Verifying an industrial system using REX.

AnnMarie Ericsson

Technical Report HS-IKI-TR-08-001
School of Humanities and Informatics
University of Skövde, Sweden

June 2, 2008

Abstract

The use of formal methods for enhancing software quality is still not used in its full potential in industry. We argue that seamless support in a high-level specification tool is a viable way to provide industrial system designers with complex and powerful formal verification techniques.

The REX tool supports specification of applications constructed as a set of rules and complex events. REX provides seamless support for specifying and verifying application specific requirement properties in the timed automata model-checking tool UPPAAL. The rules, events and requirements of an application design is automatically transformed to a timed automaton representation and verified in the UPPAAL tool.

In order to validate the applicability of our approach, we present experimental results from a case-study of an industrial system. Based on the case-study results, we conclude that complex applications can be efficiently verified using our approach.

Contents

1 Introduction **4**

2 Preliminaries **6**
2.1 The case study object 6
2.1.1 TUR 6
2.1.2 Details of TUR 8
2.1.3 Limitations 10

3 Experiment Description **11**
3.1 Purpose 11
3.2 Experiment Planning 11
3.3 Research question formulation 12
3.4 Threats to validity 12

4 Case study realization **13**
4.1 Preparation of case study 13
4.2 Developed rules 13
4.3 Verification 14
4.3.1 Modeled scenarios 15
4.3.2 Verification expressions 16
4.3.3 A Fictive Scenario 17

5 Case study Results **18**
5.1 Research questions and hypothesizes 18
5.2 Discussion 19
5.3 Project experience 19
5.3.1 Performance 19
5.3.2 Support for automatic verification 20
5.3.3 Support for simulating rule behavior 21
5.3.4 Support for modeling 21
5.3.5 Express Else statements 21
5.3.6 Compiler 21
5.4 Conclusion 22
5.5 Acknowledgement 22

6 Appendix **23**

6.1 Rules . . . 23

6.1.1 RM-rules (MAIN) . . . 23

6.1.2 Rules Main Delivery (RMD . . . 25

6.1.3 RMD-CreateDeliveryorder . . . 25

6.1.4 RD-CheckIntlevIn3040 . . . 26

6.1.5 Rules Main Production (RMP) . . . 29

6.1.6 Rules Production (RTP) . . . 32

6.1.7 Rules Derived Production (RTH) . . . 35

6.2 Verification . . . 36

6.2.1 Scenario . . . 36

6.2.2 Properties for verifying the RM rules . . . 38

6.2.3 Expressions for verifying the RMD rule set . . . 38

6.2.4 Expressions for verifying the RMP rule set . . . 40

6.2.5 Expressions for verifying the RTH rule set . . . 41

6.2.6 Expressions for verifying the RTP rule set . . . 41

6.3 Running verification expressions on scenarios . . . 42

Chapter 1

Introduction

Event triggered systems execute code in response to external stimuli. A paradigm well suited for implementing the reactive mechanism of event-triggered systems are event condition action (ECA) rules. An ECA rule executes an action A if condition C is satisfied when event E occurs.

Rule-based systems are powerful since they can react on events occurring in arbitrary order as well as reacting on combinations of event occurrences. Using rules in critical systems implies that the system behavior must be thoroughly analyzed. However, analyzing a set of low-level ECA rules is a complex task due to interactions between rules [1].

Formal methods provides a mathematically based method for specifying and verifying applications. Some formal methods, such as, methods based timed automata [2], have CASE tools supporting model-checking of requirement properties. Nevertheless, formal methods are not used in its full potential in industry. Some of the reasons may be the high threshold one needs to pass to be able to use them and the extra time it may take to construct the specifications [3].

We address the problem of analyzing rule-based applications by utilizing the power of model-checking for verifying system behavior. A graphical tool (REX) is constructed [4], serving as a rule-based front end to the timed-automata CASE-tool Uppaal [5]. Rules are specified in the high-level rule based language supported by REX. The high-level 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.

The aim of our work is to lower the knowledge threshold developers need to pass in order to utilize the power of model-checking for verifying rule-based applications. Instead of developing a new formal theory for analyzing rule-based systems, the well founded theory of timed automata is utilized. A graphical tool (REX) is constructed [6], serving as a rule-based front end to the timed-automata CASE-tool Uppaal [5].

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.

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 REX is a feasible tool for verifying and analyzing a rule based system of industrial complexity. Additionally, REX may be very useful in the maintenance phase of such systems.

The paper is structured as follows; Chapter 2 contains background information about the modeled system TUR. Chapter 3 describes the experiments and the hypothesizes for this project. Chapter 4 presents the realization of the case study and chapter 5 presents results and conclusion.

Chapter 2

Preliminaries

The feasibility of using REX for modeling and verifying a rule based system is validated in a case study. A system for producing assembly plans for engine plants at Volvo IT in Skövde was chosen as case-study object. The system has a complex behavior dependent on both values of parameters in incoming telegrams and stored values in database tables.

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 the case-study shows that REX is a feasible tool for modeling and verifying a system of industrial complexity. The chosen case-study object does not have explicit time constraints. However, the behavior is complex with respect to the logic controlling the dataflow and it is critical since a failure in this system affects the production plants.

2.1 The case study object

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. The system TUR is implemented and maintained by this group.

2.1.1 TUR

The main task for 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.

In this work, a specific engine plant with a specific structure is used to exemplify the use of TUR. However, the system is designed to be flexible with

respect to the structure of the production plant. The requirement for the system to run in different environments is why the structure of the plant and capacity of different producers are stored in database tables.

In the exemplified engine plant, TUR mainly communicates with four other systems as shown in Figure 2.1. MOPS sends high-level assembly plan to TUR (relation [1] in Figure 2.1) and TUR may request more plan from MOPS. ASSL, ASFL and MOTOR can request more assembly plan from TUR that returns a processed plan or an error message (relation [2] in Figure 2.1).

MOPS is a system used for planning the production in the factory. MOPS interacts with several other systems that monitors, for example, orders from customers and finished products. Based on customer orders, MOPS delivers high-level assembly plan and data controlling the development of assembly order to TUR. Before a new plan is sent to TUR it is simulated in MOPS in a system equal to TUR. The aim of the simulation is to check that the data that the new plan uses in the database is correct, no required data is missing and that it is possible to produce the new plan with the given control data. TUR can request more high-level assembly-plan from MOPS if required.

ASFL is an administration system for storage of sub-items produced in another part of the factory. The storage must always be ready to deliver sub-parts to the factory plants. TUR produces a delivery plan for ASFL specifying what sub-parts that must be delivered at what time. If ASFL is out of delivery order it sends a request to TUR for more delivery plan.

ASSL is an administration system for items from suppliers, e.g. flywheels. TUR tells ASSL what type of items that must be delivered from suppliers and when. If ASSL is out of delivery order it sends a request to TUR for more delivery plan.

MOTOR is an administration system for the production plants. It requests detailed assembly plans from TUR for its producers and TUR returns new assembly plan, or a failure message.

For users of ASFL, it is most convenient to produce large sequences of one type of items with few changes of type. However, producers monitored by system MOTOR requires several different types to be delivered from ASFL. TUR must consider the trade off between the desire to produce large batches of the same item type in ASFL and the requirement to always be ready to deliver a certain amount of different item types to MOTOR.

