Verification of RoseRT models using Petri nets

Marcel van Leeuwen[1], Lou Somers[1], Marc Voorhoeve[1], and Jan Martijn van der Werf[1]

[1] Department of Mathematics and Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
m.v.leeuwen@student.tue.nl,{l.j.a.m.somers,m.voorhoeve,j.m.e.m.v.d.werf}@tue.nl

Abstract. Rose RealTime is a widely used development tool for embedded software. In this paper a translation into classic Petri nets is proposed to support correctness verification of software models created with this tool. The rich tool set available for both structural and behavioral analysis of Petri nets makes the translation applicable to all sorts of correctness verifications. Verification results are automatically visualized in RoseRT, thereby providing a comprehensive overview of a possible error trace to the software developers, whilst the Petri net internals remain hidden. The method is confronted with a real-life case.

Keywords: RoseRT, UML 2.0, Petri nets, verification, deadlock traces, model transformation.

1 Introduction

The development of the embedded control software for large complex electromechanical systems is not a trivial task. Typically such a system contains hundreds of sensors and actuators (like motors, clutches, heaters) that have to be controlled. Usually, the control software is handcrafted in a language like C of C++ and consists of a number of concurrently executing threads with different priority, sharing resources. To add to the complexity, the execution platform even may be distributed over multiple processing nodes. During development, situations like deadlock or race conditions are hard to avoid.

To avoid such problems and to be able to manage the growing development effort, Océ already in 1997 decided to choose for model driven development [1]. At that time, ROOM [2] was the only readily available option. During the course of time, the ROOM method evolved into RoseRT and its concepts became encapsulated in UML 2.0. Currently, UML-RT of Rational Rose RealTime (RoseRT) [3], RT-UML of Telelogic Rhapsody [4], and TURTLE [5] are all based on the UML 2.0 standard [6], but differ somewhat on how the UML standard has been implemented.

The use of Rose-RT with its run-to-completion semantics, coupled to a strict use of architectural guidelines, greatly reduces the likelihood of introducing errors like lifelock and deadlock. However, such problems cannot be avoided completely. A nice way to check for such logical errors would be to translate the Rose-RT models into

some formal language that allows an automatic check. Some tests with translations to algebraic specifications show that this is indeed possible (with some manual work) for relatively small parts of the model, but far from feasible for the embedded models of a complete high speed Océ printer or copier [7]. In this paper, we describe the automatic translation of Rose-RT models to Petri nets and show what can be verified using this translation.

This paper is organized as follows. Section 2 presents related work. In section 3 RoseRT and its modeling technique are presented. The translation of RoseRT models to Petri nets is explained in section 4. As the resulting Petri nets can be large, reduction rules are defined (section 5). Analysis methods and a case study are presented in section 6.

2 Related work

Other attempts to formally translate (parts of) RoseRT models, or UML models in general, into mathematical formalisms are briefly presented in this section.

Tests to verify the boundedness of UML-RT communication buffers are presented in [8]. However, the transition code is ignored. In [9], [10], and [11] translations to Generalized Stochastic Petri Nets (GSPNs) are defined, mainly with the aim of performance evaluations.

Another approach is to take advantage of the object-oriented structure available in UML models by defining a translation into Object Petri nets (OPNs) [12]. A disadvantage of this approach is the current lack of analysis and simulation tools that support this class of Petri nets.

Colored Petri nets are another class of Petri nets that has been used to formally translate UML models. The approach taken in [13] focuses on the translation of sequence diagrams into colored Petri nets. Although overlap with the translation of structure diagrams in combination with state machines is present, the translation itself ignores some elements essential to our translation. Another approach, the Customization Rules (CR) approach, to formalizing UML using colored Petri nets is presented in [14]. The authors argue that one universal translation is not feasible, therefore a set of rules in which users define UML semantics as they interpreted them, hence customization, needs to be defined before making a formal translation.

In [15] semantics are defined using classic Petri nets. Some of the presented translation constructs could also be applied in our translations.

Next to Petri net-based translations, also other mathematical formalisms have been used to formalize RoseRT or UML models. [16] represents state diagrams as graphs and the transitions between elements with transformation rules. [17] captures UML-RT semantics with flow-graphs. In [18] a translation of UML elements into the specification language CASL is proposed. The authors restrict the translation to state machine constructs of which semantics are clearly defined. A translation to PROMELA is defined in [19], where the models are checked with the model checker SPIN. [20] provides a good starting point for translations into other mathematical formalisms and it presents a translation into the language BIR, the input language for the model checker BOGOR.

3 Rose RealTime

Roughly speaking, a RoseRT model consists of concurrently executing state machines. These state machines describe the state of so-called capsules, which are connected to each other. The "wires" between the capsules indicate to which capsule the signals (events) that are produced in a state transition should be sent.

The state machines are mapped to threads (more than one state machine per thread is possible). Each thread manages an event queue. Incoming signals are placed at the end of the queue. The event handler of the thread iteratively takes the top event of the queue and performs the corresponding state transition (if any). The next event is handled after the complete action code of this state transition has finished (run-to-completion semantics).

