Finding and fixing bugs in model transformations with formal verification: An experience report*

Gehan M.K. Selim
gehan@cs.queensu.ca
School of Computing
Queen's University

James R. Cordy
cordy@cs.queensu.ca
School of Computing
Queen's University

Juergen Dingel
dingel@cs.queensu.ca
School of Computing
Queen's University

Levi Lucio
levi@cs.mcgill.ca
School of Computer Science
McGill University

Bentley J. Oakes
bentley.oakes@mail.mcgill.ca
School of Computer Science
McGill University

Abstract

We report on the use of a formal verification tool for a graph-based transformation language in the context of a case study. The tool identified two bugs in the transformation that had eluded all previous testing efforts. The paper describes what we learned about the analysis of model transformations and how we intend to use these insights to improve the verification tool.

1 Introduction

In Model-Driven Development (MDD), *model transformations* are used to automate MDD tasks such as querying, model extraction, and code generation [LAD+15]. A model transformation is a program that maps input models (conforming to a source metamodel) into their corresponding output models (conforming to a target metamodel). The ability to identify and avoid bugs in model transformations is very beneficial to MDD in general, and to the successful application of MDD for safety-critical software in particular.

In this paper, we report on our experiences using a tool that we have recently developed for the formal verification of model transformations [SLC+14, Sel15]. The tool analyzes transformations expressed in a graph-based transformation language called DSLTrans with respect to pre- and post-condition pairs. We use this tool for the analysis of an existing model-to-model transformation that transforms state machine models expressed in the UML profile UMLRT [Sel98], to equivalent models in Kiltera [PD10a], a timed extension of the π-calculus.

The contributions of this experience report are two-fold: (1) We summarize some observations that are, hopefully, of value to practitioners developing model transformations and researchers working on quality assurance techniques for model transformations; some of these observations speak to known problems such as the subtleties and limitations of testing and the dangers of refactoring, while other observations are more technical and identify, for example, different kinds of properties that we found useful. (2) We describe how the case study has influenced our plans for future work on our DSLTrans verification tool and further advancing the state-of-the-art in model transformation verification.

The rest of this paper is organized as follows: Section 2 describes the UMLRT-to-Kiltera case study. Section 3 overviews DSLTrans and our property prover. Section 4 demonstrates the verification results. Section 5 discusses the lessons learned. Section 6 summarizes related work. Section 7 concludes the study.

*This work is supported in part by NSERC, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

2 The UML-RT-to-Kiltera Model Transformation: Problem Description

Model transformations are used to achieve many tasks in MDD, one of which is facilitating the analysis of models by translating models into an analyzable language in such a way that analysis results are preserved [LAD+15].

UML-RT [Sel98] is a UML profile that has been used in many industrial sectors (e.g., telecommunication) for the development of event-driven, soft real-time systems. It is supported by commercial MDD tools such as IBM RSA-RTE [IBM] and the open-source tool Papyrus-RT [Fou]. To enable the analysis of UML-RT models, Posse and Dingel recently developed a translation of UML-RT state machine diagrams and capsule diagrams into a language called Kiltera [PD14].

Kiltera [PD10a] is a language for expressing, simulating, and analyzing systems that are either concurrent or distributed. Kiltera allows for code to be executed in different, dynamically changing locations, and supports a notion of time that influences execution behaviour. A Kiltera program consists of *processes* that communicate asynchronously over *channels*. Its formal semantics is based on an timed extension of the π-calculus [PD10b]. As in the π-calculus, channels can be sent as parts of messages which allows for the easy implementation of, for example, call-backs by providing a called process with a "handle" to be used to send computation results back to the caller; interestingly, the passing of channel names allowed for the dynamic aspects of UML-RT (such as optional and dynamic capsules) to be captured succinctly and cleanly.

Paen [Pae12] has implemented a mapping of UML-RT state machines into Kiltera process models [PD14] as an ATL model transformation. We refer to this transformation as the UML-RT-to-Kiltera model transformation. We summarize a subset of the source and target metamodels of the UML-RT-to-Kiltera transformation that are relevant to this paper.

2.1 The Source UML-RT metamodel

In UML-RT, a system's structure is specified as a capsule diagram composed of system components or *capsules*. The behavior of these capsules is specified using state machine diagrams (e.g., Fig. 1). We discuss the concepts of UML-RT metamodel by stating class names (which correspond to UML-RT concepts) in italics, and referring to examples of these concepts in Fig. 1.

A UML-RT *state machine* has one or more (hierarchical) *State*s, for example, states 'n2' and 'n3' in Fig. 1. States are traversed through *Transition*s, such as transition 't1' in Fig. 1. Transitions can be sibling transitions between states in the same hierarchical level, incoming transitions from a state to one of its sub-states, outgoing transitions from a sub-state to its containing state, or initial transitions from a state's *InitialPoint* (e.g., 'init1' in Fig.1) to one of its sub-states (classes *SIBLING0, IN1, OUT2*, and association *initialTransition*). Transitions can have *Trigger*s, where each trigger is composed of a *Signal* received on a *Port*. For example, transition 't1' in Fig. 1 is triggered by signal 'sig1' on port 'p1'. A transition crossing state boundaries is broken into segments, where each segment links *EntryPoints* (e.g., 'a1' in Fig. 1) and/or *ExitPoints* (e.g., 'b1' in Fig. 1).