The producers using system MOTOR is divided into two logical units, inner (IM) and outer (YM), where YM concerns different types of engines and IM concerns different sub-items for engines. The inner unit, IM, provides detailed items to YM. In this case study, a module, for example, an assembly line receiving items from another model is called a primary producer and a module producing items to another module is called sub-producer. The same module can be both a primary and a sub-producer.

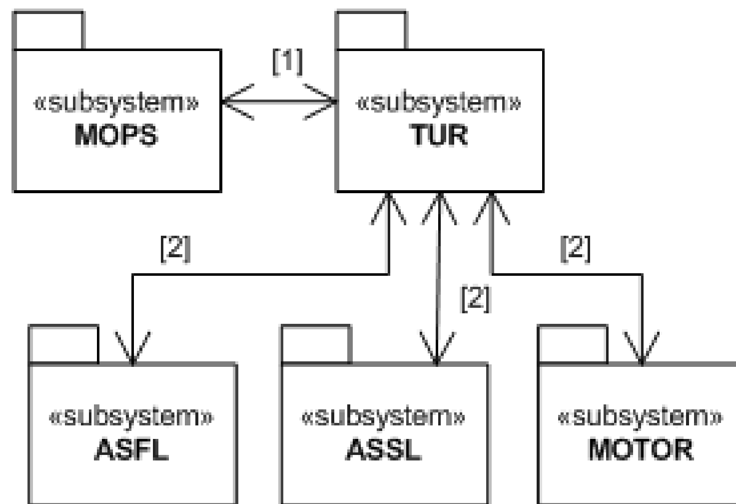


Figure 2.1: Systems communicating with TUR.

To avoid lack of items, the assembly plans provided by TUR are coordinated for IM and YM. The order of items manufactured by different plants must be correlated with each other since assembly lines combining items from different producers must receive all items contributing to their item in correct time and order. An example of relations between different modules monitored by system MOTOR are shown in Figure 2.2.

When an assembly line in IM has not enough planned items in its assembly order, system MOTOR calls TUR for more assembly plan. TUR checks if there are more assembly plan in the primary producer. If there are more items planned in the primary producer, TUR creates an order list of different types of items to be manufactured by the calling producer. If, recursively, no more items are planned for the producers primary to the caller, TUR calls the system for high-level assembly plans, (MOPS), asking for more high-level plans to convert to low-level assembly plans.

2.1.2 Details of TUR

TUR is currently implemented in a language based on FORTRAN. The main program poll a telegram queue for new messages, performs the task requested by the telegram (creates new assembly order) before it reads next message in the queue. If an error is detected, for example, that the telegram has the wrong type or that the producer does not exist in the expected table, TUR executes recovery code, for example, performs roll back of the database or, if the error is not severe, sends an error message and tries to continue. If TUR stops producing assembly order, the engine plants will stop within one or a few hours.

TUR consists of a set of batch processes with different tasks. This case

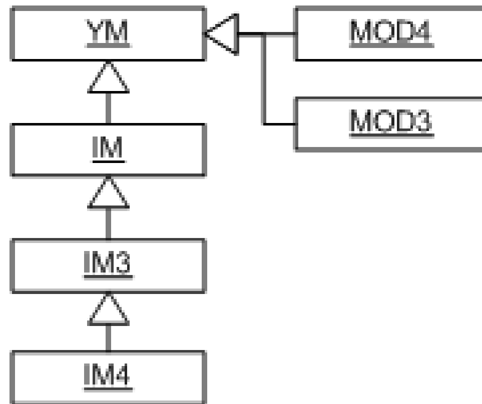


Figure 2.2: Example of relation between producers and primary producers monitored by system MOTOR.

study focus on the process creating the detailed assembly plans. This process is referred as the *main program* within this case study. The main program controls what type of plans that should be constructed and calls the correct subprogram to create the plan.

Depending on type of current producer, different types of assembly plan are generated by converting assembly plan in primary producer to assembly plan for sub-producers. The following different types of assembly plans are produced.

- **Assembly order according to group** for logical producer YM. The high-level assembly plan received from MOPS are transformed to a sequence according to planning-groups. Each planning-group contains producers with similar capacity. All items within each group is separated according to priority before assembly order is produced for sub-producers. Coordination between different planning groups are directed by simulations of how sub-producers to YM will request for future assembly order (which sub-producer will be finished first).

- **Derived assembly order according to group** for logical producer IM. Order is created by reading order lines for YM and create new order lines with respect to which sub-items are included in the planned items.

- **Production order for assembly modules (MOD3,MOD4).** The modules are controlled by the plan constructed for YM. Modules fetch next line of orders for current planning group from assembly plan in YM.

- **Production order for IM3** Similarly as for YM, future requests for assembly order are simulated for IM with respect to capacity of IM3. Order is delivered after the policy that the producer expected to finish the order first gets it.

- **Derived production order** for IM4. The assembly order for IM4 are created by reading assembly order from IM3 and derive all subitems needed to produce those orders.
- **Deliveryorder** are order produced for internal suppliers. The order reflects what (camshaft, crankshaft, etc.) should be delivered to which producer.

2.1.3 Limitations

TUR is an extensive system and due to time limitations of this project, the entire functionality can not be modeled in detail in this case study. There are two options; *i)* Model the high-level behavior of TUR, i.e. control flow for selecting type of assembly-order to construct. *ii)* Model a part of the functionality in detail, i.e. select one type of assembly-order and model the behavior in detail.

The first option i is selected in this project since rules are better suited for controlling high-level behavior than for implementing sequential details. However, details are modeled to some extent since modeling challenges such as describing SQL expressions and database tables in REX and Uppaal are interesting for the aim of this project.

To overcome differences between expressiveness in Uppaal and the written source code of TUR, the following abstractions are utilized:

- Variables of types String are modeled as integer and constant types.
- Database tables are modeled as two dimensional arrays of integers or constants.
- SQL questions over databases in TUR are written as functions over two dimensional arrays in Uppaal.

Chapter 3

Experiment Description

This section describes and motivates the design of the case study and discuss threats to validity.

3.1 Purpose

The purpose of this case study is to validate the approach of verifying the behavior of a rule based system using REX and UPPAAL. The case study aims to show that a system of industrial complexity can be modeled as rules in REX. Additionally, the approach for describing requirement properties using the patterns provided by REX are validated by verifying the constructed model.

3.2 Experiment Planning

Since there are no such thing as a typical rule based system, a system not implemented as ECA rules but with a rule based behavior was chosen as case study object. The benefit of using TUR as case study object is that it is a real-life system with a semantic that contains both rules and complex event occurrences. Additionally, the scope of the case study is dynamic since it is possible to model the system in several steps. First, a small set of rules with core functionality calling "action stubs" are modeled. The "action stubs" returns a default value. Iteratively, the "action stubs" are extended to trigger rules modeling more detailed functionality.

Although TUR is not implemented as ECA rules, the behavior of the system can be modeled as reactive rules. The strategy used for modeling the behavior as rules are presented in section 4.2.

3.3 Research question formulation

Based on the purpose of this case study, stated in section 3.1, we state the research question Q_1 and Q_2.

- Q_1: Is it possible to model the high-level behavior of the case study object TUR described in section 2.1.1 as a set of rules using REX and UPPAAL
- Q_2: Is it possible to formulate requirement questions based on the approach used in REX to verify a model of TUR.
- Q_3: Is it possible verify the behavior of an industrial system in REX.
- Q_4: Is it possible improve the maintenance process of an existing system using REX.

