Verification of an industrial rule-based manufacturing system using REX

AnnMarie Ericsson[1], Mikael Berndtsson[1], Paul Pettersson[2], and Lena Pettersson[3]

[1] Department of Humanities and Informatics, University of Skövde,
[2] School of Innovation, Design and Engineering, Mälardalen University
[3] Volvo Information Technology, Skövde

Abstract. Formal methods are not used in their full potential for enhancing software quality in industry. We argue that seamless support in a high-level specification tool is a viable way to provide system designers with powerful and paradigm specific formal verification techniques.
Event condition action (ECA) rules can be used to model and implement reactive behavior in, for example, the semantic web. Independently of target system, the behavior of rule-based systems are known to be hard to analyze. The REX tool is a rule-based front-end to the timed automata CASE-tool Uppaal. The model-checker in Uppaal is used by REX enabling seamless support for model-checking rule-based specifications.
This paper presents experiences from modeling and verifying a system of industrial complexity as interacting rules using REX. We conclude that repeatedly performing formal analysis when constructing a system with interacting rules is a viable way of coping with the complexity of the model. Additionally, we present an implemented algorithm for optimizing the model to reduce the effect of state-space explosion.

1 Introduction

The paradigm of rule-based systems is well suited for implementing the behavior of reactive systems [1]. The dynamic behavior of rules is beneficial for describing systems in such diverse areas as health care and algorithmic trading applications. In the area of web applications, ECA rules are proposed as a suitable paradigm for implementing, for example, behavioral aware web-applications [2] and for reactive solutions in the sematic web [1].

Independently of the context of the target system, if a failure of the system causes a high cost, it must be ensured that such failures can not occur. Using rules in critical systems implies that the systems behavior must be thoroughly analyzed. However, analyzing a set of low-level rules is a complex task due to interactions between rules [3]. One rule may, for example, trigger another rule causing a chain of rule-triggerings or change the outcome of the condition evaluation of other rules. Additionally, adding, removing or changing a rule without understanding the effects can be dangerous, since changes to the rule base can introduce major errors in the system. Thus, before a change is introduced in the rule base, the effect of the change need to be thoroughly analyzed to not cause any undesired side-effects in the systems behavior.

Formal methods are mathematically based methods for specifying and verifying systems. Developers of complex applications can increase their product and process quality by utilizing a formal method [4]. Additionally, several papers propose that formal methods provide means for preventing errors from entering the system in the early phases of development [5, 6]. Nevertheless, formal methods are not used in their full potential in industry. Some of the reasons may be the high knowledge threshold one needs to pass to be able to use them and the extra time it may take to construct the specifications [7].

Several CASE tools exists supporting different types of formal methods. The drawback for our purpose is that none of them supports rule-based applications. Modeling the behavior of a rule-based system in, for example, the timed automata verification tool Uppaal is an error prone and time consuming task since the tool is not designed for modeling rule-based systems.

We address the problem of formally analyzing rule-based applications by utilizing the power of model-checking for verifying system behavior. A graphical tool (REX) is constructed [8], serving as a rule-based front-end to the timed automata CASE-tool Uppaal [9]. Rules are specified in a high-level rule-based language provided by REX. The specification is automatically transformed from REX to a timed automata representation of the rule set implying that the model-checker in Uppaal can be utilized to verify the rules.

This paper reports experiences gained from using REX in a case study where an existing industrial system is modeled and verified. The case-study object is a system for producing assembly plans for engine plants at Volvo IT in Skövde, Sweden. The chosen system has a complex behavior dependent on both values of parameters in incoming telegrams and stored values in database tables. The correctness issue of the case-study object is critical since a failure in this system stops the production plants and causes severe economical loss for the company.