2.2 The Target Kiltera Metamodel

Next we introduce the concepts of the Kiltera metamodel considered by Posse and Dingel [PD14]. Kiltera includes five classes of constructs: expressions (class *Expr*), patterns (class *Pattern*), guards (Class *ListenBranch*), definitions (Class *Def*), and processes (class *Proc*). Expressions and patterns can be constants, variables, and tuples. Expressions also include function calls. Table 1 enumerates a subset of the guards, definitions, and processes relevant to this study (including their syntax and their corresponding classes from the Kiltera metamodel). We discuss the semantics of these Kiltera constructs in the following.

A process definition of the form $proc\ A(x_1, \ldots, x_n) = P$ defines a process A with parameters x_i that are used in the body of the process P. Thus, the instantiation process $A(E_1, \ldots, E_n)$ instantiates a process defined by $Proc\ A(x_1, \ldots, x_n) = P$ where the parameters x_i are substituted in P by the values of the expressions E_i. The process *done* represents a successfully terminated process. A trigger (i.e., $a!E$) outputs the value of expression E over channel a. In listener processes (i.e., $when\{G_1 \rightarrow P_1 | \ldots | G_n \rightarrow P_n\}$), G_i is an input guard which takes the form $a_i?R_i@y_i$, where a_i is a channel, R_i is a pattern, and y_i is a variable. A listener listens to channels a_i of the guards G_i. When a channel a_i is triggered with a value matching the pattern R_i of guard G_i, three steps are carried out: (1) process P_i is executed, (2) variable y_i of guard G_i stores the time waited by the listener, and (3) the alternative guards are ignored. The new process (i.e., $new\ a_1, \ldots, a_n\ in\ P$) creates the channels a_i that are private to process P. Conditionals have the standard semantics. Local definitions (i.e., $\text{def}\{D_1; \ldots; D_n\} in\ P$) declare the definitions D_i and executes P, where the scope of D_i is the entire term. Parallel and sequential processes represent the parallel and sequential composition of the processes in the term.

Table 1: The names of Kiltera's constructs, their syntax, and their representative classes

Name	Syntax	Corresponding class from Fig. 2
Input Guards	`a?R@y`	Class *ListenBranch* : attributes *channel* and *after and* an association to class *Pattern* represent *a, y,* and *R* in `a?R@y`
Process Definition	`proc A(x1,...,xn)=P`	Class *ProcDef* : attribute *name* and associations with classes *Name* and *Proc* represent *A, xi,* and *P* in `proc A(x1,...,xn)=P`
Termination Process	`done`	Class *Null*
Trigger Process	`a!E`	Class *Trigger* : attribute *channel* and association with class *Expr* represent *a* and *E* in `a!E`
Listener Process	`When(G1→P1|...|Gn→Pn)`	Class *Listen* : associations with classes *ListenBranch and Proc* represent *Gi* and *Pi* in `When(G1→P1|...|Gn →Pn)`
New Process	`New a1...an in P`	Class *New* : associations with classes *Name* and *Proc* represent *ai* and *P* in `New a1...an in P`
Conditional Process	`if E then P1 else P2`	Class *ConditionSet* : associations with classes *ConditionBranch* and *Proc* represent the "if/then" clause and the "else" clause.
Instantiation Process	`A(E1, ...,En)`	Class *Inst* : attribute *name* and association with class *Name* represent *A* and *Ei* in `A(E1, ...,En)`
Local Definition Process	`def{D1;...;Dn} in P`	Class *LocalDef* : associations with classes *Def* and *Proc* represent *Di* and *P* in `def{D1;...;Dn} in P`
Parallel Composition Process	`P1 || P2`	Class *Par* : association with class *Proc* represent *Pi* in `P1||P2`
Sequential Composition Process	`P1;P2`	Class *Seq* : associations with class *Proc* represent *Pi* in `P1;P2`

2.3 The UML-RT-to-Kiltera Model Transformation Mapping Rules

Due to space limitations, we describe the required mapping informally, using the examples shown in Figs. 1 and 2. The detailed mapping rules between the UML-RT and Kiltera metamodels are described in [PD14].

Fig. 1 shows a state machine with one composite state $n2$ and Fig. 2 shows the equivalent Kiltera mapping of state $n2$. The UML-RT-to-Kiltera transformation, in general, maps (a) any state n to a Kiltera process definition named S_n, (b) the entering of a state n to an instantiation of Kiltera process S_n, and (c) signals of transitions' triggers to Kiltera channels in the output. Thus, the composite state $n2$ in Fig. 1 is mapped to a process definition S_{n2} (Fig. 2) with some parameters. Sub-states $n3$ and $n4$ of state $n2$ are mapped to nested process definitions S_{n3} and S_{n4} of process S_{n2} (line 3 of Fig. 2).

To encode transitions with triggers for state $n2$, process S_{n2} has a sub-process $Handler$ (lines 9-15 in Fig. 2) which is a listener process that handles all events of state $n2$. For example, one branch of the $Handler$ sub-process waits for input on channel $sig1$ (representing waiting for the reception of $sig1$ by state $n2$). Once an input is received, the $Handler$ sends an exit request to state $n2$'s active sub-state on the $exit'$ channel. When the sub-state sends an acknowledgement on the $exack'$ channel, the $Handler$ 'instantiates' process S_{n1} corresponding to the transition's target state $n1$.