3.4 Threats to validity

A threat to validity is that the resulting set of rules does not correctly model the behavior of TUR due to wrong interpretation of requirements. Access is given to the source code of TUR, but no ability to perform tests on the running system. Hence, it is not possible to perform tests on the running system and no typical test cases exists.

To validate that the requirements are correctly interpreted, employees at Volvo IT has manually verified the correctness of the expected results of the test-cases. Additionally, the behavior of the rules has been compared to the expected behavior of the source code by the developer.

Chapter 4

Case study realization

4.1 Preparation of case study

In the preparation phase, an introduction to the system TUR was given by employees at Volvo IT. Additionally, knowledge of the behavior of the system TUR was retrieved by reading documentation and source code and asking questions to the project supervisor at Volvo IT. Moreover, the system was partly modeled in UPPAAL as it is, i.e. modeling each procedure as a timed automata and calls to procedures as synchronization over channels.

Although the system was only partly modeled in UPPAAL, the experience of modeling database tables and SQL-expressions in UPPAAL was valuable for the forthcoming work. The experience gained was utilized to extend the functionality of REX with ability to model database tables and SQL-expressions in REX and automatically transform them to UPPAAL.

In order to facilitate the modeling phase, REX was extended with ability to graphically simulate rules and events and ability to attach comments to all items in the model. Additionally, the ability to classify rules into different groups in order to focus on particular behavior in the simulator was introduced.

4.2 Developed rules

TUR is currently implemented in a language based on FORTRAN. It is a sequential program where choices are implemented as *if* or *case* statements. The structure of the implementation can be described as a tree structure where the root of the tree is the main program, the sub-programs or procedures called from main are the nodes in the first level of the tree and the sub program called from the nodes in the first level is the second level nodes. The leafs in the tree are typically executing SQL expressions.

Figure 6.1 in Appendix shows a tree constructed from the source code of TUR where each node in the tree represent a procedure. The source code represented as gray nodes are modeled as rules and events in this project. The

code represented by the gray nodes mainly contains choices and calls to sub-programs executing simple SQL expressions that can be modeled in REX. The white nodes represent procedures that are not modeled as rules in this project. The reason why the procedures are not modeled as rules is that they mainly contain code that is not suitable to express in a high-level model, for example, complex SQL expressions with joins and cursors or large sequences of sequential code that is not suitable to model as rules (if not represented as one large action part). Additionally, the entire behavior of TUR can not be modeled within the time frames of this project. The tree does not contain all procedure calls made by procedures marked as white. In the REX model, the white nodes are modeled as "stubs", returning a default value.

The implementation of the procedures represented by the gray nodes was transformed to a set of rules by using the following mapping:

- Each *If* statement is transformed to two rules, R_i and R_j triggered by the same event. The expression EXP in the if statement is used to form the condition expressions in R_i and R_j where the condition in R_i is EXP and the condition in R_j is $\neg EXP$.

- Each case statement with n different entries is transformed to n rules triggered by the same event. The condition and action parts of rule R_i, where $1 <= i <= n$, maps to the corresponding case statement and its following action.

- Each procedure call, where the procedure does not itself contain calls to other procedures are transformed to a rule. The action part of the rule performs the procedure (in many cases, executes an SQL expression). A new event is generated when the action is executed. Possible return values are mapped to parameters in the generated event.

- Each call to a sub-program, or a procedure that itself contains procedure calls, are transformed to a set of rules according to the previous mapping rules.

- Single sequences of code, such as resetting variables, that are not calls to procedures or choices are either included in related rules or transformed to a rule on its own. Large sequences of code including, for example, complex SQL expressions are not transformed to the rule base.

- Calls to other systems, for example, connections to databases and calls for setting up and testing communication are not modeled.

A complete description of the resulting rule set is presented in section 6.1 in Appendix.

4.3 Verification

A complete verification search by the model-checker requires that all verification expressions are checked against all permutations of possible values in database

tables and telegrams since the behavior of the system depend on these values. Such search is not possible to perform since the number of permutations is too high. In order to reduce the search space, test-scenarios are constructed. The test-scenarios combined with the verification expressions should cover the following criteria:

- Each rule should be checked at least once, implicitly or explicitly.
- For rules with conditions, the condition must be checked,implicitly or explicitly, both for true and false evaluation of the expression.

A rule R is said to be explicitly verified if property P_i checks that R always executes in the given scenario S_m and property P_j checks that R never executes in the given scenario S_n. A rule R_i is said to be implicitly verified if rule R_i is executed to verify a property P, and P can not be satisfied if R_i is not executed.

4.3.1 Modeled scenarios

There are two different main types of telegrams in TUR; *request for production assembly order* denoted T_p in this project and *request for delivery assembly order* denoted T_d in this project. Telegrams of type T_p are sent from MOTOR and telegrams of type T_d are sent by ASFL or ASSL. If the telegram type is unknown (not delivery order or production order), recovery code is executed before next telegram is read from the buffer. In the default case, only one message exists in the buffer.

The database characteristics are based on a default case where all database tables contains consistent and correct values, i.e. reading the database should not cause any error message. When the database is altered to control the behavior, for example, to force an error message to be raised, this is seen as a database characteristic in the test scenario.

Based on the two main types of messages and the database characteristics, test scenarios are constructed. The following four scenarios exemplifies the test scenarios, a complete list of test scenarios are presented in section 6.2.1 in Appendix.

- **Scenario** S_{001}
  Message type: T_d
  Database characteristics: default
- **Scenario** 2_{002}
  Message type: T_p
  Database characteristics: default (Producer type = PROD, Order according to PLAN)
- **Scenario** S_{003}
  Message type: T_p
  Database characteristics: default (Producer type = HPROD)

- **Scenario** S_{004}
  Message type: T_p
  Database characteristics: default (Producer type = PROD, Order according to PRIORITY)

4.3.2 Verification expressions

A list of the properties created to fulfill the test criteria are presented in section 6.2.2 in Appendix. The listed properties uses the Universality and Existence patterns in scope Globally. The Universality pattern in Globally scope is transformed to the "leads to" property in UPPAAL to check wether executing the model will always lead to the specified state. The Existence pattern in scope Globally checks if there exists a path to the specified state.

The Universality pattern is suitable when checking that the system will *always* reach a specific state. In this case study, a correct final state is reached when rule RM-Ready is executed with no error messages in the error log (FELMED == EMPTY) and correct number of items ordered (UTFANT == PLANT).

Property P_{004}, for example, checks if a correct final state is always reached with pattern Universality and in Globally scope. If there is an interaction or race condition in the system that causes that the system is not reaching a correct final state, property P_{004} will return False (not satisfied).

The Existence pattern in scope Globally is suitable when checking if an erroneous state can be reached. In this case study, rule RTP-ActualDifferenceReached executes when the sum of the number of planned items and current status are too high compared to a maximal value. If there is some interaction or race condition in the model causing RTP-ActualDifferenceReached to execute, P_{405} will return True (satisfied).