In general, a RoseRT model consists of four views: the use case view, the logical view, the component view, and the deployment view. Our translation is based on elements from the logical view, as this view defines the system behavior (i.e. executable code is generated from this view). The relevant elements are *protocols*, *classes* and *capsules*. Protocols describe *signals* that can be sent or received by ports defined on capsules, classes are classes as known from object-oriented programming [21], and capsules are the constructs with which actual system behavior is modeled.

A RoseRT model is defined as a hierarchy of capsule instances, called capsule roles, encapsulated in one another. Encapsulation of capsule roles is defined in structure diagrams. For an encapsulated capsule role, the cardinality might be specified, which sets an upper bound on the number of runtime instantiations that can be created. A structure diagram defines, next to encapsulation, the connections to other capsule roles (the "wiring"). Ports, i.e. the constructs via which capsule roles communicate, are also defined in the structure diagram. Each port realizes a protocol, which is a merely a list of signals. An instance of a port is referred to as a port role for which cardinality can be specified. Connections are created between port roles. Connections from a capsule role can be created to capsule roles on the same level in the hierarchy or one level higher or lower.

Each capsule has its own state diagram in which the dynamic behavior is defined. A state diagram consists of an *initial point*, *junction points*, *choice points*, *composite* and *simple states*, and *transitions*. A transition has a source state or point and a destination state or point. Transitions can be triggered and possibly execute code. A transition is triggered by the reception of a certain signal on one of its ports and the code specified on the transition is atomically executed; its execution cannot be interrupted. A self-transition is a transition with the same source and destination state. Choice points have one incoming and two outgoing transitions, a Boolean predicate specifies which outgoing transition is executed. Transitions are connected to states using junction points. In a composite state, a state which is a state machine itself, junction points can be connected to a state or point within the composition. A terminating junction point does not have outgoing transitions.

State machines follow the run-to-completion paradigm, meaning that execution of a transition cannot be interrupted until it terminates on a simple state. Such a path from one simple state to another one might contain multiple choice and junction points. The current state of a state machine is defined by the simple state it is currently in: the active state.

Composite states enable the creation of a state hierarchy. When the active state is at a level *x* in the hierarchy and a trigger is received, a search for a transition to execute is started. First all outgoing transitions of the active state are examined, if none is found the search is continued at the composite state in which it is enclosed until the top state is reached. Transitions that originate on a composite state, and therefore can only be executed when one of the states it is composed of do not have a transition triggering on a certain incoming event, are referred to as border transitions.

4 Translation to Petri nets

Classic place-transition Petri nets [22] have been chosen to formally translate RoseRT models to. Their semantics are well defined and numerous methods to prove certain properties of a Petri net have been developed throughout the years.

The translation of RoseRT models into Petri nets concerns the translation of the specific UML constructs that define the interaction between the components of a software model, and the translation of the C++ code fragments in which actual system behavior is specified. The translation of C++ code is static, which means that most "coloring" is removed from the code, i.e. iteration or selection statements are identified and translated, but their conditions are not translated into the Petri net.

This section presents twelve different patterns with which a RoseRT model can be translated into a Petri net. The patterns are divided into three groups: the structural, the behavioral, and the C++ code translation patterns.

4.1 Structural translation

The structural patterns are concerned with the translation of capsule roles, ports and the connections between them. In the examples below, the ports all realize the same protocol. This protocol has in signals *A* and *B* and out signals *Y* and *Z*.

Pattern SD01: This pattern is used to create Petri net places for a port realizing a certain protocol on a capsule role. For every port $p \in P_{cr}$, where P_{cr} denotes the set of ports modeled on a certain capsule role *cr*, the set of signals S_p that can be either sent to or received from port *p* is defined by its protocol. The signals that can be sent are denoted as $out(S_p)$, the signals that can be received as $in(S_p)$. For every signal $s \in S_p$ a uniquely labeled place is created. The set of unique labels, and thus the set of places, is obtained by taking the union of (1) and (2) where $c(\ldots)$ denotes the cardinality of a port or capsule role, $pr(p)$ a port role of a port *p*, and $n(\ldots)$ the name of an element. Every label ends with either *.in* or *.out* to distinguish between the sending and receiving of signals.

$$\bigcup_{i=1}^{c(cr)} \bigcup_{p \in P_{cr}} \bigcup_{j=1}^{c(pr(p))} \bigcup_{s \in out(S_p)} n(cr).i.n(p).j.n(s).out \tag{1}$$

$$\bigcup_{i=1}^{c(cr)} \bigcup_{p \in P_{cr}} \bigcup_{j=1}^{c(pr(p))} \bigcup_{s \in in(S_p)} n(cr).i.n(p).j.n(s).in \tag{2}$$

Pattern SD02: Pattern SD01 denotes the translation of a structure diagram for a capsule role. Pattern SD02 adds the connections between different capsule role translations, i.e. between the places representing the port roles. For every two connected port roles *pr*1 and *pr*2 ∈ *PR*, i.e. *connected*(*pr*1,*pr*2) = *True* ∧ *pr*1 ≠ *pr*2, where *PR* denotes the port roles in the RoseRT model, transitions from out to in signals are created between the places representing them.