When state $n2$ is entered, the choice of the sub-state to enter next is encoded using sub-process $Dispatcher$ of process S_{n2} (lines 6-8 in Fig. 2). If state $n2$ is entered through entry point $a1$ (identified by the argument passed to parameter *enp* of the $Dispatcher$) that is connected to sub-state $n4$, then the $Dispatcher$ instantiates S_{n4}. If, however, state $n2$ is entered through an entry point that is not explicitly connected to a sub-state, then the $Dispatcher$ follows state $n2$'s initial transition and enters the initial sub-state (i.e., instantiates S_{n3}).

Exit point $b1$ of state $n2$ is mapped to a sub-process B_{b1} of process S_{n2}. Subprocess B_{b1} executes two steps in parallel: (1) triggers a stop handler request on channel sh (short for stop handler), and (2) instantiates process S_{n1} corresponding to the target state $n1$ of the transition leaving the exit point.

3 Background

We briefly overview the DSLTrans model transformation language, the property prover we built for verifying DSLTrans transformations, and properties that are provable using our prover.

3.1 The DSLTrans Model Transformation Language

DSLTrans [BLA+11] is a graphical model transformation language that is Turing incomplete, i.e., DSLTrans can not specify unbounded loops. Transformations built using DSLTrans are confluent and terminating by construction. In DSLTrans, a transformation is composed of a set of ordered layers that are executed sequentially. A layer contains one or more transformation rules that execute in a non-deterministic order but produce a deterministic result. Each rule is a pair (*MatchModel*, *ApplyModel*) where the MatchModel/ApplyModel is a

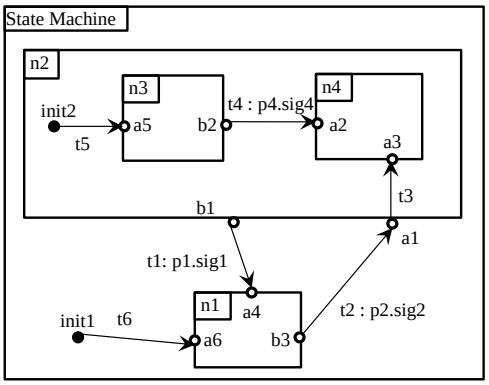

Figure 1: State machine with composite state 'n2'.

```
1   proc Sn2 (exit, exack, sh, enp, ...)=
2   def{
3     proc Sn3 (...)=... ; proc Sn4 (...)= ...;
4     proc Bb1 (sh)=
5       sh! || Sn1(...)
6     proc Dispatcher (exit',exack',sh',enp,...)=
7       if enp="a1" then Sn4(exit',exack',sh',a1,...)
8       else Sn3(exit', exack', sh', init2)
9     proc Handler (exit', exack', sh', ...)=
10      when{
11          exit? →//actions to do on exit request
12          |sig1? →exit'!;
13                   when{exack'? → Sn1(…) }
14          |...
15          }
16    } in
17    new exit',exack',sh' in(
18      Dispatcher(...) || Handler(...)  )
```

Figure 2: The Kiltera mapping of state 'n2' in Fig. 1

pattern of source/target metamodel elements (called match/apply elements in DSLTrans). Match elements can be of two types: *Any* match elements are bound to all matching instances in the input model, and *Exists* match elements are bound to only one matching instance in the input model.

Fig. 3 shows an example of a DSLTrans rule (called 'State2ProcDef') from the first layer of the UML-RT-to-Kiltera transformation. The MatchModel of the 'State2ProcDef' rule has a 'State' element of type *Any* from the UML-RT metamodel and the ApplyModel has one 'ProcDef' element and three 'Name' elements from the Kiltera metamodel. This means that every 'State' input model element will be transformed into a 'ProcDef' output model element connected to three 'Name' elements (with literals *exack, exit*, and *enp*). The attribute name of the 'ProcDef' element is the concatenation of S and the name of the State element in the MatchModel. When a DSLTrans rule executes, traceability links are created between each element in the rule's MatchModel and each element in the ApplyModel. These keep track of which output elements came from which input elements.

Rule 'MapBasicStateNoTrans' in Fig. 4 shows three additional DSLTrans constructs: attribute conditions on match elements, free variables, and backward links. Attribute conditions on match elements (e.g., the conditions on the attributes 'isComposite' and 'hasOutgoingTransitions' of the 'State' match element in Fig. 4) act as a filter on the matching process, where only 'State' elements fulfilling these attribute conditions are matched. DSLTrans uses free variables and backward links to allow a rule to refer to a specific element that has already been created in a previous layer.