After verifying that each rule executes correctly, correctness between relations of rules were verified by using more complex verification patterns. In addition to the patterns listed in section 6.2.2, the following properties exemplifies how requirements were identified and transformed to properties and verified in scenario 1-4:

Requirement1 Rules producing derived production order shall not be executed when production order is created and the producer type is PROD.
Property verifying requirement 1 (Expected result = true):
Pattern: P is Absent Between R and S
Predicate P: Rule TurHProdStart.EXECUTE
Predicate R: Rule RequestForProductionOrder.EXECUTE
Predicate S: Rule RM-Ready.EXECUTE

Requirement2 Each reading of a telegram results in a correct final state (Expected result = true):
Pattern: P respond to Q
Predicate P: Rule RM-Ready.EXECUTE
Predicate R: E-ReadNewTelegram.OCCUR

Requirement3 The rule RMD-CheckGenerateDeliveryorder must always be executed before rule RMD-GenerateDeliveryOrder (Expected result = true):
Pattern: P is Absent before Q
Predicate P: RMD-GenerateDeliveryOrder.OCCUR
Predicate R: RMD-CheckGenerateDeliveryorder.OCCUR

Table 6.1 shows an overview of expected results for the verification properties listed in section 6.2.2 in each scenario. The verification of the model were ended when the result from running the model-checked corresponded to the expected results.

4.3.3 A Fictive Scenario

The system TUR was only partially modeled in this projet. The high-level behavior of TUR has a rule based semantic, i.e. the behavior can be described as ECA rules. The low-level behavior executes sequential code or complex SQL expressions that does not have a rule based semantics. However, if REX were extended with more powerful support for generating SQL expressions, then these expressions could be generated to the action parts of rules.

Assuming that a complete model of TUR is specified in REX and that the model uses a complete representation of the database, users of TUR were asked if REX could contribute to the management of TUR. The primary use identified by the employees was to find the future time point when the production run out of sub-items.

Requirement TUR runs in simulated mode to check feasibility of planning 18-19000 engines ahead. The aim of running the simulation is to, for each item, find the time when we run out of sub-items. A property to check is "at what time is the number of items of type X equal to 0". REX does not give specific return values, however, pattern P can be used to check if "nr of X == 0" can be reached and the time can be found in the trace returned by UPPAAL.
Pattern: P Exists
Predicate P: nr of X == 0

In addition, the employees identified that it would be useful to check the effect of changes, for example, change of time it takes to perform different task, include breaks for workers, check the effects of missing or inconsistent data in specific tables, etc. Given a complete model of the system and a complete set of verification properties, such changes are can be performed in the model without affecting the actual system. After changing the model, the properties are run and users can check whether the change resulted in a difference in behavior.

Chapter 5

Case study Results

The aim of this project is to validate the use of REX when modeling a rule based system of industrial complexity.

5.1 Research questions and hypothesizes

The result of the case study is presented with respect to the research questions.

Question Q_1

Recall research question Q_1, concerning whether it is possible to model the high-level behavior of the case study object as a set of rules in REX.

Section 4 describes how rules were developed from the semantics of the existing source code. The developed model concerns the high-level behavior of TUR focusing on how different ordering algorithms are executed based on contents of incoming telegrams and values in database tables. Hence, the answer to question Q_1 is yes.

Question Q_2

Recall research question Q_2, concerning whether REX can be used to formulate requirement questions to verify a model of TUR. Section 4.3 presents how scenarios and properties are combined to fulfill the test criteria that each rule should be verified at least once. In addition to fulfilling the test criteria, examples of how correctness of relations between rules can be verified are presented in section 4.3.2. Hence, the answer to question Q_2 is yes.

Question Q_3

Recall research question Q_3, concerning whether it is possible verify the behavior of an industrial system in REX. Given the results of research questions Q_1

and Q_2 together with the fact that all the specified verification properties were possible to verify in REX, the answer to question Q_3 is yes.

Question Q_4

Recall research question Q_4, concerning whether it is possible to improve the maintenance process of an existing system using REX. Given the discussion in section 4.3.3, a formal model of a system with a user friendly interface and ability to check different types of properties are appealing to developers.

5.2 Discussion

The verification properties used in this project checks the correctness of each rule in the specified scenarios. However, the fact that two different rules always executes in a scenario does not necessarily mean that the behavior of the system is correct. First, the rules may behave differently in a different scenario. Secondly, it may be the case that rule R_1 is required to execute before rule R_2 for a correct result. Such behaviors are not captured by the verification expressions stated in 6.2.2. However, some examples of how such expressions can be stated are presented in 4.3.2. The test criteria that each rule should be tested at least once for each outcome of the condition evaluation can be generalized as a general criteria for all different rule sets while criteria for testing relations between rules are application dependent.

5.3 Project experience

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

5.3.1 Performance

Previous experiments has shown that the performance of REX and Uppaal is heavily dependent on the level of non-determinism in the model (ref to previous experiment). A summary of the number and types of rules and events included the rule model of TUR is shown in Table 5.1.

The model of TUR consists of 63 rules, 12 complex events, 50 primitive events, 30 data objects and 45 event parameters separated on different events. Some of the complex events are composed by other complex event in an event hierarchy. One of the complex events (CE_StartTurProd) with depth three, and four of the complex evens have depth two.

Item type	Amount
Rules	63
Primitive Events	50
Complex events conjunction	8
Complex events disjunction	4
Data objects	30
Database tables	12

Table 5.1: Summary of TUR model.

Additionally, the model consist of 12 modeled database tables with 5 rows filled with values. 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.

Each property verified in scenarios are verified in less than 1 second implying that even if the model is complex for a human user, the behavior has a low level of non-determinism in each scenario. The main reason why the verification performs so well in this model is that the scenario is deterministic. Each scenario tests the input of one type of message, there are no tests on different types of events occurring in non-deterministic order, which would cause a state space explosion.

5.3.2 Support for automatic verification

Without tool support, it is hard to track the behavior even for a small set of rules. When complex events and conditions reading event parameters are supported, it gets even worse. Hence, increased expressiveness of the language results in increased complexity of the behavior of the set of rules.

In the case of REX, support for verification is the very aim of the 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 continuously retrieves feedback about whether the set of rules behave as intended.

The advantage of using a model-checker as a complement to testing is that it is possible to test the behavior of the system before it is executable. Additionally, it is possible to formulate an undesired state and ask the model-checker if it is possible to reach that state. If the state is reachable, a trace is given showing how to get there. The designer can manually analyze the trace and take actions to ensure that this state is not reachable.

5.3.3 Support for simulating rule behavior

A graphical simulator can visually show execution traces. If the simulation can be performed step-by-step, it provides the developer with the ability to manually choose in case of non-deterministic choices and inspect the behavior in specific phases of execution.

Visualizing rule execution means showing a large amount of data. A good simulator has ability to focus the simulation on a part of the rule set by only visualizing a part of the data (excluding rules, events or variables). The developer may, for example, be interested in the behavior of a small set of rules in a particular part of the execution. It is desirable to be able to select a state where the simulation should start from and which rules or events that shall be visible in that particular simulation.

The simulator in REX uses Uppaal to retrieve step-by-step information. 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.

The ability to start the simulation in a selected state is currently not supported by REX. However, it can be implemented by supporting traces from the verifier to be visualized in the simulator. The traces can then be used as starting states in the simulator. This ability are present in Uppaals original front-end, but not yet implemented in REX.

5.3.4 Support for modeling

In the current version of REX, relations between rules are shown in a tree structure. However, the tree structure is insufficient when rules are triggered by complex events. A more powerful visualization ability is desirable. For example, a modeling section based on UML-A [7] where connection of rules and events are shown in a model based on UML or a triggering graph extended with ability to model complex events. Currently, when using REX with rules and complex events, the developer need to draw the triggering relations manually or in an additional tool to get an overview of the structure of the rule set.

5.3.5 Express Else statements

In some cases it is desirable to express that a rule should execute action A_1 if the condition C is true and action A_2 if condition C is false. However, REX does not support this ability. The developer must specify two different rules R_1 and R_2 triggered by the same event. R_1 executes A_1 if C is true and R_2 executes A_2 if $\not C$ is true.