The behavior of the system is modeled as a set of rules and complex events in REX. The correctness of the model is verified by formulating verification expressions in REX that are automatically executed in Uppaal. The results of our case study shows that using REX as a front-end to Uppaal is a feasible approach for model-checking a rule-based system of industrial complexity. The experiences gained from performing the case study teaches us that iteratively utilizing the ability to formally verify properties of the system during development is a viable way of coping with the complexity arising in a system built by interacting rules triggered by complex events. This paper focus on reporting the lessons learned from the case study, for a detailed description of the actual case study we refer to [10].

2 Preliminaries

An event condition action (ECA) rule executes a sequence of code (action A) if a specified condition C is true when event E occurs. The action part of the rule can be an arbitrary sequence of code and the condition is commonly a boolean expression or an SQL expression returning true or false. The triggering event can be a primitive event, i.e. a single occurrence such as an update of a database, a sensor reading or a message arriving to the system. Additionally, some systems support complex events, implying that the triggering event may be a combination of both primitive and complex events.

Events contributing to a complex event type is composed by operators such as conjunction($\wedge$), disjunction ($\vee$) and sequence (;). A complex event $E = E_1 \wedge E_2$, for example, is generated when there is an occurrence of both type E_1 and type E_2 in the event history.

2.1 REX

The Rule and Event eXplorer (REX) [8] is a rule-based front-end to the timed automata CASE tool UPPAAL [9]. A system and its requirement properties are specified in terms of rules in REX. The rules are automatically transformed to a timed automaton representation while the requirement properties to check are transformed to an expression in computational tree logic (CTL) and a designated test-automaton [11]. REX automatically runs the model-checker in UPPAAL and forwards the result from the model-checker to its user.

Ideally, a model of the system is created in REX to support early detection of errors in the development phase. Additionally, in the maintenance phase, the model can be used to ensure that changes in the system, i.e. removing, changing or adding rules and events, does not cause any undesired side-effects.

Specifying models in REX The item types (rules, events, conditions, actions, dataobjects, data tables) supported by REX are specified in graphical property tables. Relations between items of the same type are shown in tree structures [8]. The operators currently supported in REX includes conjunction, disjunction, sequence, non-occurrence, times, delay and sliding window. The times operator generates an event E when a specified event E_1 has occurred n times, the delay operator generates a specified event E t time units after the occurrence of a specified event E_1.

In addition to the item types, a scenario editor provides ability to specify different scenarios. A scenario may, for example, be that event E_1 occurs at time 2 with event parameter x=4; event E_2 occurs at time 4 and event E_5 occurs at time 6. In the *default* scenario, primitive events occur at any time in any order. However, this behavior is likely to cause a state space explosion in the timed automaton model. Providing the user with ability to create scenarios gives the user means to reduce the search space for the model checker and to focus the verification on specific problems.

Specifying verification properties in REX Application specific properties, such as, "can rule R_1 execute before rule R_3", are specified using property templates. Two useful examples of property templates are *Universality* and *Existence*. The *Universality* pattern is suitable when checking that the system will *always* satisfy a specific property. It is, for example, useful when checking that a correct final state will always be reached. The *Existence* pattern is suitable when checking if an erroneous state can be reached. Each pattern has a *scope*, defining the part of execution when the property must hold. Scope *Globally*, for example, defines that the property must hold during the entire execution while scope *P Before Q* defines that property P must hold before Q holds [12].

A property template may, for example be "EXISTS Predicate P BEFORE predicate Q". The user defined parts of the templates are predicates P,Q,R and S. In the example, if predicate P="Event E occurs" and predicate Q="Rule R executes and variable x == 5". Running this property pattern checks whether event E can occur before rule R executes simultaneously as x == 5. For an in-depth discussion of how the patterns are used in REX we refer to [11].

2.2 Uppaal

Uppaal is a toolbox for modeling and analyzing specifications built on the theory of timed automata [13] extended with additional features. The tool is developed jointly by Uppsala University and Aalborg University [9]. A model in Uppaal is built by a network of synchronizing timed automata. Each timed automaton simulates a process which is able to synchronize with other automata.