Figure 1 depicts two capsule roles with a connection between them, all elements have a cardinality of one. Figure 2 depicts the Petri net translation.

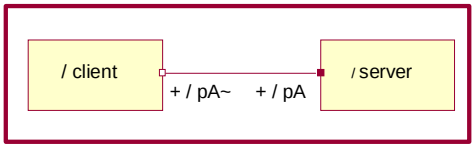

Figure 1. SD02 RoseRT view.

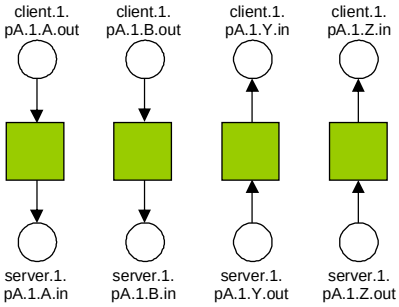

Figure 2. SD02 Petri net view.

4.2 Behavioral translation

The behavioral patterns are divided in five pattern groups: state and point translation (prefixed by SP), regular transitions (RT), border transitions (BT), choice points (CP), and composite states (CS). For reasons of comprehensibility, the labels identifying port translations are shortened in this section to a port identification and a signal name.

Pattern SP01: Behavioral translation starts with the creation of places representing the relevant states and points in a state diagram. For all simple states modeled in the state diagram a place in the Petri net is created. The initial point is also translated into a Petri net place, labeled the same way as a simple state. Choice points in the state diagram are also translated as a place.

Junction points on simple states are not translated: they are immediately mapped onto the simple state they are defined on. Junction points on composite states are translated into places in the Petri net as they are required to model a transition path from one simple state to another.

Pattern RT01: Pattern RT01 describes the translation of a state transition *u* between two states *SA* and *SB*. State transition *u* might be triggered by a set of triggers *E*, and execute code *c*, resulting in a bag of out signals *C*. An extra place, p_u, is added to indicate that state transition *u* can be fired. For each trigger $e \in E$ we add a transition u_e with input places *SA* and *e*, and output place p_u. If the set *E* is empty, a transition

labeled τ is added between *SA* and p_u. State transition u is translated as a transition labeled u, with input place p_u and output place *SB*, and for each out signal $e \in C$ to the corresponding place e with arc weight $C(e)$.

Figure 3 depicts the RoseRT view of this pattern extended with notation that makes visible which triggers are defined for a transition and which signals are being generated.

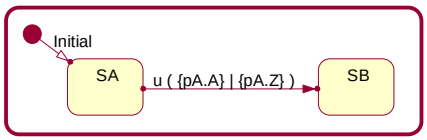

Figure 3. RT01 RoseRT view.

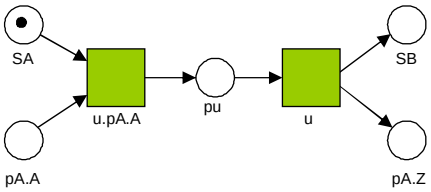

Figure 4. RT01 Petri net view.

Pattern BT01: Border transitions are modeled on composite states, their execution is initiated from simple states in the composition. Therefore, they are translated as self-transitions using pattern RT01 on each place representing a simple state of the composition. If a trigger is already handled by a transition defined on a state lower in the hierarchy, the trigger is ignored in the translation.

An example of a border transition is depicted in Figure 5 and its translation in Figure 6. Transition *v* can be initiated from both states *SA* and *SB*.

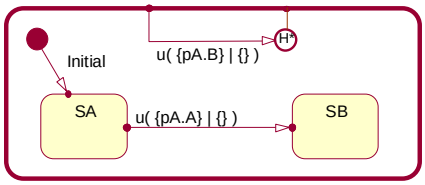

Figure 5. BT01 RoseRT view.

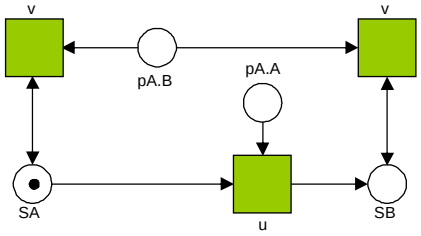

Figure 6. BT01 Petri net view.

Pattern CS01: Composite state occurrences in RoseRT state machines are translated using patterns CS01 and CS02. Pattern CS01 translates composite states that are initialized via an incoming state transition ending on a non-terminating junction point, from which a state transition into the composite state is started.

Each state transition in this pattern can be translated using RT01. State transitions to a non-terminating junction point are translated with RT01, where the end state now is a non-terminating junction point. Accordingly, the junction point is used as the start state when translating the outgoing transition.

Figure 7 shows an example of such a composite state. State *SB* is a composite state that is composed of simple states *SK* and *SL*. The state machine of *SB* is started with the ending of state transition *u* on a non-terminating junction point, *JP*1. From this junction point state transition *w* is taken to state *SK*. The composite state is left with state transition *y* followed by state transition *v*. The translation is depicted in Figure 8.