5.3.6 Compiler

Currently, REX maintain relations between all items in the rule set if the actions consist of assignments.

However, by using the ability to write simple functions in Uppaal, REX can be used for modeling, for example, simple SQL expressions or any code possible to write in Uppaal. The function written in REX is simply copied to the function ability in Uppaal. This means that no relations are maintained between the data-objects used in functions and other items. A good parser could be useful for maintaining the relations and a compiler could stop, for example, syntax errors in the code written in REX from being revealed first when the model is transformed to Uppaal.

5.4 Conclusion

This paper reports the results of a case-study where the tool REX is used to model and verify the high-level behavior of the system TUR. REX act as a rule based front end to the timed automata case tool Uppaal. Models specified in REX are automatically transformed to a timed automaton representation of the set of rules to enable model-checking to be performed on the rule set.

TUR is a system used for constructing assembly order for construction plants. The high-level behavior of TUR are transformed to ECA rules and specified in REX. The behavior of the model specified in REX is verified to have equal behavior as the corresponding source code in TUR.

The results from the case study shows that REX is a feasible tool for modeling and verifying the high-level behavior of a rule based system of industrial complexity.

5.5 Acknowledgement

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net) and CUGS (the national graduate school of computer science). The author would like to thank the staff at Volvo IT Skövde for help and support during this project.

Chapter 6

Appendix

6.1 Rules

6.1.1 RM-rules (MAIN)

RM-Read-Telegram

The rule is triggered when a new event should be read from the telegram buffer.
Condition evaluated to true covered by test-case scenario S_2 and verification expression P_{001}.
Condition evaluated to false covered by test-case scenario S_2 and verification expression P_{004}.

ON E-ReadNewTelegram?
IF DontStopReadingTelegrams
DO StoreTelegramValues();
E-TelegramSaved!

RM-RequestForDeliveryOrder

The rule triggered when a telegram is received and the event parameters from the telegram is stored in local variables. The condition is evaluated to true if the telegram type is delivery order.

Condition evaluated to true covered by test-case scenario S_{001} and verification expression P_{001}.
Condition evaluated to false covered by test-case scenario S_{002} and verification expression P_{001}.

ON E-TelegramSaved
IF TELTYPE==PRODTURORDNING AND AppNotStopped
DO E-RequestForProductionOrder!

RM-RequestForProductionOrder

The rule triggered when a telegram is received and the event parameters from the telegram is stored in local variables. The condition is evaluated to true if the telegram type is production order.

Condition evaluated to true covered by test-case scenario S_{002} and verification expression P_{001}. Condition evaluated to false covered by test-case scenario S_{001} and verification expression P_{001}.

ON E-TelegramSaved
IF TELTYPE==PRODTURORDNING AND AppNotStopped
DO E-RequestForProductionOrder!

ER-UnknownTelegramType

The rule is triggered if the telegram type is neither T_p or T_d. Condition evaluated to true covered by test-case scenario S_{000} and verification expression P_{003}. Condition evaluated to false covered by test-case scenario S_{001} and verification expression P_{001}.

ON E-TelegramSaved
IF (TelegramType != LEVTURORDNING) AND
(TelegramType != PRODTURORDNING)
DO APPL = HOLD
ErrorMessage = Unknown telegram type

RM-Tursttid

This is a stub for getting first opening time for producer.
Covered by test-case scenario S_{001} and verification expression P_{001}.

ON E-Tursttid
DO E-Tursttid-Done!

RM-Ready

The rule is triggered when the execution is stopped and there are no more messages in the message buffer.

Condition evaluated to true covered by test-case scenario S_2 and verification expression P_{004}.
Condition evaluated to false covered by test-case scenario S_2 and verification expression P_{001}.

ON E-ReadNewTelegram?
IF StopReadingTelegrams()
DO E-Ready!

6.1.2 Rules Main Delivery (RMD

RMD-Get-S3041-Dist-Intlev

Gets system id for the internal deliverer from table S3041. Covered by test-case scenario S_{001} and verification expression P_{101}.

ON E-RequestForDeliveryOrder
DO A-Get-S3041-DIST-INTLEV()
E-GotSystemIdIn3041!

RMD-CheckS3043Stopped

Checks if the producer exists in table S3043. In that case, the producer is stopped and an error message should be sent. The result is sent as a parameter with the triggered event.

Covered by test-case scenario S_{001} and verification expression P_{101}.

ON E-RequestForDeliveryOrder
DO A-CheckS3043Stoppad()
E-StoppadCheckedIn3043!

ER-Leverantor-Stoppad

Triggered by rule RMD-checkS3043Stopped. The condition is true if the producer exists in S3043.

Condition evaluated to true is covered by expression P_{103} and scenario S_{101}, condition evaluated to false is covered by S_{001} and verification expression P_{103}.
ON E-StoppadCheckedIn3043?
IF LeverantorStoppadIn3042()
DO Error-turordningsko-for-leverantor-stoppad()
E-ReadNewTelegram!

6.1.3 RMD-CreateDeliveryorder

After all checks that the database is ok, the system starts a loop calling subprograms for executing delivery order. Covered by test-scenario S_{001} and expression P_{101}.
ON CE-StartCreatingDeliveryOrder
DO A-CheckProdAndTypIn3020()
E-Check-ProdAndTypIn3020!

E-Start-Creating-Deliveryorder!

6.1.4 RD-CheckIntlevIn3040

Checks that data concerning internal deliverer exists in table S3040. Covered by test-scenario S_{001} and expression P_{107} (true) and test-scenario S_{105} and expression P_{104} (false).

ON E-ProducerTypeCheckedIn3020?
IF C-ProducerTypeOKIn3020
TELTYPE==LEVTURORDNING **DO** A-Get-S3040-Intlev()
E-IntlevCheckedIn3040!

ER-NoIntlevIn3041

Executes if the intlev (internal deliverer) is not in table S3040. An error message is sent before the loop continues.

Covered by test-case scenario S_{102} and verification expression P_{105} (true) and test-case scenario S_{001} and verification expression P_{105} (false).

ON E-GotSystemIdIn3041
IF C-NoIntlevIn3041()
DO E-ReadNewTelegram!

ER-NoIntlevIn3040

Executes if the intlev (internal deliverer) is not in table S3040. An error message is sent before the loop continues.

Covered by test-case scenario S_{103} and verification expression P_{107} (true) and test-case scenario S_{001} and verification expression P_{004} (false).

ON E-IntlevCheckedIn3040
IF IntlevCheckedIn3040False()
DO ErrorIntlevNotIn3040()
E-CheckDeliveryLoop!

RD-GetDeliveryCondition

Gets delivery condition from table 3042

ON E-IntlevCheckedIn3040
IF IntlevCheckedIn3040True()
DO A-GetDeliveryCondition()()
E-GotArtantIn3042!

Expected result is T for property P_{106} and scenario S_{001}
Expected result is F for property P_{106} and scenario S_{103}.

ER-NoProducerTypeIn3020LEV

ON E-ProducerTypeCheckedIn3020
IF C-NoProducerTypIn3020()
TELTYP==LEVTURORDNING **DO** A-Error-NoProducerIn3020()()
E-CheckDeliveryLoop!

Expected result is T for property P_{104} and scenario S_{105}
Expected result is F for property P_{104} and scenario S_{001}.

