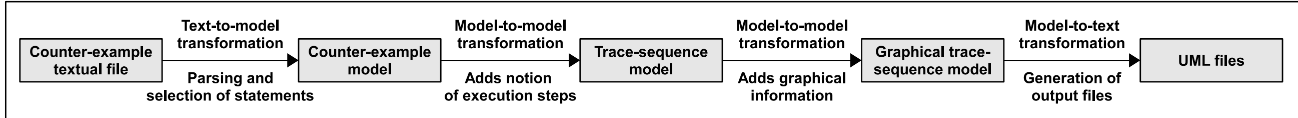

Figure 5: Strategy for generating counter-examples in terms of UML sequence diagrams.

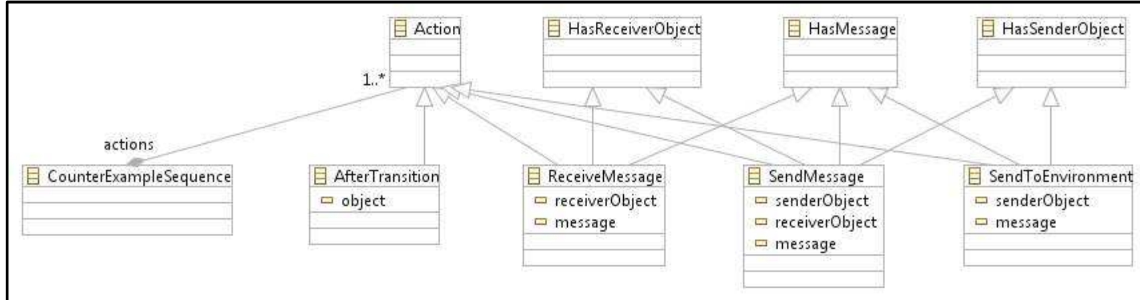

Figure 7: Meta-model for the counter-example model.

```
1  RULES{
2
3    CounterExampleSequence ::= (actions)+;
4
5    AfterTransition ::= "AfterTransition"
6      "(" object[IDENTIFIER] ")";
7
8    ReceiveMessage ::= "ReceiveMessage"
9      "(" receiverObject[IDENTIFIER] ","
10         message[IDENTIFIER] ")";
11
12   SendMessage ::= "SendMessage"
13     "(" senderObject[IDENTIFIER] ","
14         receiverObject[IDENTIFIER] ","
15         message[IDENTIFIER] ")";
16
17   SendToEnvironment ::= "SendToEnvironment"
18     "(" senderObject[IDENTIFIER] ","
19         message[IDENTIFIER] ")";
20
21 }
```

Figure 8: Syntax for selecting events for the counter-example model.

C. Trace-sequence model

With the counter-example model, we implement a model-to-model transformation using the ETL part of the Epsilon tool-set, which generates the trace-sequence model. The trace-sequence model explicitly introduces the notion of execution steps that is used to label each one of the actions in the counter-example. Besides being useful to calculate the correct actions of sending and receiving messages from the counter-example model, it facilitates the positioning of the graphical components of the UML sequence diagram that is defined in the next transformation.

The meta-model for the trace-sequence model is shown in Fig. 9. In the model, the *TraceSequence* class holds the basic components (objects, messages and steps) of the generated trace. Indeed, the abstract class *Step* (that holds the current execution of the event in the trace) is inherited by the four required events, which are defined in the form of the classes: *SendEvent*; *ReceiveEvent*; *ReceiveEventEnvironment*; and *TimeEvent* (used to represent the occurrence of an *after*-transition). Besides the use of the execution steps, this transformation also defines attributes in the form of *xmi_id*'s. Those are used to provide a unique reference value for the elements and are required to correctly generate UML sequence diagrams that can be opened in Papyrus.