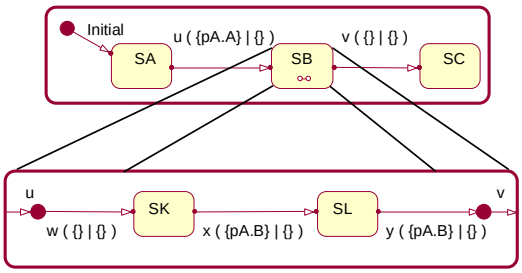

Figure 7. CS01 RoseRT view.

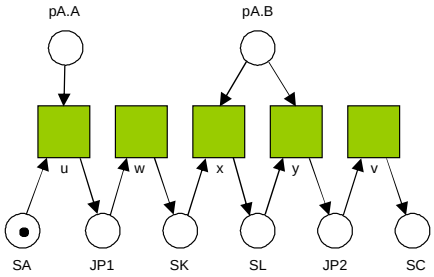

Figure 8. CS01 Petri net view.

Pattern CS02: When a state transition to a composite state ends on a terminating junction point, the initial transition is triggered. The translation follows CS01 but an additional transition is added between the states representing the junction point and the initial point.

4.3 C++ code translation

The execution of a C++ code block on a state transition *c* produces a bag of out signals *C*. To determine this bag, the C++ code has to be analyzed. The analyzed code is translated into an open workflow [23] state machine Petri net that produces all possible outcomes.

As mentioned before, most "coloring" is removed from the code, i.e. iteration or selection statements are identified and translated into the Petri net, but their conditions are not. Thus we only consider selection, repetition, composition, and signal sending constructs: the skip action is denoted by τ, a send action of signal *a* is denoted by *a*!, the composition of two statements is denoted by $C \bullet C$, a selection statement is denoted by $C+C$, and the repetition statement is denoted by C^*. Then the grammar is denoted as: $C \rightarrow \tau \mid a! \mid C \bullet C \mid C+C \mid C^*$.

Except for the composition, which is translated by fusing the start and end place of a translated statement, all grammar elements are translated using a pattern. The translated state transitions in the Petri net are refined with the application of these patterns, according to the structure depicted in Figure 9. The cloud represents the translated C++ statements.

Figure 9. Transition refinement.

Pattern CA01: Statements not influencing control flow and not sending signals are translated to a τ transition.

Pattern CA02: Pattern CA02 translates statements sending a signal. Sending a signal *A* over port *pA*, in C++ code pA.A().send(), is translated to the Petri net depicted in Figure 10.

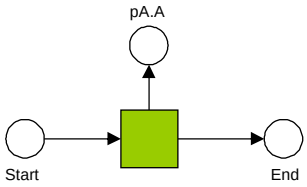

Figure 10. CA02: Sending signal *A* over port *pA*.

Pattern CA03: Pattern CA03 translates language element *C+C*, the selection statement. An example selection statement in which two alternatives and a default alternative are available might be: if(exp){…} else if(exp){…} else{…}. Presence of a default alternative implies that exactly one alternative is always selected. Its translation is depicted in Figure 11. Note that it is assumed that no signal send actions are present in selection expressions, which is in accordance with the coding guidelines of the cases in section 6.

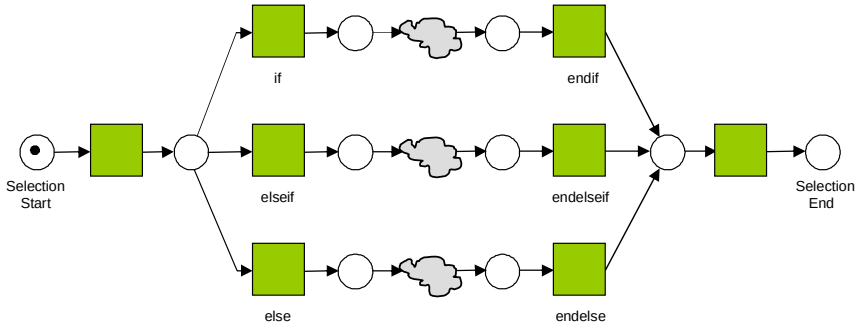

Figure 11. CA03: Translation of if-elseif-else construction.

Pattern CA04: Pattern CA04 defines the translation of the repetition statement. Two patterns to translate the repetition statement are introduced. The first pattern CA04.1 is a straightforward translation, where the number of iterations is unbounded. It is depicted in Figure 12. The second pattern CA04.2, which is depicted in Figure 13, translates the repetition statement with a loop limited by a predefined upper bound. Place p_7 is used to guarantee that no iterations are being executed in parallel.

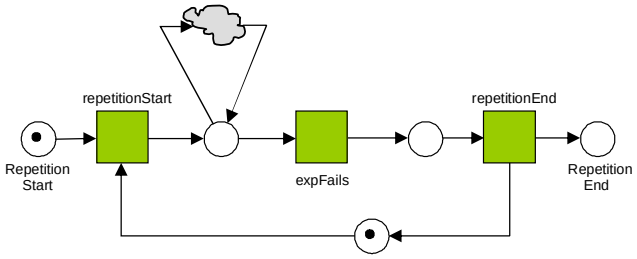

Figure 12. CA04.1: Unbounded repetition.