RD-UtfBearb

Stub for creating delivery order
ON CE-RD-UTFBearb
IF UTFANT¡PLANT)
DO UTFANT++
E-CheckDeliveryLoop!

Expected result is T for property P_{106} and scenario S_{001}
Expected result is T for property P_{106} and scenario S_{103}

RD-CheckProducerAndTypeIn3020

Gets delivery condition from table 3042 **ON** E-Check-ProdAndTypIn3020
DO A-CheckProdAndTypeIn3020()()
E-ProducerTypCheckedIn3020!

Expected result is T for property P106 and scenario S001
Expected result is F for property P106 and scenario S105.

ER-NoDeliveryConditionIn3042

Executes if delivery conditions is not in table S3042. An error message is sent before the loop continues.

ON E-GotArtantIn3042
IF C-NoDeliveryConditionIn3042() and TELTYP ==LEVTURORDNING()
DO ErrorLeveransvillkorEjUpplagt()
E-CheckDeliveryLoop!

Expected result is T for property P_{108} and scenario S_{104}
Expected result is F for property P_{108} and scenario S_{101}.

RMD-CheckLoopCondition

Checks the that the loop condition is true
ON CE-DeliveryLoop?
IF $UTFANT >= PLANT$
DO E-ReadNewTelegram!

Expected result is T and F for property P_{004} and scenario S_{001}

RMD-generateDeliveryOrder

Stub for generating deliveryorder
ON CE-DeliveryLoop?
IF $DeliveryLoop > 0$ and $UTFANT < PLANT$
DO E-DeliveryOrderGenerated!

Expected result is T and F for property P_{004} and scenario S_{001}

RMD-CheckGenerateDeliveryOrder

Stub for generating deliveryorder in the first iteration.
ON CE-DeliveryLoop?
IF $DeliveryLoop == 0$ and $UTFANT < PLANT$
DO DeliveryLoop++
E-DeliveryOrderGenerated!

Expected result is T and F for property P_{004} and scenario S_{001}

RMD-CreateDeliveryProductionOrder

Stub for generating deliveryorder.
ON E-DeliveryOrderGenerated?
DO E-deliveryProductionOrderCreated!

Expected result is T for property P_{004} and scenario S_{001}

6.1.5 Rules Main Production (RMP)

RMP-CheckProducerIn3020

Checks if the current producer exists in table S3020 The event signalled when the action is done has a parameter with result
Covered by test-scenario S_{002} and expression P_{204} (true) .

ON E-RequestForProductionOrder
IF AppNoStopped, OK==YES
DO A-CheckProducerAndTypIn3020
E-ProducerTypeCheckedIn3020!

RMP-CheckPProducerIn3020

Checks if the current primary producer exists in table S3020 The event signalled when the action is done has a parameter with result
Covered by test-scenario S_{002} and expression P_{204} (true) and S_{202} and expression P_{204} (false). **ON** E-PProducerCheckedIn3024
IF PProducerExistsIn3024, OK==YES
DO A-CheckPProducerIn3020
E-PProducerCheckedIn3020!

RMP-GetPProducerIn3024

Checks if the current producer has a primary producer in 3024. Covered by test-scenario S_{002} and expression P_{204} (true).
ON E-RequestForProductionOrder
IF AppNotStopped, OK==YES
DO A-GetPProducerIn3024
E-PducerCheckedIn3024!

RMP-CheckClosedProducerIn3025

Checks if the current producer exists in table S3025 If it does, it implies that this producer is stopped and execution can not proceed. The event signalled when the action is done has a parameter with result.
Covered by test-scenario S_{002} and expression P_{204} (true).

ON E-RequestForProductionOrder
IF AppNotStopped, OK==YES
DO A-CheckClosedProducerIn30205
E-ClosedProducerCheckedIn3025!

RMP-TurProdStart

Starts the chain of rules for TURPROD.
Covered by test-scenario S_{002} and expression P_{201} (true) and test-scenario S_{003} and expression P_{201} (false).
ON CE-StartTurprod
  IF C-TYPE==PROD
  DO E-Get-S3026-Batchstorlek!
    E-Sum-3018-Extra-sekvensrad!
    E-sum-3052-plangrupp!

RMP-TurHProdStart

Starts the chain of rules for TURHPROD.

Covered by test-scenario S_{002} and expression P_{202} (true) and test-scenario S_{003} and expression P_{202} (false).
ON CE-StartTurProd
  IF TYPE==HPROD
  DO E-StartTurHProd

RMP-GenerateProductionOrder

When TurProd or TurHProd rules are finished, they are triggering this rule.

ON E-TurDone
IF UTFANT¡PLANT
DO E-ProductionOrderGenerated

RMP-CheckLoopCondition

When TurProd or TurHProd rules are finished, they are triggering this rule.

Covered by test-scenario S_{002} and expression P_{004} (true).
ON E-TurDone
  IF UTFANT¿=PLANT
  DO E-ReadNewTelegram!

ER-ProducerStopped

If the current producer exists in table S3025, the order queue is stopped for additional processing. A message is sent that the producer exist in 3025, for $TYPE = PROD$, execution is stopped, however, for $Type == HPROD$, the execution is not stopped. Covered by test-scenario S_{203} and expression P_{203} (true) and test-scenario S_{002} and expression P_{203} (false).

ON E-ClosedProducerCheckedIn30205?
IF ProducerIn3025() AND $TYPE == PROD$
DO OkisNo
E-TurDone!

ER-PProducerCheckedIn3020Error

No primary producer is registered for the producer in 3020 A message is sent that the primary producer is missing in 3020. Covered by test-scenario S_{205} and expression P_{208} (true) and test-scenario S_{002} and expression $P_{208}(false)$

ON E-PProducerCheckedIn3020
IF PProducerNotIn3020
DO A-ErrorNoPrimaryProducer,OkisNo
E-TurDone!

ER-NoPProducerIn3024Error

No primary producer is registered for the producer in 3024 A message is sent that the primary producer is missing in 3024. Covered by test-scenario S_{202} and expression P_{204} (true) and test-scenario S_{002} and expression $P_{204}(false)$

ON E-PProducerCheckedIn3024
IF PProducerNotIn3024, AppNotStopped
DO A-Error-NOPrimaryProducer,OKisNo
E-TurDone!

ER-NoProducerTypeIn3020Prod

Covered by test-scenario S_{204} and expression P_{207} (true) and test-scenario S_{002} and expression $P_{207}(false)$

ON E-ProducerTypeCheckedIn3020
IF C-NoProducerTypeIn3020
TELTYPE==PRODTURORDNING
DO A-Error-NOPrimaryProducer,OKisNo

E-TurDone!

6.1.6 Rules Production (RTP)

RTP-CheckProducerIn3032

Covered by test-scenario S_{002} and expression P_{202}

ON CE-TestData2
A-Get-S3032-Producent-M-Plan()
DO E-ProducerCheckedIn3032!

ER-S3036MAXDIFF

Covered by test-scenario S_{401} and expression P_{408} (true) and test-scenario S_{002} and expression $P_{408}(false)$

ON E-S3036-MAX-DIFF-DONE?
IF $E - S3036 - MAX - DIFFRetkod == NO$
OKisNO()
DO E-TurDone!

RTP-GetTurordningslage

Covered by test-scenario S_{002} and expression P_{408} (true) and test-scenario S_{401} and expression $P_{408}(false)$

ON E-S3036-MAX-DIFF-DONE?
IF $E - S3036 - MAX - DIFFRetkod == YES$
A-Turordningslage()
DO E-TurordningslageDone!