Given an Uppaal model of a system, model-checking can be performed by specifying the properties needed to be checked in timed CTL. The property can quantify over specific states or over a trace of states. It is, for example, possible to ask if variable x will always have a value less than 5 in location S_1 in process P ($A[]P.S_1$ and $x < 5$) or if it is possible to reach location S_2 within 3 time units ($E <> P.S_2$ and $globalClock < 3$). In this paper, Uppaal syntax is used to describe property expressions.

2.3 TUR

The group Manufacturing Production Systems at Volvo IT Skövde is responsible for development and maintenance of IT solutions, mainly in the areas Supply Chain and Production Planning, for the engine plants in the car factory. The system TUR is implemented and maintained by this group.

The main task of TUR is to convert a high-level assembly plan for items (different types of engines) to be manufactured to a detailed ordered plan for each sub-item (camshaft, crankshaft, etc.) to be constructed or delivered by each assembly-line. The detailed assembly-plan contains an ordered sequence of items to be produced or delivered by each assembly-line.

To avoid lack of items, the assembly plans provided by TUR must be coordinated. The order of items manufactured by different plants must be correlated with each other since assembly lines combining items from different plants must receive all items contributing to their item in correct time and order.

The system TUR is a sequential program with a rule-based semantics. The control flow of TUR was modeled as rules and complex events in REX. The model created in this project consists of 63 rules, 12 complex events, 50 primitive events, 30 data variables and 45 event parameters separated on different events. Some of the complex events are composed by other complex events in an event hierarchy. One of the complex events with depth three, and four of the complex evens have depth two. The rules are heavily interacting with each other, i.e. the execution of most rules trigger one or several other rules.

Additionally, the specification consists of 12 modeled database tables with 5 rows filled with values. Moreover, 20 of the actions performed by the rules consist of a function modeling an SQL SELECT expression over one or more of the modeled tables. The verification of the model was performed by constructing verification expressions in REX that were automatically executed in the model-checker provided by Uppaal.

The main problem in verifying TUR was that the behavior of the system to a high degree is dependent on values in database tables. Checking all execution paths with all different combinations of initial database values is an unsolvable problem for any model-checker. A manual analysis were performed to find a set of initial database states, each representing one of the observable behaviors. The observable behaviors are identified to be a correct execution with no error message for each possible telegram arriving to the system and different types of error messages that can be reported for each type of telegram. This approach resulted in 20 different initial database states, each of them were checked by 34 different verification expressions.

3 Lessons learned

The experience of using REX for modeling a system larger than toy-size pin-pointed abilities of REX that are useful in a tool for developing rules. However, it also revealed some areas where REX can be improved. The following subsections discuss what features a modeling and verification tool for rule-based system need to support, based on experiences gained from this project.

3.1 Performance

The performance of REX and Uppaal is dependent on the level of non-determinism in the model. The non-determinism of the TUR system origins in the fact that the behavior is dependent on values in database tables. The approach taken for overcoming this problem is to trade memory consumption with time. Instead of checking one large non-deterministic model, several rather deterministic models are checked. In the case-study, each model was verified in less than 1 second. One can argue that reducing the number of checked initial states to 20 is more testing than model-checking. However, the model-checker can still find erroneous traces that would have been unrevealed by a testing approach although some errors might be uncovered. Since checking for all different initial database states is impossible, one have to put a lot of effort in finding a representative set of initial database states in systems like TUR.

The approach with scenarios is a trade off between time and memory since dividing a specification into several more deterministic models consumes less memory for checking each model. However, it takes time to run, create and administrate a large set of models. Additionally, some errors may not be reveal if we don't check all possible combinations of initial states.

The behavior of TUR depends on values in database tables, however, the problem with state space explosion is similar for systems where the input to the system is a large number of different events occurring in nondeterministic order. If the number of different events that can occur in any order is n, then the number of orders that must be checker is n!. In REX, preprocessing is performed to analyze the rule set and reduce the number of events in the input domain. The preprocessing complements the approach of trading memory for time.