In order to better understand how this transformation works, in Fig. 10 we provide some parts of the code for the main a rule definition in ETL (that populates the class *TraceSequence*). Line 1 describes the name of the rule, followed by the element we want to transform from (line 2, element *CounterExampleSequence*) and to (line 3, element *TraceSequence*). In the partial code between lines 15 and 27, all the elements in the list of actions for the *CounterExampleSequence* are selected and transformed (by calling the appropriate function) if the action is an *after*-transition (line 17) or a send event (lines 19 and 20). At the end of the repetition structure, the translated elements are added to the *steps* list of elements (lines 29 and 31). In particular, the *current_events* list holds all the send events in the counter-example and is used later in order to match it to the correct receiving event.

D. Graphical trace-sequence model

Another model-to-model transformation, which uses the Epsilon's ETL, occurs in order to generate the graphical trace-sequence model. Besides introducing graphical positioning information, this transformation code creates blocks that facilitate generating the output files.

According to the meta-model shown in Fig. 11, two main abstract classes are defined: *BasicUML* and *BasicDI2*. These

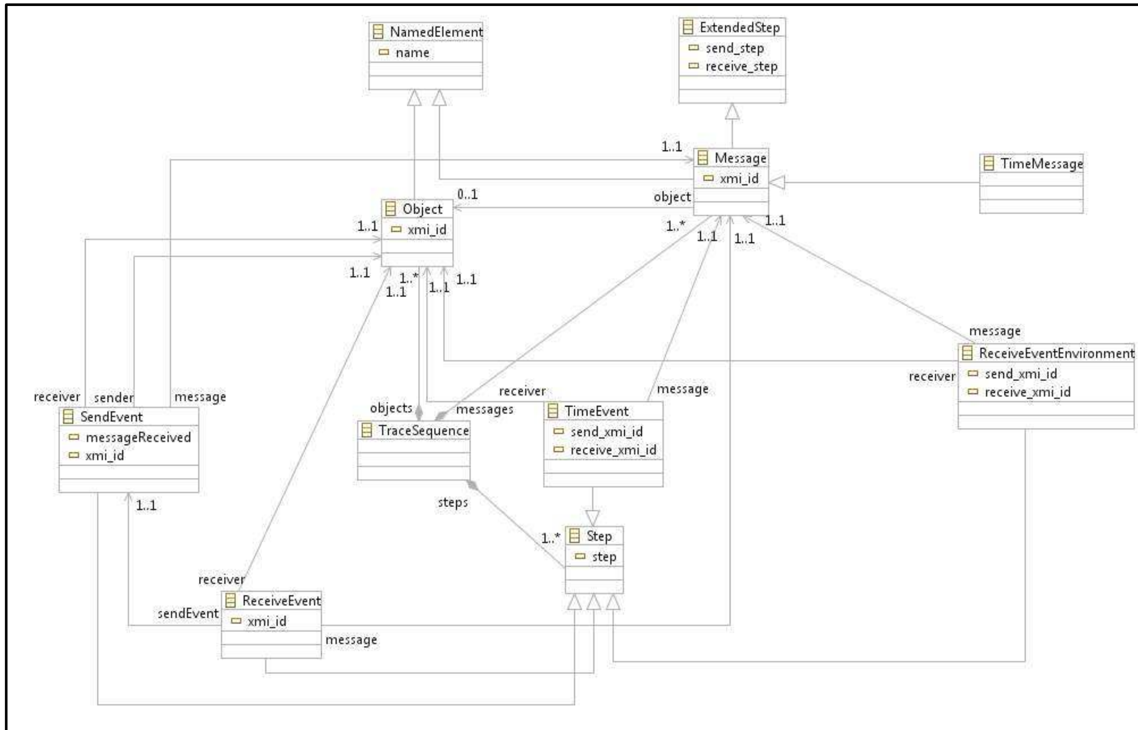

Figure 9: Meta-model for the trace-sequence model.

```
rule GenerateTraceFile
  transform ces : CES!CounterExampleSequence
  to ts : TS!TraceSequence {

  ...

  -- Set time events
  var time_events := OrderedSet{};
  -- Set send events
  var send_events := OrderedSet{};
  -- Set of currently sent events
  var current_events := OrderedSet{};

  step := 0;
  for (a in ces.actions) {
    -- Translate time event
    ces.translateTimeEvent(a, time_events, step);
    -- Translate send event