The two rules in Figs. 3 and 4 show an example of how free variables and backward links are used, where (i) both rules have a free variable with a value of 'procdef' in the apply element 'ProcDef', and (ii) rule 'MapBasicStateNoTrans' has a backward link appearing as a vertical dashed line between the 'ProcDef' apply element and the 'State' match element. The first occurrence of the free variable 'procdef' (without a backward link) in rule 'State2ProcDef' (Fig. 3) of the first transformation layer binds the 'procdef' variable to the 'ProcDef' element

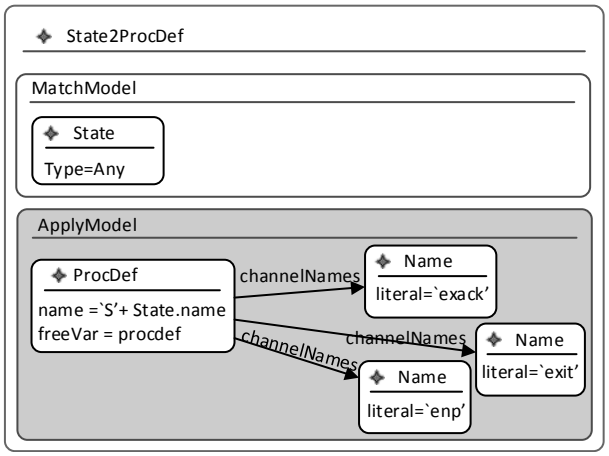

Figure 3: 'State2ProcDef' rule from layer 1 of the UML-RT-to-Kiltera transformation.

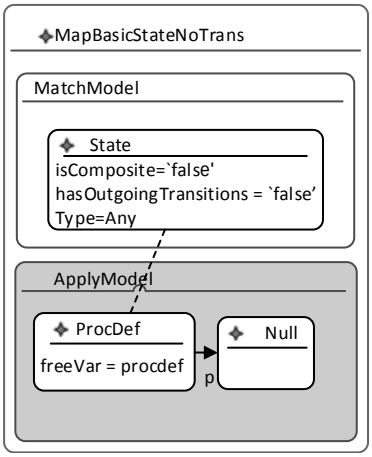

Figure 4: 'MapBasicStateNoTrans' rule from layer 2 of the UML-RT-to-Kiltera transformation.

Layer	Rule Name	Input Types	Output Types
1	State2ProcDef	State	ProcDef, Name
2	MapBasicStateNoTrans	State	ProcDef, Null
	MapBasicState	State	ProcDef, Listen, ListenBranch, Trigger
	MapCompositeState	State	ProcDef, LocalDef, New, Par, Inst, Name
3	ExitPoint2ProcDef	State, ExitPoint	LocalDef, ProcDef, Name, Par, Trigger
	State2Handler	State	LocalDef, ProcDef, Name, Listen, ListenBranch, Null, Seq, Trigger
	State2Dispatcher	State, Transition, EntryPoint, StateMacine	LocalDef, ProcDef, Name, ConditionSet, Inst
4	Trans2InstSIB	Transition, Vertex, StateMachine, SIBLING0	Inst, Name
	Trans2InstOUT	Transition, StateMachine, Vertex, OUT2	Inst, Name
	Trans2Inst	State, Transition, EntryPoint, StateMacine, IN1	Inst, Name
5	Trans2ListenBranch	State, Transition, Trigger, Signal	Listen, ListenBranch, Inst
	MapExitWithTrans	ExitPoint, Transition	Par, Inst
	Trans2HListenBranch	State, Transition, Vertex, StateMachine, Trigger, Signal	Listen, ListenBranch, Seq, Trigger, Inst
	MapStatesINtrans	State, Transition, IN1, Vertex	ConditionSet, ConditionBranch, Expr, Inst
6	MapNesting	State	LocalDef, ProcDef

Table 2: The rules in each layer of the UML-RT-to-Kiltera transformation and their input and output types

generated by the rule. Any occurrences of the free variable 'procdef' in successive layers with backward links (e.g., in rule 'MapBasicStateNoTrans' of the second transformation layer shown in Fig. 4) matches only previously generated 'ProcDef' elements that have been bound to the same free variable 'procdef'. Thus, rules with apply elements that are not connected by backward links (e.g., 'ProcDef' element of rule 'State2ProcDef' in Fig 3) create output elements of the same type each time the MatchModel of the rule is found in the input. However, apply elements that are connected by backward links (e.g., 'ProcDef' element of rule 'MapBasicStateNoTrans' in Fig 4) are used to match an element that has been previously created.

3.2 DSLTrans Implementation of the UML-RT-to-Kiltera Transformation

Table 2 summarizes the rules in each layer of the UML-RT-to-Kiltera transformation, and the input/output types that are mapped/created by each rule. The complete DSLTrans implementation of the transformation is demonstrated in [Sel15].

3.3 DSLTrans Symbolic Model Transformation Property Prover

Fig. 5 demonstrates the architecture of our property prover [SLC+14], now called SyVOLT. Our prover takes four inputs: the DSLTrans transformation of interest, the transformation's source and target metamodels, and the property to verify. Verification is then carried out in two steps, as shown in Fig. 5.

In the first phase, the prover generates the set of *path conditions* representing all possible symbolic executions of the input transformation. Each path condition is generated by accumulating a possible combination of rules that can be triggered by some input model. We refer to the accumulated MatchModels (or ApplyModels) of all the rules in a path condition as the path condition's match pattern (or apply pattern). The path condition generation algorithm is explained in detail in [LOV14].

In the second phase, the prover verifies the input property on each path condition generated in the first phase. The prover renders the property to be either *true* (if the property holds for each of the generated path conditions) or *false* with a counter example (if the property does not hold for at least one path condition). Our property prover is input-independent [ALS+12], i.e., property verification is performed once for the transformation and the verification result is guaranteed to hold for the transformation when run on any input model.