Preprocessing in REX Transforming the entire set of rules from REX to Uppaal will in many cases result in an unnecessarily big automaton. Some of the rules in the rule set will not affect the outcome of running the query independently of if they are triggered or not. In order to reduce the search space for the model checker, REX removes rules and events not related to the queried property before transforming the rule set.

Since properties in REX are expressed in terms of rules and events with clearly specified relations, it is possible to preprocess the model in order to reduce its size. If, for example, property P is defined to check whether rule R_1 can be executed before rule R_2, only rules and events that can affect when R_1 or R_2 will execute is included in the automaton generated to verify P.

We utilize the knowledge of relations between rules given by the triggering (TG) and activation (AG) graphs [14]. However, the representation of an arc $a_{i,j}$ in the activation graph is extended to represent any change in the condition evaluation of rule R_j when R_i is executed, not just a change from false to true.

Let the set N represent all rules in the rule set and R represent the set of all rules included in property P. The nodes in the triggering and activation graphs are representing the rules in set N. For each node $n_i \in$ N, add all rules representing nodes that can reach n_i in TG or AG to the set R. After preprocessing the set of rules and the property to check in REX, the set R contains all rules which directly or transitively triggers or activates some of the rules in the property. All rules in R and all events that may trigger some rule in R are included in the transformed model.

Assume, for example, a set of rules $N = \{R_1, R_2, R_3, R_4, R_5, R_6\}$. For $i = 1$ to 6, each rule $R_i \in N$ are triggered by an event E_i. The triggering and activation graph of the rule set is shown in Figure 1, filled lines represents triggerings and dashed lines represents activations. Assume that property $P = R_2$ *always executes before* R_1 must be checked and assume a non-deterministic environment where all events E_1, E_2, E_3, E_4, E_5 and E_6 can occur in any order. Checking property P in Uppaal without preprocessing the set of rules results in 6! different occurrence orders for the specified events. This implies that there are 6! different occurrence orders for Uppaal to check.

If the preprocessor is utilized, the set R initially contains R_1 and R_2. After preprocessing the rule set, $R = \{R_1, R_2, R_3\}$. Hence, if the rule set is preprocessed, the set of rules transformed to Uppaal is reduced to R_1, R_2, R_3 and the environment automaton only consist of E_1, E_2, E_3 resulting in 3! different occurrence orders.

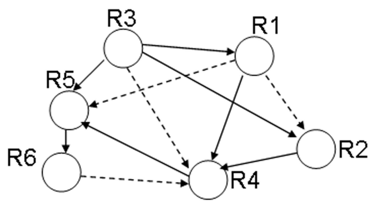

Fig. 1. Triggering and activation graph.

In the case study, the preprocessing did not have a significant impact of the performance result since the non-determinism mainly originated in data from database tables and not in events occurring in non-deterministic order. However, it may be possible to improve the algorithm for preprocessing rules. If a tuple in a table is never read by any rule in the rules resulting from the preprocessing, it should be possible to remove the table from the rule set before it is transformed to Uppaal and in that way reduce the search space even further.

3.2 Support for automatic verification

Support for verification is the main aim of the REX tool. The ability to formulate and run verification questions in every stage of the modeling phase, even when the system is not yet executable, is invaluable when designing a set of rules. The developer need to think very hard about the behavior of the system to be able to formulate verification expressions and define expected results. By running the expressions in the verifier, the developer retrieves feedback about whether the set of rules behave as intended. If an error is introduced at some stage of development it may be revealed when the verifier is executed.

During the case study, the ability to model-check the behavior of the model in Uppaal was repeatedly used. The experience from this approach is that several errors were detected immediately after entering the model. As an example, a new rule R_1 was introduced in the model. The action part of R_1 generate event E_1, E_1 triggers rule R_2 which is perfectly right. However, E_1 is also the initiator of the complex event E_3. The fact that E_1 initiates E_3 was not observer by the developer, but revealed in the model-checker since it found an execution path where rules triggered by E_3 could execute in wrong order due to R_1.