    ces.translateSendEvent(a, send_events,
                           current_events, step);
    if (a.isTypeOf(CES!AfterTransition)) {
      -- Two steps for the time events
      step := step + 2;
    } else {
      step := step + 1;
    }
  }
  -- Add time events to the model
  ts.steps.addAll(time_events);
  -- Add send events to the model
  ts.steps.addAll(send_events);

  ...

}
```

Figure 10: Portion of an ETL rule describing the generation of the trace-sequence model.

classes provide the code blocks via the fields *basicUMLCode* and *basicDI2Code*, respectively. Elements that are used to generate the final UML file of the sequence diagram, without the graphical positioning information, inherit the class *BasicUML*. The elements that are used to generate the UML file that provides the graphical positioning information inherit the class *BasicDI2*.

Note that the name of the classes also change in this transformation (becoming closer to the UML sequence diagram notation). For instance, *Lifeline*s are mostly *Object*s transformed from the previous model.

E. UML files

The final transformation step is to generate the UML files (without and with graphical information). This last step uses a model-to-text transformation, defined using Epsilon's EGL. EGL allows the definition of templates that can be populated with information extracted from the input model of the transformation. Fig. 12 and 13 show parts of the code used to generate the two needed UML files.

In Fig. 12, the initial lines (from 1 to 6) are used to define the template of the UML file (without graphical data), with its header information. The elements are introduced from lines 10 to 21. For instance between lines 13 to 15, all the elements in the previous model that represent lifelines are added to the file. They are added using the *basicUMLCode* that was generated in the previous transformation step.