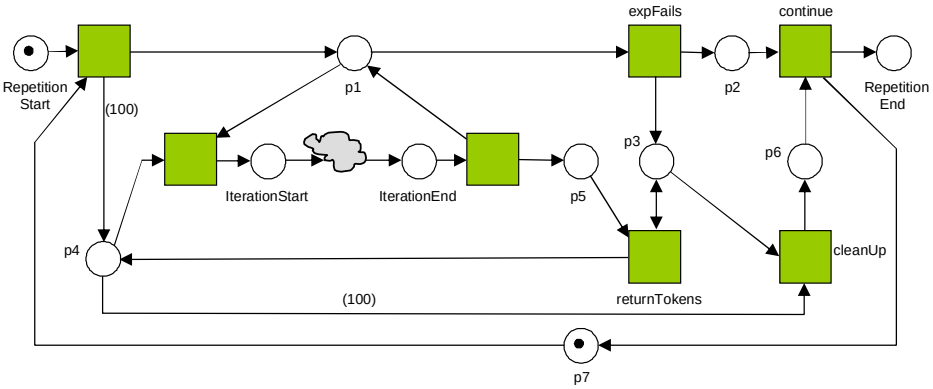

Figure 13. CA04.2: Bounded repetition.

5 Optimizations and reductions

When performing analysis on real life models, the need for reductions and optimizations arises, as the number of places and transitions in the generated Petri nets easily reaches 100.000. This section presents a number of translation optimizations and translation reductions to reduce the number of places and transitions created in the Petri net during the translation.

5.1 Optimizations

The optimizations to reduce the number of places and transitions used in a translation are presented as heuristics.

H01: Disregard unconnected ports: Connections for ports unconnected at design time can not be determined by the translation, therefore no places representing the signals of such a port need to be created. All RoseRT transition triggers defined to trigger on a signal received on an unconnected port are disregarded in the translation.

H01 is one of the optimizations that are always applied. Without its application deadlocks might occur since transitions would never be enabled as tokens in the signal places would never arrive.

H02: Disregard unconnected capsule roles: An unconnected capsule role is defined as a capsule role for which none of its ports is connected. With the exception of the top capsule role, unconnected capsule roles can be disregarded in the translation as no signals can be sent to or received from them.

H03: Relay ports disregarded: The only purpose of relay ports is to forward signals up or down the hierarchy. Signals can only be sent or received at end ports, therefore it is sufficient to translate and connect end ports only.

H04: Translation of in signals disregarded: One port instance can only be connected to exactly one other port instance. The application of pattern SD02 creates two places and one transition for every connection. The Murata [24] reduction rules allow this to be reduced to a single place.

H05: Limited capsule role cardinality: Low level capsule roles are mostly dynamically instantiated: they are instantiated at the bottom of the capsule role hierarchy and mainly represent the software implementations of hardware sensors. At design time it is often not exactly known how many instances will be created, this unawareness is translated into a RoseRT model by specifying a theoretical upper bound X for the cardinality of such a capsule role. During translation it results in the capsule role being translated X times. Before translation a new (lower) upper bound might be specified.

H06: Limited port cardinality: An implication of specifying theoretical capsule role cardinality upper bounds is that all ports that might be connected to these instances need a cardinality equal to or greater than this upper bound. They might even be higher since it is possible that connections to multiple instances of different capsule roles are modeled, therefore an optional upper bound limiting the number of port instances may be specified prior to a translation.

H07: Non-behavioral border transitions not regarded: Border (self-)transitions of which the action code does not contain send action statements do not change the active state. This means that translation is not necessary. Instead, it is sufficient to generate a dummy transition τ that consumes the token from the trigger place.

5.2 Reductions

To reduce the number of places and transitions involved in code translations, five reduction rules have been defined. These rules are denoted in R01-R05:

R01:	$a!b!$	$\leftrightarrow$	$(a,b)!$
R02:	$a!\tau b!$	$\leftrightarrow$	$a!b!$
R03:	$a!(b!+c!)$	$\leftrightarrow$	$(a!b!) + (a!c!)$
R04:	$a!(b!)^*c!$	$\leftrightarrow$	$a!c!(b!)^*\tau$
R05:	$a!(b!c!^*)^*\tau$	$\leftrightarrow$	$a! + a!b!(b! + c!)^*\tau$

An example application of reduction rule R01 through R03 is graphically depicted in Figure 14 and Figure 15. The example could be the translation of a random action code block in which two selection statements are executed after each other.

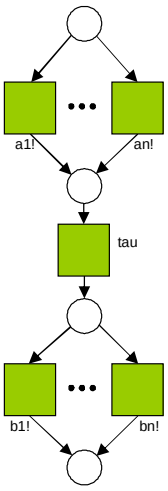

Figure 14. Unreduced example.