RTP-Get-S3026-BatchStorlek

Covered by test-scenario S_{002} and expression P_{004}.

ON E-Get-3026-BatchStorlek?

DO A-Get-S3026-Batchstorlek()
E-Get-S3026-BatchstorlekDone!

RTP-Get-S3054-Senaste-Prodturordning

Covered by test-scenario S_{002} and expression P_{004}

ON E-Get-S3054-senaste-prodturordning?
A-GetS3054SenasteProdturordning()
DO E-SetUTFANTtoPLANT!

RTP-Get-MaxDiffProducer

Covered by test-scenario S_{002} and expression P_{004}

ON E-wantalOK?
A-S33-Get-S3036-Maxdiff()
DO E-S3036-Max-Diff-Done!

RTP-SUM-Wantal

Covered by test-scenario S_{002} and expression P_{004} (true) and test-scenario S_{201} and expression $P_{203}(false)$

ON E-wantal?
IF C-Wantal>=Plant
DO E-wantalOK!

RTP-SetUTFANTtoPLANT

Covered by test-scenario S_{002} and expression P_{403} . **ON** E-SetUTFANTtoPLANT?
DOA-SetUTFANTeqPLANT()
E-TurDone!

RTP-Sum-3018

Covered by test-scenario S_{002} and expression P_{402} . **ON** E-Sum-3018-Extra-Sekvensrad?
DOSUM-3018()
E-Sum-3018-Done!

RTP-Sum-3052-Plangrupp

Covered by test-scenario S_{002} and expression P_{402} . **ON** E-Sum-3052-Plangrupp?
DO SUM-3052-PLANT()

E-Sum-3052-Done!

RTP-UppdatTurordningEnlPlan

Covered by test-scenario S_{002} and expression P_{007} (true) and test-scenario S_{004} and expression $P_{007}(false)$

ON E-ProducerCheckedIn3032?
IF C-TurordnEnlPlan
DO E-Get-S3054-Senaste-prodturordning!

RTP-UppdatTurordningEnlPrioritet

Covered by test-scenario S_{004} and expression P_{007} (true) and test-scenario S_{002} and expression $P_{007}(false)$
ON E-ProducerCheckedIn3032?
IF C-TurordEnlPrioritet()
DO E-SetUTFANTtoPLANT!

RTP-ActualDifferenceNotReached

Covered by test-scenario S_{004} and expression P_{406} (true) and test-scenario S_{402} and expression $P_{406}(false)$

ON CE-KTRL-DIFF-Lage-PLANT?
IF LagePlusPLANT<=S3036Artant
DO E-MaxDiffValuesOK!

RTP-ActualDifferenceReached

Covered by test-scenario S_{402} and expression P_{406} (true) and test-scenario S_{004} and expression $P_{406}(false)$
ON CE-KTRL-DIFF-Lage-PLANT?
IF LagePlusPLANT¿S3036Artant
AERR-FelmedMaxDiffUppnatt()
DO TurDone!

ER-RTP-WANTAL-ERR

Covered by test-scenario S_{201} and expression P_{205} (true) and test-scenario S_{002} and expression $P_{205}(false)$
ON E-wantal?

IF C-Wantal¡PLANT

DO OkIsNo() ErrorTurordningNotEnoughForProducer()
TurDone!

6.1.7 Rules Derived Production (RTH)

RTH-CheckArtAntIn3042

Covered by test-scenario S_{003} and expression P_{205}
ON E-StartTurHProd

DO A-GetDeliveryCondition()
E-GotArtantIn3042!

RTH-CheckLoopCondition

Covered by test-scenario S_{003} and expression P_{301} (true) and test-scenario S_{301} and expression P_{302} (false)
ON E-CheckTurHProdLoop?
IF UTFANT¡PLANT
DO E-SetTurHProdPlanteqUtfant!

RTH-DoHProd

Covered by test-scenario S_{003} and expression P_{301} (true) and test-scenario S_{302} and expression P_{301} (false)
ON E-GorArtAntIn3042?
IF C-DeliveryConditionIn3042
DO TYPE==HPROD
E-CheckTurHProdLoop!

RTH-Looprule

Covered by test-scenario S_{003} and expression P_{301}
ON E-SetTurHProdPlanteqUtfant?
DO E-CheckTurHProdLoop!

RTH-StopLoop

Covered by test-scenario S_{301} and expression P_{302} (true)

ON E-CheckTurHProdLoop?
IF UTFANT>=PLANT
DO E-TurDone!

ER-RTH-CheckPProducerType

Covered by test-scenario S_{302} and expression P_{303} (true) and test-scenario S_{003} and expression P_{303} (false)
ON E-StartTurHProd?
IF PPType ! = PROD AND
PPTYPE ! = HPROD
TELTYPE==PRODTURORDNING
DO TurDone!

ER-RTH-NoDeliveryConditionIn3042

Covered by test-scenario S_{303} and expression P_{303} (true) and test-scenario S_{003} and expression P_{303} (false)
ON E-GotArtantIn3042?
IF C-NoDeliveryConditionIn3042()
TELTYPE == PRODTURORDNING
DO Error-Leveransvillkor-ej-upplagt() TurDone!

6.2 Verification

The verification properties are tested through different scenarios. The behavior depends both on the content of the received telegram and values in database tables. The default database contains ”correct” tables, i.e. all SQL expressions returns result expected for a correct execution. If nothing else is specified, default values in tables are expected. If the database is not default, the deviation from the default database is reported.

6.2.1 Scenario

Scenario S0 Telegramtype = Unknown, database = default.

Scenario S1 Telegramtype = Deliveryorder, default database.

Scenario S2 Telegramtype = Productionorder, Type = Productionorder, order according to plandefault database,.

Scenario S3 Telegramtype = Productionorder, Type = Derived productionorderdefault database, .

Scenario S4 Telegramtype = Productionorder, Type = Productionorder, order according to prioritydefault database,.

Scenario S101 Telegramtype = Deliveryorder, database; intlev exists in table 3043.

Scenario S102 Telegramtype = Deliveryorder, database; intlev does not exists in table 3041.

Scenario S103 Telegramtype = Deliveryorder, database; intlev does not exists in table 3040.

Scenario S104 Telegramtype = Deliveryorder, database; intlev and producer does not exists in a row in table 3042.

Scenario S105 Telegramtype = Deliveryorder, database; producer does not exists in table 3020.

Scenario S201 Telegramtype = Productionorder, Type = Productionorder, order according to plan, database;PLANT ¿ Wantal (Wantal=Sum(Plangrupp in s3042, Extra sequence row in S3018)).