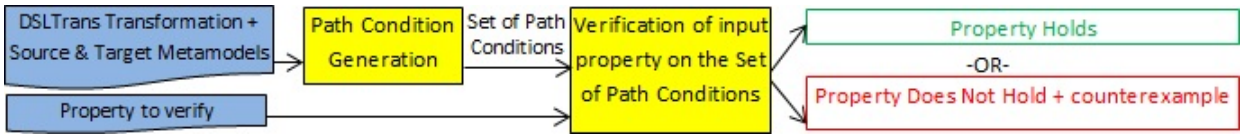

Figure 5: The architecture of our symbolic model transformation property prover.

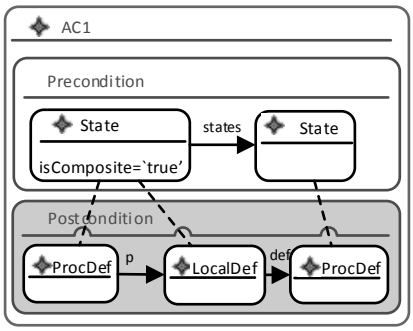

Figure 6: *AtomicContract AC1* used to express property *P1*.

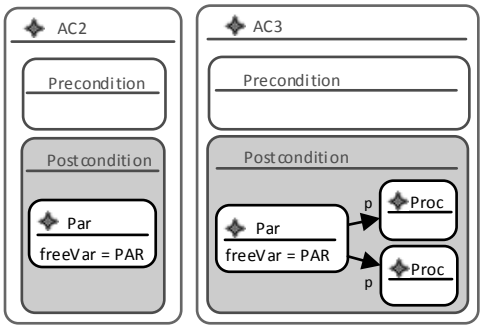

Figure 7: *AtomicContracts AC2* and *AC3* used to express multiplicity invariant *M1* as $AC2 \implies AC3$.

3.4 Properties Verifiable Using the Symbolic Model Transformation Property Prover

Three property types can be expressed and verified using our property prover: *AtomicContracts*, propositional formulae on *AtomicContracts*, and rule reachability. For this study we focus only on the first two property types.

An *AtomicContract* is a pair (*pre, post*) that specifies a property of the form: "if the input model satisfies the precondition *pre*, then the output model should satisfy the postcondition *post*". A (pre- or) postcondition is a constraint on the (input or) output model of the transformation in the form of a structural relation between (input or) output model elements. Pre- and postconditions are expressed using the same constructs as rules (described in Section 3.1). Postconditions may also have traceability links to link postcondition elements to precondition elements. Traceability links in postconditions signify that the property will only match an output model element that was previously created from (and hence, linked to) the input model element.

Fig. 6 demonstrates an *AtomicContract* AC1 used to express a property (referred to as *P1*) of the UML-RT-to-Kiltera transformation. AC1 (Fig. 6) is interpreted as: "two nested *State*s in the input will always be transformed to two nested *ProcDef*s in the output". Using three traceability links in Fig. 6 (appearing as three vertical, dashed lines) mandates that AC1 will only match *ProcDef* and *LocalDef* elements that were previously created from *State* elements. Our property prover should prove that *AC1* will always hold for the *UML-RT-to-Kiltera* transformation.

AtomicConstracts can be composed using standard propositional connectives. For instance, the implication '$AC2 \implies AC3$' in (Fig. 7) captures the '2..*' multiplicity invariant (referred to as *M1*)[1]: In the output, every *Par* element (i.e., a parallel composition) is associated with two or more *Proc* elements (i.e., processes) through the association *p*. More precisely, if an element of type 'Par' (referred to as variable 'PAR') is generated in the output (again as variable 'PAR') in *AC*2, then this element must be connected to at least two 'Proc' elements.

4 Testing and Verification of the UML-RT-to-Kiltera Model Transformation

We begin by briefly describing how the transformation was tested during its development (Section 4.1). We then identify some relevant properties that the transformation should satisfy to be considered correct (Section 4.2). The application of our property prover to the transformation then follows (Section 4.3).

4.1 Testing

The transformation was extensively unit tested using the following process: Each time a rule was created, appropriate input models to test that rule were created depending on the complexity of the rule. If the rule produced the expected output, development would proceed with the next rule; otherwise, the rule would be debugged. In total, the transformation was tested on 25 different input models, none of which revealed any bugs.

4.2 Properties of Interest

We divide the desired properties of the UML-RT-to-Kiltera transformation into four categories: *pattern contracts*, *multiplicity invariants*, *syntactic invariants*, and *rule reachability*. Contracts are properties that relate elements of the source and target metamodels, and are expressed using *AtomicContracts*. Invariants are properties defined on elements of the target metamodel only, and are expressed using propositional formulae of *AtomicContracts*.

[1]Note that the two *AtomicContracts* in Fig. 7 have empty preconditions meaning that they will match on any input model.

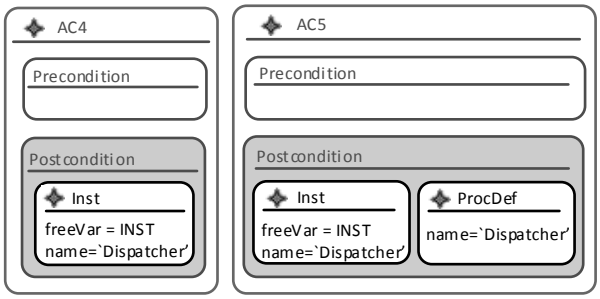

Figure 8: *AtomicContracts AC4* and *AC5* that are used to express a syntactic invariant as $AC4 \implies AC5$.