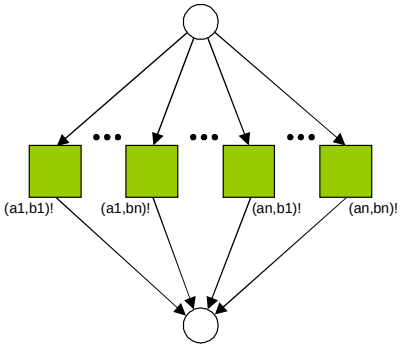

Figure 15. R01 to R03 applied.

Structural reductions can also be applied following the translation of a RoseRT model into a Petri net. The Yasper PNML libraries currently also contain [25] the reductions as defined by Murata [24], Desel and Esparza [26], and Berthelot [27]. A minimal set of reductions that has to be applied for maximal effect are the abstraction, self-loop places, and parallel places rules as defined by Murata and the identity transition and identical transition rules defined by Berthelot. This minimal set of reductions has been implemented as a last optional step in the Petri net generation process.

6 Case study

For practical usage, it is important that the analysis results can be visualized in the original RoseRT model, thereby hiding the underlying Petri net from the developers and enabling them to trace the results in a familiar environment. UML sequence diagrams are the means by which developers are enabled to trace results. A result, defined as a Petri net trace ending in a violation of the property being checked, is

translated into such a diagram. The, automatically generated, diagram depicts in what order which messages have been exchanged prior to the violation of the property.

Currently, verification (state space analysis) of the translated RoseRT models is performed using LoLA [28]. In theory, all properties of which violation returns a trace, that are supported by LoLA, can be verified.

The architecture of the tool set used to perform the Petri net analysis of RoseRT models is depicted by the Petri net in Figure 16. The RRTDataExtractor component of the RoseRTAnalyzer creates a meta model from the RoseRT model. The RRT2PNML component generates a Petri net from this meta model, using a.o. the PNML libraries of Yasper [29]. A LoLA instance analyses this Petri net. This analysis evaluates to either true or false. In the false case a counterexample trace is generated, that is translated back into the RoseRT model as a RoseRT message sequence chart by the Trace2RRT component of RoseRTAnalayzer.

The LoLA instance is created by setting up compilation parameters that are used in the LoLA source code to compile a specific version of LoLA, optimized for the chosen analysis method.

The Microsoft .NET platform has been used for the development of the RRTDataExtractor, RoseRTAnalyzer, and Trace2RRT components, with C# as programming language. Storing the meta model into a Database Management System (DMBS) has been considered, but as the meta model merely serves as an intermediary format, it is not worth the effort to create a database mapping when the .NET streaming capabilities provide a ready-to-use efficient storage method.

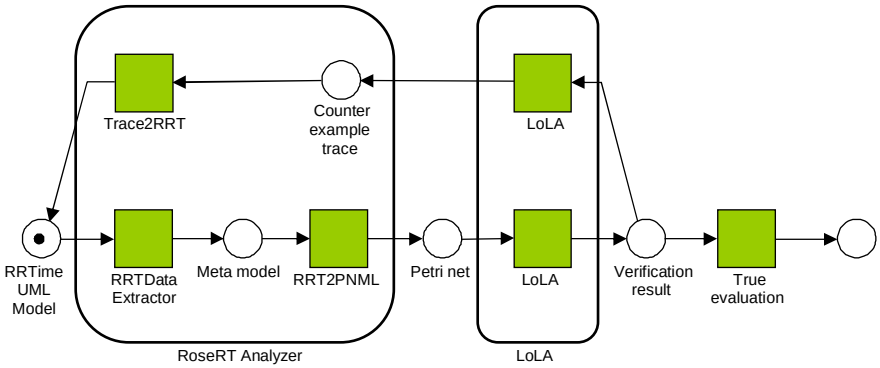

Figure 16. Petri net view on the RoseRT analysis process.

6.1 Coffee Machine case

As a first case we used a slightly adapted version of the coffee machine example that comes with the RoseRT tool set. It consists of six capsule roles. The coffee machine has been analyzed for the absence of deadlocks, as well as for the presence of a well-defined end state.

Deadlock analysis on the generated Petri net revealed the presence of a possible deadlock in the system. A sequence of 27 transitions forms the counterexample trace,

of which 21 transitions initiate the sending of a signal. Five capsule roles are involved in the generation of the deadlock. Analysis of the trace revealed that the deadlock trace actually was an expected one, and not a design flaw. The path leads to a system deadlock state in which the active state of capsule role "drivers" is "Done". It is correct that no further transitions can be taken from there: this use case only considers the implementation of exactly one brew cycle, i.e. it is implemented to end in state "Done".

The coffee machine example is defined such that capsule role "drivers" is initialized to simple state "Waiting" and, after one brew cycle, ends up in simple state "Done". However, LoLA analysis method AMC2 (which verifies that from a given marked place p_1 it is always possible to reach another given place p_2) with "Waiting" as start state and "Done" as end state generated a counterexample trace. A short inspection revealed that the trace was generated as a result of not being able to model the message queue in the Petri net. As a last test, deliberately introducing an error in the coffee machine, e.g. by commenting a send() statement in a transition, indeed produces a counterexample trace.