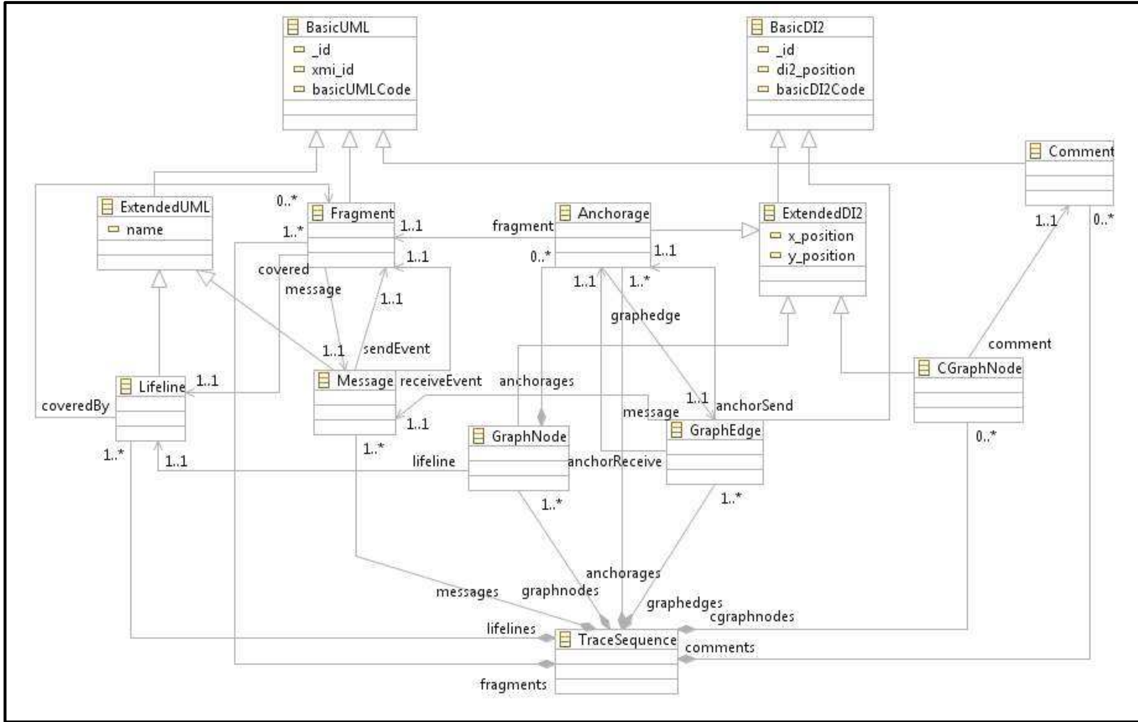

Figure 11: Meta-model for the graphical trace-sequence model.

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http:
//schema.org/spec/XMI/2.1" xmlns:ecore=
"http://www.eclipse.org/emf/2002/Ecore" xmlns
:uml="http://www.eclipse.org/uml2/2.1.0/UML"
xmi:id="_pi2A4Mo9Ed-mtKySmNVTLQ" name="Trace">

  ...

[%for (c in TSG!Comment.allInstances){%]
[%=c.basicUMLCode%]
[%}%]
[%for (l in TSG!Lifeline.allInstances){%]
[%=l.basicUMLCode%]
[%}%]
[%for (f in TSG!Fragment.allInstances){%]
[%=f.basicUMLCode%]
[%}%]
[%for (m in TSG!Message.allInstances){%]
[%=m.basicUMLCode%]
[%}%]

  ...

</uml:Model>
```

Figure 12: Portion of an EGL rule generating the UML file.

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
xmlns:di2="http://www.papyrusuml.org" xmlns:
uml="http://www.eclipse.org/uml2/2.1.0/UML">

  ...

[* POPULATE WITH GRAPHNODES *]
[%
var i = 1;
while (i <= TSG!GraphNode.all.size()) {
  var g :=
   TSG!GraphNode.all.
     select(g_ | g_._id = i).first();
%]
[%=g.basicDI2Code%]
[%
       i := i + 1;
}
%]

  ...

</xmi:XMI>
```

Figure 13: Portion of an EGL rule generating the DI2 file.

Fig. 13 also provides a portion of the code used to generate the other UML file, which holds information about the graphical positioning of the sequence diagram. For instance, from lines 10 to 22 the information about the graphnodes in the sequence diagram is provided.

In order to illustrate the actual UML sequence diagram generated by the transformation process, we present in Fig. 14 a small counter-example obtained from a property verified using a model translated to the SPIN model checker. This property states that route 1 (object R1 in the scenario defined in Section II-B) is reserved only once in the execution of the scenario. This was defined for illustration purposes, since the Micro model should be able to allow