We summarize the four property categories and we demonstrate how exemplar properties from the four categories are formulated in our prover. The property categories are described in detail in [Sel15].

Pattern contracts require that if a certain pattern of elements exists in the input model, then a corresponding pattern of elements exists in the output model. For example, pattern contract *P1* (Section 3.4, Fig. 6) ensures that "two nested *State*s in the input will always be transformed to two nested *ProcDef*s in the output".

Multiplicity invariants ensure that the transformation does not produce an output that violates the multiplicities in the target Kiltera metamodel. For example, multiplicity invariant *M1* (discussed in Section 3.4, Fig. 7) ensures that each output *Par* element is associated to two or more *Proc* elements through the *p* association.

Syntactic invariants ensure that the generated Kiltera output is well-formed with respect to Kiltera's syntax. An example of a syntactic invariant (referred to as *S1*) ensures that if a process named *Dispatcher* is instantiated, then a process named *Dispatcher* is also defined. Using the *AtomicContracts* in Fig. 8, *S1* can be expressed as $AC4 \implies AC5$. The former propositional formula can be interpreted as "If the output has an *Inst* element named *Dispatcher* (*AC4*), then the output must have the same *Inst* element accompanied with a *ProcDef* element named *Dispatcher* (*AC5*)". The free variable 'INST' in Fig. 8 mandates that if *AC4* holds for a specific *Inst* element, then *AC5* should also hold for the same *Inst* element.

Rule reachability checks whether a specific rule can be triggered in any path condition of the transformation. A rule that is not reachable is said to be a *dead rule* and indicates a transformation bug that needs to be fixed.

We defined and formulated 11 multiplicity invariants, 3 syntactic invariants, 5 pattern contracts, and 15 rule reachability checks (for each of the 15 rules summarized in Table 2).

4.3 Verification

We used our property prover to verify these properties for the UML-RT-to-Kiltera transformation.

4.3.1 Performance

The first phase of the verification, the generation of the path constraints (Figure 5), completed in less than 14 seconds[2] and resulted in 57 different path conditions, i.e., 57 different feasible sequences of rule applications.

In the second phase of the verification, the path conditions are checked to see whether or not they satisfy the property input. For properties $P1$, $M1$, and $S1$ described in Section 4.2, this check completed in 12.72 secs, 1.59 secs, and 5.5 secs, respectively. Overall, none of the 11 multiplicity invariants took more than 2 secs to check, while the verification of the 5 pattern contracts took between 3 and 22 secs. The check of two syntactic invariants completed in less than 6 secs, while the third required 241 secs; the reason is that it is by far the most complex property, with 20 elements distributed over 4 atomic contracts.

4.3.2 Bugs found

To our surprise, SyVOLT determined that the transformation was not correct, because it did not guarantee properties *M1* and *S1* (Figs. 7 and 8). More precisely, there are input models for which the transformation generates: (i) an output in which a *Par* is associated to fewer than two *Proc*s (violating *M1*), and (ii) an output where an *Inst* named *Dispatcher* is created but a corresponding *ProcDef* named *Dispatcher* is not created (violating *S1*). After examining the generated counter examples, we determined that both bugs were caused by two rules $R1$ and $R2$ in different layers that were not guaranteed to be "applied together", i.e., that it was possible that $R1$ was applied, but not $R2$.

[2]All timings were done on a 2.8 GHz AMD Opteron processor running Ubuntu Linux.

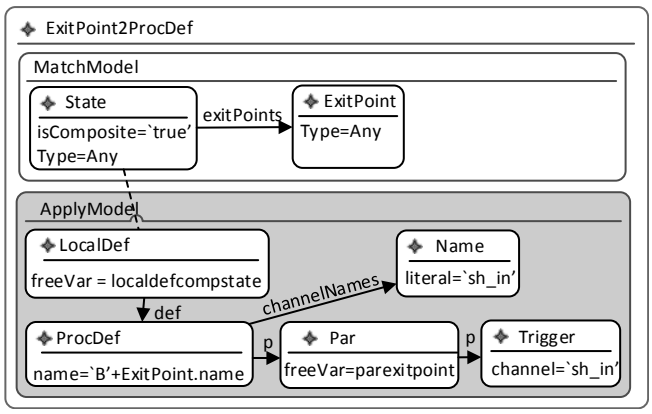

Figure 9: Rule 'ExitPoint2ProcDef' in layer 3 of buggy transformation.

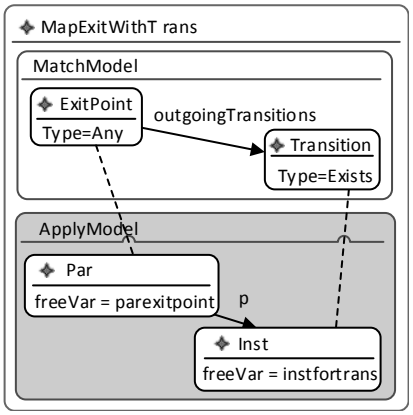

Figure 10: Rule 'MapExitWithTrans' in layer 5 of buggy transformation.