6.2 Copier case

Our ultimate goal is to verify properties of a real life model that defines the embedded control software of a high speed digital copier. The model chosen for analysis contains all the elements that are actually available on the copier engine, i.e. everything between hardware sensors at the lowest level and page commands at the highest level. It consists of 272 capsule and 2581 state definitions.

Generation of the meta model took quite some time for this model, around three hours. (The analysis was conducted on a 2.4 Ghz processor with 2 Gigabytes of RAM.) Profiling the analyzer program revealed the Component Object Model (COM) layer between the RRTEI library and the actual RoseRT program being the major reason for this lack of performance. Translation of the meta model into PNML format never finished (at least not within 92 hours). Profiling showed that the PNML libraries gradually slowed down the process as the number of places went over 30 thousand.

After the failure to translate and analyze the complete RoseRT model, we tried to analyze a subset, about one-third of the total size, that only models the scanner part of the copier. The meta model was generated in about one hour. The translation of the meta model into PNML format, with optimizations being applied, took three hours. Table 1 shows the size of the resulting Petri net, with and without optimizations, compared to the simple coffee machine example. The larger the code fragments on a transition, the larger the gains in place and transition reductions, as these are defined on the C++ code fragments.

After generation of the Petri net it was, naively, tried to perform a deadlock analysis. The LoLA tool ran for 12 hours without generating counter examples. Investigation of the model learned that many busy-waiting processes are modeled in the system. Obviously, the presence of busy-waiting loops prohibits the presence of deadlocks in the system.

A number of instances were defined for three LoLA analysis tasks, named AMC1, AMC2, and AMC3. AMC1 verifies whether a marked place will always, i.e. over all subsequent paths, become marked again. Such a check is often needed in real-time systems as these usually define some idle state from which it is possible to perform a task. After execution of the task the system should always return to this idle state and wait for an event that starts a new task. As already mentioned, AMC2 verifies that from a given marked place it is always possible to reach another given place. AMC3 verifies that from a given marked place it is always possible to reach one out of two given end places. These usually denote the desired end state and an "error" state in the engine control, the latter for example to be reached in case of a paper jam. The verification of the CTL formulas on a Petri net of this size made LoLA run out of memory in less than 10 seconds. Experimenting with the LoLA setup parameters and reductions techniques did not bring any solution.

Finally, we also considered a small subset, the stubbed version of a single component from the total copier model. Translation of this model into a Petri net is only a matter of minutes. The stubs are modeled as use cases, thus the system is modeled such that it ends in a deadlock state when all use cases have been executed. One AMC1 instance has been defined, its verification generated a counter example trace. Inspection revealed that it was actually the lack of runtime code analysis that generated the trace. The violation of the formula was caused by a sequence of if statements being taken that would not have been executed in a real environment. The component allowed for the definition of an AMC2 instance concerning the initialization process of the component. Its execution evaluated to true, so it can be concluded that initialization of this component has been correctly modeled. Another AMC2 instance evaluated to false for the same reason as the AMC1 analysis task evaluated to false.

Table 1. Optimization and reductions results.

	Coffee machine		Océ model	
Not reduced and not optimized	283 274	places transitions	123216 82480	Places transitions
Not reduced and optimized	283 274	(≈ −0 %) (≈ −0 %)	16891 18150	(≈ −85 %) (≈ −78 %)
Reduced and optimized	218 207	(≈ −23 %) (≈ −24 %)	6906 6091	(≈ −94 %) (≈ −93 %)
Reduced, optimized, and post translation reductions.	74 64	(≈ −74 %) (≈ −77 %)	6100 3394	(≈ −95 %) (≈ −96 %)

7 Conclusions and future work

We have defined a translation of Rose RealTime models into Petri nets and a translation of the analysis results back into the original RoseRT model. The initial translation resulted in Petri nets far too big for analysis purposes, therefore optimization and reduction rules have been implemented. Optimized translations have

been successfully applied on a coffee machine model and, partly, on a real life model of a high speed copier. Together, the two translations may serve as a basis for RoseRT model verification using well known methodologies from Petri net theory. However, handling large models is still a problem.

Future work consists of two parts. In the near future more research effort will be put in determining analysis methods relevant to RoseRT model verification. In the long run it will be investigated whether a more thorough code analysis is feasible, i.e. adding some of the runtime code model behavior to the translation.