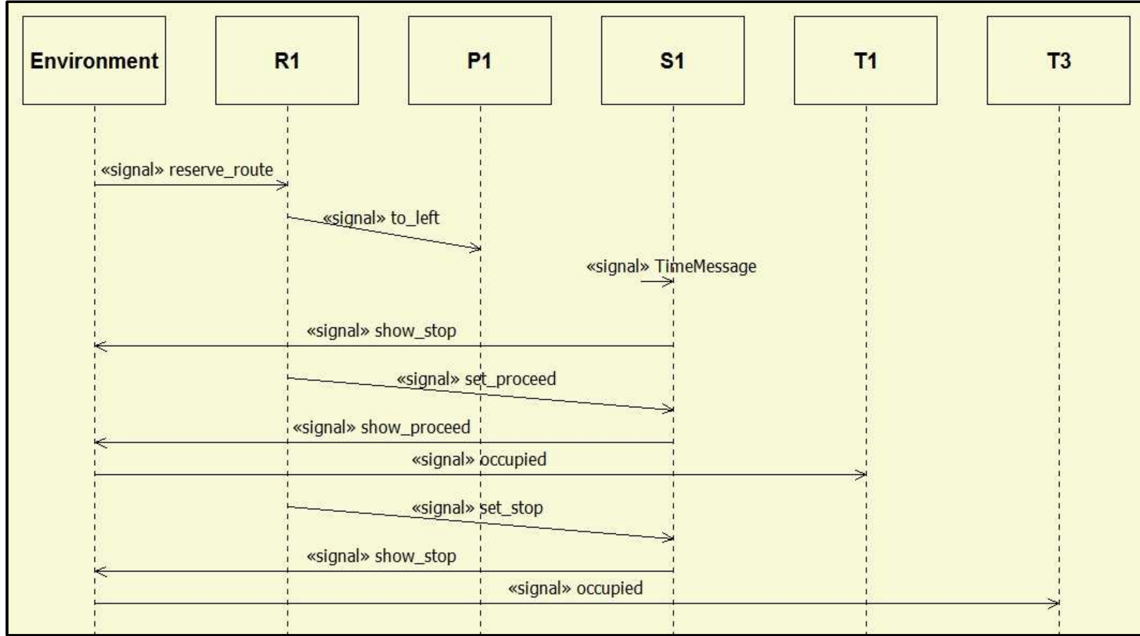

Figure 14: UML sequence diagram of an automatically generated counter-example.

the route to be reserved more than once (after its previous reservation has finished). Although we are not showing the state machines composing the Micro model, understanding this counter-example is straightforward, since it provides the communication exchanged in the system between the reservation of a route and leaving that reservation.

As depicted in Fig. 14 (this is the way Papyrus shows UML sequence diagrams), a *reserve_route* signal is sent from the Environment to R1. In order to reserve the route, R1 sends a *to_left* message to P1 (the point), correctly setting the positioning of the point. Since the track objects T1 and T3 are free and ready, time passes in the execution of the model and the signal object S1 becomes ready (*TimeMessage* represents the execution of an *after*-transition). When ready, the first thing S1 does is to send a *show_stop* message to the Environment (stopping all the traffic). The route reservation continues with R1 sending a *set_proceed* message to S1, which is processed and a *show_proceed* message is generated to the Environment. At this point, the route is reserved and the train can use the route; therefore, an *occupied* message from the Environment arrives to the first track T1. Once the track becomes occupied, this means to the model that the reserved route is being used. R1 then sends a *set_stop* message to S1, which passes the information to the Environment, and R1 tries to reserve another route. Though, before getting to the reservation stage, another *occupied* message is received by T3.

IV. Final remarks and future work

When using a translation-based method for the formal analysis of a modelling language, such as xUML used in this work, one important aspect is to provide an automatic and transparent mechanism for the analysis. This way, the user does not have to understand the translated model in order to use the verification approach. In this paper we presented a method, based on model transformation technology, to automatically generate counter-examples for verified properties (with a false result during the analysis) of translated xUML models. The key point of our work is that, in order to reuse the mechanism described for other formal languages, few modifications have to be made. Indeed, the only language-dependent part of the work described is related to the definition of the syntax used to obtain the minimal information needed to generate the counter-example, in terms of UML sequence diagrams.

This decoupling of language is very important for the INESS project, since we aim to reuse the method to deal with other formal languages (potentially using other verification tools). In this paper we briefly describe how we can generate the counter-examples for the verification of models using a previous translation we defined in [18] for the SPIN model checker. Moreover, we have successfully experimented with the modifications of this implementation to deal with another translation, to the mCRL2 language [2], proposed by one of our project partners at [3]. Though, as a future work, we still want to use the defined framework for other target languages. By increasing the number of languages supported,

we can review the basic requirements of events and add more functionality to the UML sequence diagrams currently proposed.

Our work relies heavily on the use of model transformation technology, which allowed us to break the immediate generation of complex text files (the UML files). As described in the paper, our proposed framework uses four different transformations steps to achieve that. This way, at every new transformation step, we introduced new features to the model and keep the transformation programs small: a very important feature for proposing future modifications and extensions.