Neither of these bugs had been exposed by our rule testing while developing the transformation, and when we went back to the the original UML-RT-to-Kiltera transformation in ATL presented in [Pae12], it too, although also having been tested quite thoroughly, turned out to also contain the exact same two bugs.

An investigation of the source of these bugs revealed some details:

1. **Bug 1**: Rule 'ExitPoint2ProcDef' in layer 3 (Fig. 9) and rule 'MapExitWithTrans' in layer 5 (Fig. 10) are supposed to generate the two *Proc* elements belonging to a *Par* element. First, rule 'ExitPoint2ProcDef' (Fig. 9) generates a *Par* element associated to a *Trigger* element (which extends *Proc*). Then, rule 'MapExitWithTrans' (Fig. 10) associates an *Inst* element (i.e., a second *Proc* element) with the same *Par* element previously generated by rule 'ExitPoint2ProcDef' in layer 3, as shown by the free variable 'parexitpoint'. *However, execution of rule 'ExitPoint2ProcDef' does not mandate execution of rule 'MapExitWithTrans'*: e.g., a composite *State* in the input model with an *ExitPoint* that has no outgoing *Transition*s will cause rule 'ExitPoint2ProcDef' (layer 3) to execute but not rule 'MapExitWithTrans' in layer 5, resulting in an output containing a *Par* associated with only one *Proc*, violating *M1*.

2. **Bug 2**: Rule 'MapCompositeState' in layer 2 generates an *Inst* named 'Dispatcher' and rule 'State2Dispatcher' in layer 3 generates a *ProcDef* named 'Dispatcher'. Rule 'State2Dispatcher' matches composite *State*s with a positive application condition, or a PAC (specified using *Exists* match elements, described in Section 3.1). On the other hand, rule 'MapCompositeState' matches any composite *State*, without specifying a PAC. Thus, rule 'MapCompositeState' will match some composite *States* that are not matched by rule 'State2Dispatcher' (if they do not satisfy the PAC), resulting in an output containing an *Inst* named *Dispatcher*, but not a *ProcDef* named *Dispatcher* (violating *SS2*).

4.3.3 Fixing the bugs

For the first bug, we merged the two rules 'ExitPoint2ProcDef' and 'MapExitWithTrans' into a new rule in layer 5. For the second bug, the MatchModel of the rule 'MapCompositeState' (layer 2) was updated to include the PAC (specified as *Exists* match elements) of rule 'State2Dispatcher' (layer 3), to guarantee that the two rules necessarily execute together. Due to page limitations, the new rules are not shown here (see [Sel15] for details). After these changes, our prover proved the revised transformation correct with respect to all 11 properties.

5 Observations

Our case study allowed us to make the following observations:

O1: "Bugs not triggered by a test input will not be found": The limits of testing are well-known, of course. However, the unwarranted trust we subconsciously placed in our tests surprised us and highlights the value of formal verification.

O2: "Make sure test inputs cover the input metamodel": Testing proved insufficient, because the metamodel was assumed to be more restrictive than it actually was, i.e., the input models produced by the prover as counter examples had been assumed to be malformed, but they proved to be permissible UML-RT state machines.

O3: "Effective input-independent verification of graph-based model transformations is possible": While the performance of an earlier version of our prover [SLC+14] was already quite good, the performance of SyVOLT observed in this non-trivial case study is encouraging and provides additional, albeit still anecdotal, evidence that the formal verification of transformations with respect all possible inputs is feasible and practical.

O4: "Refactoring is hard": The first bug was introduced by a refactoring step that broke a single rule into the two rules 'ExitPoint2ProcDef' and 'MapExitWithTrans' described in the previous section. Unfortunately, the refactoring did not preserve correctness and was undone to fix the bug.

O5: "Input/output-level properties" vs "rule-level properties": The bugs suggested to us that it is useful to distinguish two different kinds of properties: 1) *input/output-level properties*, i.e., pre- and post-condition-type properties that describe the desired shape of the output (e.g., all properties in Section 4.2); and (2) *rule-level properties*, i.e., properties that impose restrictions on the way the rules are applied in a transformation execution (e.g., "Rule R1 fires in an execution if and only if rule R2 also fires").

Input/output-level properties capture user-level requirements, while rule-level properties capture when the rules in an implementation work together properly to guarantee the requirements. The relationship is akin to standard pre- and post-conditions for programs (i.e., "contracts") and, e.g., behavioural interface specification and API method usage rules such as "method *close* should only be invoked after method *open*". The benefit of rule-level properties thus is that they, in some sense, describe how the transformation works and may provide necessary conditions useful for transformation development and documentation.

O6: Towards a development methodology for provably correct DSLTrans transformations: A hallmark of DSLTrans is that transformations are structured in sequentially executed layers $L_1, \ldots, L_n$. The property language supported by SyVOLT is perfectly suited to capture the purpose of each layer L_{i+1} through a pair of formulas F_i and F_{i+1} capturing input/output-level properties, such that the rules in L_{i+1} are deemed correct, if they transform input satisfying the pre-condition F_i into output satisfying the post-condition F_{i+1}. Development of the rules in layer L_{i+1} would go hand-in-hand with the development of the formulas F_{i+1} with the iterative use of the prover until (i) the rules are correct with respect to F_{i+1}, and (ii) F_{i+1} is considered strong enough to allow the construction of L_{i+2} and a suitable F_{i+2}. Failure to establish F_{i+1} may force the developer to revisit a previous level L_j ($j \leq i$) and revise F_j and, possibly, also the rules in L_j.