Lots of research concerning analysis of wether a set of rules is terminating or not exists within the area of active databases [15–17]. However, when a complex system, such as TUR, is modeled where the rules are heavily interacting with each other, guarantees for termination and confluence is not sufficient. The issue of ensuring that the system behaves as intended requires that application specific properties can be verified *iteratively* during development. If verification is performed for the first time when the model is finished, the errors may be hard to correct in a rule-based system with interacting rules. The complexity of the behavior of the rule base quickly increases with its size and if the rules are interacting with each other it may be hard to correct an erroneous rule without causing undesirable side-effects.

3.3 Support for simulating rule behavior

The simulator in REX uses Uppaal to retrieve step by step information about the execution. Each step is a step of execution in the timed automata representation of the rule set. The information is parsed to rules and events before it is presented to the user. REX supports the ability to exclude single rules and events or groups of rules and events from being visualized in the simulator in order to focus the simulation on specific items. In each step, the user can choose which rule to execute in the next step. Additionally, traces retrieved from executing a verification expression can be visualized in the simulator.

The simulator in REX provided good support for understanding the behavior of TUR during the case study. However, with the amount of rules that exists in the TUR model, it is hard to follow the behavior of all rules simultaneously. REX shows the state of each rule in each step in a separate color coded swim-lane. The swim-lanes are organized horizontally and scrolling is required to view all rules. The solution is good if the user want to follow the behavior of one or a group of rules, however, it is hard to grasp the behavior of the entire rule set.

4 Conclusion and Related Work

Most previous work addressing formal analysis of rule based systems concern the properties termination and confluence (e.g. [15–17]). As far as we know, no previous work addresses model-checking of application specific properties on rules with complex events.

Some previous works exist within the area of visually showing executing rules (e.g. [18, 19]). Most works only shows examples on small rule sets and it is not clear how the solutions scale when the set of rules is increasing. In the work of [20], Vizar, a 3D approach for showing rules is presented where rules can be shown in different levels of abstraction. The simulator in Vizar displays an existing trace file while REX utilizes a model-checker supporting step-wise simulation. The experience from the case study clearly indicates that it is hard to view the execution of a large set of rules in a comprehensive way. Using different views for different levels of abstractions as in Vizar may be a feasible way to improve rule visualization in practise.

4.1 Conclusion

Independently of application area (e.g. Semantic Web, manufacturing, algorithmic trading) analyzing a set of low-level rules is a complex task due to interactions between rules. This paper presents experiences from modeling and verifying a system of industrial complexity as interacting rules using REX. In particular, we have presented our experiences with respect to i) performance, ii) support for automatic verification and iii) support for simulating rule behavior using a model-checker.

The performance of a rule analysis tool is one of the crucial issues if it is going to be used in practice. Using a model-checker can potentially lead to state space explosion problems. In order to avoid such problems, we have implemented an algorithm that preprocess rule sets.

Automatic verification of a rule set in every stage of the application development, even when the system is not yet executable, provides support for coping with the complexity of interacting rules.

Simulating rule behavior visually is a good option when verifying a rule set together with domain experts. However, there are no obvious approaches to visualize large rule sets or the entire behavior of a rule set.

We argue that the approach of creating high-level tools to existing formal analysis tools is a viable way of reducing the burden of analyzing ECA rules.

Finally, we believe that automatic verification of rule sets and simulating rule sets visually desperately need to be integrated with existing software development methods and notations. To the best of our knowledge, many software projects develop ECA rules in ad hoc manners rather than developing ECA rules as a natural component of their software development process.

Acknowledgement

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS and CUGS (the national graduate school of computer science). The authors would like to thank the staff at Volvo IT Skövde for access to the system TUR and for help and support during this project.