Although we have focused our efforts on the verification of xUML models, it is possible to reuse the framework provided for other modelling languages, as long as the counter-example used is provided in terms of UML sequence diagrams.

Moreover, regarding the verification strategy presented for the INESS project, one important future work we will be looking at is the definition of a DSL for describing the counter-examples in terms of Track Layouts.

Acknowledgments The work in this paper was funded by the European Commission via the INESS project, Seventh Framework Programme (2008-2011).

REFERENCES

[1] Artisan Software Tools Inc. Artisan studio UML modelling tool. http://www.artisansoftwaretools.com/, 2010.

[2] J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In *Proceedings of Methods for Modelling Software Systems*, volume 06351 of *Dagstuhl Seminar Proceedings*, pages 1–15, Germany, 2007. IBFI.

[3] H. H. Hansen, J. Ketema, B. Luttik, M. R. Mousavi, and J. van de Pol. Towards model checking executable UML specifications in mCRL2. *ISSE*, 6(1-2):83–90, 2010.

[4] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[5] Z. Hu and S. M. Shatz. Explicit modeling of semantics associated with composite states in uml statecharts. *Journal of Automated Software Engineering*, 13(4):423–467, 2006.

[6] INESS Project. INtegrated European Signalling System (INESS) Project Web Page. http://www.iness.eu/, 2010.

[7] KnowGravity Inc. Cassandra/xUML User's Guide. http://www.knowgravity.com/eng/value/cassandra.htm, 2008.

[8] D. S. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement Project Website. http://www.eclipse.org/gmt/epsilon, 2010.

[9] D. S. Kolovos, R. F. Paige, and F. Polack. Merging Models with the Epsilon Merging Language (EML). In *9th International Conference Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 215–229, Italy, 2006. Springer-Verlag.

[10] D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *2nd European Conference Model Driven Architecture - Foundations and Applications*, volume 4066, pages 128–142, Spain, 2006. Springer-Verlag.

[11] D. S. Kolovos, R. F. Paige, and F. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis*, volume 5115 of *LNCS*, pages 204–218, Germany, 2008. Springer-Verlag.

[12] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *1st International Workshop on Global Integrated Model Management*, pages 13–20, China, 2006. ACM Press.

[13] D. S. Kolovos, R. F. Paige, and F. A. Polack. The Epsilon Transformation Language. In *1sr International Conference on Theory and Practice of Model Transformations*, volume 5063 of *LNCS*, pages 46–60, Switzerland, 2008. Springer-Verlag.

[14] J. Lilius and I. P. Paltor. vUML: a tool for verifying UML models. In *14th International Conference on Automated Software Engineering*, pages 255–258, USA, 1999. IEEE CS Press.

[15] S. J. Mellor and M. J. Balcer. *Executable UML*. Addison Wesley, USA, 2002.

[16] Papyrus UML - CEA LIST. Open source tool for graphical UML2 modelling. http://www.papyrusuml.org/, 2008.

[17] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack. The Epsilon Generation Language. In *4th European Conference on Model Driven Architecture Foundations and Applications*, volume 5095 of *LNCS*, pages 1–16, Germany, 2008. Springer-Verlag.

[18] O. M. Santos, J. Woodcock, R. F. Paige, and S. King. The use of model transformation in the INESS project. In *8th International Symposium Formal Methods for Components and Objects*, volume 6286 of *LNCS*, pages 147–165, The Netherlands, 2009. Springer-Verlag.

[19] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *ENTCS*, 55(3):1–13, 2001.

[20] Software Technology Group - Dresden University of Technology. Emftext concrete syntax mapper. http://www.emftext.org/, 2010.

[21] F. Xie, V. Levin, R. P. Kurshan, and J. C. Browne. Translating software designs for model checking. In *7th International Conference Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 324–338, Spain, 2004. Springer-Verlag.