6 Related Work

This paper presents results from our ongoing work on verifying graph-based model transformations. While an earlier version of the prover has been described before [SLC+14], the UML-RT-to-Kiltera case study and our experience verifying it is new to this paper. In a performance comparison in [SBC+13] (using a different case study), our prover performed substantially better than the approach based on satisfiability solving described in [BECG12].

Tools to evaluate the coverage provided by a set of input models with respect to a given metamodel have been proposed [FBMLT09] and may have revealed the limitations of our initial tests. Tools that allow what we call "rule-level properties" include Groove [GdMR+12] and AGG [Tae03]. AGG supports a "critical-pair analysis" which checks if transformation rules are confluent, i.e., applications of different rules to the same graph will produce the same result. Groove's analysis is more comprehensive and supports a complete exploration of the state space of the transformation; during the exploration a CTL formula is checked in which a atomic proposition captures the applicability of a rule; this way, both input-output- and rule-level properties can be checked.

7 Conclusion and Future Work

The case study has been useful in the following ways: First, it has reinforced some already widely held, but unproven, beliefs (Observations O1, O2, and O4). It provides some additional evidence of the promise of our approach to model transformation verification (Observation O3). Also, it provided us with a stimulus to think more about different kinds of properties; rules form the building blocks transformations are made of and occupy

a higher level of abstraction than, say, statements in programming languages, while also perhaps being more uniform in their effect and role than, say, methods and procedures in programming languages; this may mean that the development of rule-based transformation systems in general and graph-based model transformation systems in particular may benefit greatly from the kind of rule-level properties discussed in Observation O4. Finally, the case study has given us new ideas about how to evolve our work into a transformation system providing suitable tool support for the rigorous development of correct DSLTrans transformations (Observation O5). In particular, support for the expression and verification of properties of individual layers and rule-level properties will be a focus. Encouragingly, it should be fairly straight-forward to extend our prover to support these additions.

References

[ALS+12] M. Amrani, L. Lúcio, G.M.K. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J.R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. *VOLT 2012*, pages 921–928, 2012.

[BECG12] F. Buettner, M. Egea, J. Cabot, and M Gogolla. Verification of atl transformations using transformation models and model finders. In *ICFEM 2012*, pages 198–213, 2012.

[BLA+11] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *SLE 2011*, pages 296–305. 2011.

[FBMLT09] F. Fleurey, B. Baudry, P.-A. Muller, and Y. Le Traon. Qualifying input test data for model transformations. *Software and Systems Modeling*, 8(2):185–203, 2009.

[Fou] Eclipse Foundation. Papyrus for real time (papyrus-rt). https://projects.eclipse.org/projects/modeling.papyrus-rt, 2015.

[GdMR+12] A.H. Ghamarian, M.J. de Mol, A. Rensink, E. Zambon, and M.V Zimakova. Modelling and analysis using groove. *Int. J. on Softw. Tools for Technology Transfer*, 14(1):15–40, 2012.

[IBM] IBM. IBM Rational Software Architect Real-time Edition, version 8.0. 2015.

[LAD+15] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. Selim, E. Syriani, and M. Wimmer. Model Transformation Intents and Their Properties. *Software and Systems Modeling*, 2015. To appear.

[LOV14] L. Lúcio, B. Oakes, and H. Vangheluwe. A Technique for Symbolically Verifying Properties of Graph-Based Model Transformations. Technical Report SOCS-TR-2014.1, McGill University, 2014.

[Pae12] E. Paen. Measuring Incrementally Developed Model Transformations Using Change Metrics. Master's thesis, School of Computing, Queen's University, 2012. MSc thesis.

[PD10a] E. Posse and J. Dingel. Kiltera: A Language for Timed, Event-Driven, Mobile and Distributed Simulation. In *DS-RT 2010*, pages 87–96, 2010.

[PD10b] E. Posse and J. Dingel. Theory and Implementation of a Real-Time Extension to the π-Calculus. In *Formal Techniques for Distributed Systems*, pages 125–139. 2010.

[PD14] E. Posse and J. Dingel. An Executable Formal Semantics for UML-RT. *Software and Systems Modelling*, pages 1–39, 2014.

[SBC+13] G.M.K. Selim, F. Buettner, J.R. Cordy, J. Dingel, and S. Wang. Automated verification of model transformations in the automotive industry. In *MODELS 2013*, pages 690–706, 2013.

[Sel98] B. Selic. Using UML for Modeling Complex Real-Time Systems. In *LCTES*, pages 250–260. 1998.

[Sel15] Gehan M.K. Selim. *Formal Verification of Graph-Based Model Transformations*. PhD thesis, School of Computing, Queen's University, 2015.

[SLC+14] G. M. K. Selim, L. Lúcio, J. R. Cordy, J. Dingel, and B. J. Oakes. Specification and Verification of Graph-Based Model Transformation Properties. In *ICGT 2014*, pages 113–129, 2014.

[Tae03] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *AGTIVE 2003*, pages 446–453, 2003.