References

1. Dohmen, L., Somers, L.: Experiences and lessons learned using UML-RT to develop embedded printer software. In: Proceedings Product Focused Software Process Improvement (PROFES 2002), Rovaniemi, Finland, December 9-11, LNCS Volume 2559, pp. 475-484, Springer, Berlin (2002)
2. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley, New York (1994)
3. IBM: Rational Rose RealTime, http://www.ibm.com/ (2007)
4. I-Logix: Rhapsody, http://modeling.telelogic.com/ (2007)
5. Apvrille, L., de Saqui-Sannes, P., Lohr, C., Senac, P., Courtiat, J.P.: A new UML profile for Real-Time system formal design and validation. In: Proceedings of the 4th International Conference on the Unified Modeling Language. Modeling Languages, Concepts, and Tools. Toronto, Canada, October 1-5, 2001, LNCS Volume 2185, pp. 287-301, Springer, Berlin / Heidelberg (2001)
6. Object Management Group: Unified modelling language: Superstructure, version 2.0 (2006)
7. Ploeger, S., Somers, L.: Analysis and verification of an automatic document feeder. In: Proceedings 22nd Annual ACM Symposium on Applied Computing (SAC'07), Seoul, Korea, March 11-15, 2007, pp. 1499-1505, ACM Press (2007)
8. Leue, S., Mayr, R., Wei, W.: A scalable incomplete test for the boundedness of UML RT models. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), LNCS Volume 2988, pp. 327-341, Springer, Berlin / Heidelberg (2004)
9. Bernardi, S., Donatelli, S., Merseguer, J.: From UML sequence diagrams and statecharts to analyzable Petri net models. In: Proceedings of the 3rd international workshop on Software and performance (WOSP 2002), pp. 35-45, ACM Press (2002)
10. Merseguer, J., Campos, J., Bernardi, S., Donatelli, S.: A compositional semantics for UML state machines aimed at performance evaluation. In: Proceedings of the Sixth International Workshop on Discrete Event Systems (WODES 2002), pp. 295-302, IEEE Computer Society (2002)
11. Trowitzsch, J., Zimmermann, A., Hommel, G.: Towards quantitative analysis of real-time UML using stochastic Petri nets. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005) - Workshop 2, pp. 139b, IEEE Computer Society (2005)
12. Saldhana, J., Shatz, S., Hu, Z.: Formalization of object behavior and interactions from UML models. International Journal of Software Engineering and Knowledge Engineering, 11(6), pp. 643-673 (2001)
13. Eichner, C., Fleischhack, H., Meyer, R., Schrimpf, U., Stehno, C.: Compositional semantics for UML 2.0 sequence diagrams using petri nets. In: Proceedings 12th International SDL

Forum: Model Driven Systems Design (SDL 2005), LNCS Volume 3530, pp. 133-148, Springer, Berlin / Heidelberg (2005)
14. Baresi, L., Pezzè, M.: On formalizing UML with High-Level Petri nets. In: Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets. LNCS Volume 2001, pp. 276-304, Springer, Berlin / Heidelberg (2001)
15. Gehrke, T., Goltz, U., Wehrheim, H.: The dynamic models of UML: Towards a semantics and its application in the development process. Hildesheimer InformatikBericht 11/98, Institut für Informatik, Universität Hildesheim (1998)
16. S. Kuske, A.: A formal semantics of UML state machines based on structured graph transformation. In: Proceedings of the 4th International Conference on the Unified Modeling Language: Modeling Languages, Concepts, and Tools (UML 2001), LNCS Volume 2185, pp. 241-256, Springer, Berlin / Heidelberg (2001)
17. Grosu, R., Broy, M., Selic, B., Stefanescu, G.: Towards a calculus for UML-RT specifications. In: Proceedings 7th OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications, Technical Report TUM-I9820, Technische Universität Munchen (1998)
18. Reggio, G., Astesiano, E., Choppy, C., Hussmann, H.: Analysing UML active classes and associated state machines - a lightweight formal approach. In: Proceedings Fundamental Approaches to Software Engineering (FASE 2000), LNCS Volume 1783, pp. 127-146, Springer, Berlin / Heidelberg (2000)
19. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science, 55(3), 2001
20. Yau, D.: Model Checking of Models of Real-Time Systems. Master's thesis, Queen's University at Kingston (2006)
21. Mili, H., Mili, A., Yacoub, S., Addy, E.: Reuse-Based Software Engineering: Techniques, Organization and Measurement, John Wiley & Sons (2002)
22. Reisig, W.: Petri Nets: An Introduction. Volume 4 of Monographs in Theoretical Computer Science, Springer, Berlin (1985)
23. Schmidt, K.: Controllability of open workflow nets. In: Enterprise Modelling and Information Systems Architectures, Lecture Notes in Informatics (LNI), Volume P-75, pp. 236-249, EMISA, RWTH Aachen, Köllen Druck+Verlag (2005)
24. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE, 77(4), pp. 541-580 (1989)
25. van der Werf, J.: Analysis of well-formedness and soundness by reduction techniques and their implementation. Master's thesis, Technische Universiteit Eindhoven (2006)
26. Desel, J., Esparza, J.: Free Choice Petri Nets. Volume 40 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge (1995)
27. Berthelot, G.: Verification de Reseaux de Petri. These de 3eme Cycle, Université Pierre et Marie Curie, Paris (1978)
28. Schmidt, K.: LoLA: a low level analyser. In: Proceedings 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), LNCS Volume 1825, pp. 465-474, Springer, Berlin / Heidelberg (2000)
29. Hee, K. van, Oanea, O., Post, R., Somers, L., Werf, J. van der: Yasper: A Tool for Workflow Modeling and Analysis. In: Proceedings 6th International Conference on Application of Concurrency to System Design (ACSD 2006), Turku, Finland, June 28-30, 2006, pp. 279-282, IEEE, Brussels (2006)