References

1. J. J. Alferes, J. Bailey, M. Berndtsson, F. Bry, J. Dietrich, A. Kozlenkov, W. May, P. L. Patranjan, A. Pinto, M. Schroeder, and G. Wagner, “State-of-the-art on evolution and reactivity., Tech. Rep. REWERSE deliverable I5-D1, 2004, 2004.
2. S. Ceri, F. Daniel, and F. M. Facca, “Modeling web applications reacting to user behaviors,” *Comput. Netw.*, vol. 50, no. 10, pp. 1533–1546, 2006.
3. J. Bailey, G. Dong, and K. Ramamohanarao, “Decidability and undecidability results for the termination problem of active database rules,” in *PODS ’98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* New York, NY, USA: ACM Press, 1998, pp. 264–273.

4. J. Bowen and M. Hinchey, "Ten commandments of formal methods... ten years later," *Computer*, vol. 39, no. 1, pp. 40 – 48, 2006.
5. J. P. Bowen and M. Hinchey, "Ten commands of formal methods," in *High-integrity system specification and design*, J. P. Bowen and M. Hinchey, Eds. Springer-Verlag, 1998, pp. 215–230.
6. A. Hall, "Using formal methods to develop an atc information system," *IEEE Softw.*, vol. 13, no. 2, pp. 66–76, 1996.
7. L. Baresi, A. Orso, and M. Pezze, "Introducing formal specification methods in industrial practice," in *ICSE '97: Proceedings of the 19th international conference on Software engineering*. New York, NY, USA: ACM Press, 1997, pp. 56–66.
8. A. Ericsson and M. Berndtsson, "Rex, the rule and event explorer," in *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, 2007, pp. 71–74.
9. J. Bengtsson, K. G. Larsen, P. Pettersson, and Y. Wang, "Uppaal - a tool suite for automatic verification of real-time systems," *In Hybrid Systems III: Verification and Control*, pp. 232–243, 1996.
10. A. Ericsson, "Verifying an industrial system using rex," in *Technical Report HS-IKI-TR-08-001, School of Humanities and informatics, University of Skövde*, June 2008.
11. A. Ericsson, P. Pettersson, M. Berndtsson, and M. Seiriö, "Seamless formal verification of complex event processing applications," in *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, 2007, pp. 50–61.
12. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, M. Ardis, Ed. New York: ACM Press, 1998, pp. 7–15.
13. R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 26, pp. 183–235, 1994.
14. E. Baralis, S. Ceri, and S. Paraboschi, "Improved Rule Analysis by Means of Triggering and Activation Graphs," in *Rules in Database Systems (RIDS 95)*, T. Sellis, Ed., 1995, pp. 165–181.
15. J. Bailey, G. Dong, and K. Ramamohanarao, "On the decidability of the termination problem of active database systems," *Theoretical Computer Science*, vol. 311(1-3), pp. 389–437, 2004.
16. A. Aiken, J. Hellerstein, and J. Widom, "Static analysis techniques for predicting the behavior of active database rules," *ACM Transactions on Database Systems*, vol. 20, pp. 3–41, 1995.
17. E. Baralis, S. Ceri, and S. Paraboschi, "Compile-time and runtime analysis of active behaviors," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 10, no. 3, pp. 353–370, 1998.
18. T. Fors, "Visualization of rule behaviour in active databases," in *VDB*, 1995, pp. 215–231.
19. E. Benazet, H. Guehl, and M. Bouzeghoub, "Vital: A visual tool for analysis of rules behaviour in active databases," in *Proceedings of the Second International Workshop on Rules in Database Systems*. Springer-Verlag, 1995, pp. 182–196.
20. T. Coupaye, C. L. Roncancio, C. Bruley, and J. Larramona, "3d visualization of rule processing in active databases," in *NPIV '97: Proceedings of the 1997 workshop on New paradigms in information visualization and manipulation*, 1997, pp. 39–42.