Scenario S202 Telegramtype = Productionorder, Type = Productionorder, order according to plan, database; No PrimaryProducer in S3024

Scenario S203 Telegramtype = Productionorder, Type = Productionorder, order according to plan,database; Producer in S3025 (producerStopped)

Scenario S204 Telegramtype = Productionorder, Type = Productionorder, order according to plan, database; No Producer in S3020

Scenario S205 Telegramtype = Productionorder, Type = Productionorder, order according to plan,database; No PProducer in S3020

Scenario S301 Telegramtype = Productionorder, Type = Derived Productionorder, database;$PLANT < UTFANT$)

Scenario S302 Telegramtype = Productionorder, Type = Derived Productionorder,database; Not correct type for primary producer)

Scenario S303 Telegramtype = Productionorder, Type = Derived Productionorder,database; No delivery condition in 3042)

Scenario S401 Telegramtype = Productionorder, Type = Productionorder, database; order according to priority, Maxdiff returns NO

Scenario S402 Telegramtype = Productionorder, Type = Productionorder, database; order according to priority,ActualDifferenceReached $(Lage + PLANT) > 3036Artant$)

6.2.2 Properties for verifying the RM rules

- Property **P001**: RM-DeliveryProductionOrder
  Pattern: Universality
  Scope: Globally
  P : RM-RequestForDeliveryOrder.EXECUTE
  Comment: IF LEVERANSTURORDNING is message type, this expression shall return T, else F
- Property **P002**: RM-ExecuteRequestForProductionOrder
  Pattern: Universality
  Scope: Globally
  P: RM-RequestForProductionOrder.EXECUTE
  Comment: Checks if rule RM-RequestForProductionOrder executes. If type in telegram is PRODTURORDNING it returns true, else false
- Property **P003**: ER-UnknownTelegramTypeExecute
  Pattern: Universality
  Scope: Global
  P: ER-UnknownTelegramType.EXECUTE
  Comment: Returns true if telegramtype is unknown.
- Property **P004**: Ready
  Pattern: Universality
  Scope: Globally
  P: RM-Ready.EXECUTE and UTFANT == PLANT and FELMED == EMPTY
  Comment: Returns true if RM-ReadyExecute and $UTFANT == PLANT$ and NoErrorMessages has been sent.

6.2.3 Expressions for verifying the RMD rule set

- Property **P101**: RD-CE-Create-Deliveryorder-OCCUR
  Pattern: Universality
  Scope: Globally
  P : CE-Create-Deliveryorder.OCCUR
  Comment: Expected result is T for Scenario S_{001}.
- Property **P102**: RD-CE-RD-UtfBearb-OCCUR
  Pattern: Universality
  Scope: Global
  P : CE-RD-UtfBearb.OCCUR
  Comment: Expected result is T for Scenario S_{001}.
- Property **P103**: ER-LeverantorStoppad
  Pattern: Universality
  Scope: Global
  P : LeverantorStoppad
  Comment: Expected result is F for Scenario S_{001}.

- Property **P104**: ER-NoProducerTypeIn3020LEV
  Pattern: Universality
  Scope: Global
  P : ER-NoProducerTypeIn3020LEV.EXECUTE
  Comment: Expected result is F for Scenario S_{001}.

- Property **P105**: DeliveryOrderNoIntlev
  Pattern: Universality
  Scope: Global
  P : ER-NoIntlevIn3041.EXECUTE
  Comment: Expected result is T for Scenario S_{102}.
  Checks that the error detection rule is executed when intlev is missing in 3041

- Property **P106**: DeliveryOrderUtforBearb
  Pattern: Universality
  Scope: Globally
  P : RD-Utfor-Bearb.EXECUTE
  Comment: Expected result is T for Scenario S_{001}.
  Checks that the construction of delivery order starts given a correct database

- Property **P107**: DeliveryOrderNoIntlevIn3040
  Pattern: Universality
  Scope: Globally
  P : ER-NoIntlevIn3040.EXECUTE
  Comment: Expected result is T for Scenario .
  Checks that the error detection rule is executed when intlev is missing in 3040

- Property **P108**: ER-RD-NoDeliveryConditionExecute
  Pattern: Universality
  Scope: Globally
  P : ER-NoDeliveryConditionIn3042.EXECUTE
  Comment:Checks that the error detection rule is executed when deliveryCondition is missing in 3042

- Property **P109**: CheckGenerateDeliveryOrderExec
  Pattern: Universality
  Scope: Globally
  P : RMD-CheckGenerateDeliveryOrder.EXECUTE
  Comment: Expected result is T for Scenario S_{001}.
  Checks that the rule RMD-CheckGenerateDeliveryOrder is executed in default scenario

6.2.4 Expressions for verifying the RMP rule set

- Property **P201**: TurProdExecute
  Pattern: Universality
  Scope: Global
  P : RMD-TurProd.EXECUTE
  Comment: Expected result is T for Scenario S_{004}.
  Checks that the rules for production order starts to execute

- Property **P202**: TurHProdExecute
  Pattern: Universality
  Scope: Global
  P : RMD-TurHProd.EXECUTE
  Comment: Expected result is T for Scenario S_{002}.
  Checks that the rules for derived production order starts to execute

- Property **P203**: ER-ProducerStoppedExecute
  Pattern: Existence
  Scope: Global
  P : ER-ProducerStopped.EXECUTE
  Comment: Expected result is F for Scenario S_{004}.

- Property **P204**: CE-ProducerIn3020AndNotStoppedOccur
  Pattern: Universality
  Scope: Global
  P : CE-ProducerIn3020AndNotStopped.OCCUR
  Comment: Expected result is F for Scenario S_{004}.

- Property **P205**: ER-RTP-WANTAL
  Pattern: Universality
  Scope: Global
  P : FELMED == ERROR-TURORDNING-RACKER-EJ-FOR-PROUCENT
  Comment: Expected result is T for Scenario S_{202}.

- Property **P206**: NoPrimaryProducerIn3024
  Pattern: Existence
  Scope: Global
  P : ER-NoPProducerIn3024.EXECUTE
  Comment: Expected result is T for Scenario S_{203}.

- Property **P207**: NoProducertypeIn3020
  Pattern: Existence
  Scope: Global
  P : ER-NoProducerTypeIn3020PROD.EXECUTE
  Comment: Expected result is T for Scenario S_{204}.

6.2.5 Expressions for verifying the RTH rule set

- Property **P301**: RTH-CheckLoopConditionExecute
  Pattern: Universality
  Scope: Global
  P : RTH-CheckLoopCondition.EXECUTE
  Comment: Expected result is T for Scenario S_{002}.

- Property **P302**: RTH-StopLoopExecute
  Pattern: Universality
  Scope: Global
  P : RTH-StopLoop.EXECUTE
  Comment: Expected result is T for Scenario S_{002}.

- Property **P303**: RTH-ER-CheckPProducerType
  Pattern: Universality
  Scope: Global
  P : ER-RTH-CheckPProducerType.EXECUTE
  Comment: Expected result is T for Scenario S_{002} and type of PProducer is not PROD or HPROD

- Property **P304**: ER-RTH-NoDeliveryCondition
  Pattern: Universality
  Scope: Global
  P : ER-RTH-NoDeliveryCondition.EXECUTE
  Comment: Expected result is T for S_{002} and no tuple with producer and intlev in 3042

6.2.6 Expressions for verifying the RTP rule set

- Property **P401**: RTP-CheckProducerIn3032EXECUTE
  Pattern: Universality
  Scope: Global
  P : RTP-CheckProducerIn3032.EXECUTE
  Comment: Expected result is T for Scenario S_{004} and default database

- Property **P402**: CE-WANTAL-TRIGGERED
  Pattern: Universality
  Scope: Global
  P : CE-WANTAL.EXECUTE
  Comment: Expected result is T for Scenario S_{004} and default database

- Property **P403**: RTP-SetUtfantToPlant
  Pattern: Universality
  Scope: Global

P : RTP-SetUtfantToPlant.EXECUTE
Comment: Expected result is T for Scenario S_{004} and default database

- Property **P404**:ER-RTP-WANTAL-EXECUTE
Pattern: Existence
Scope: Global
P : RTP-ER-RTP-Wantal.EXECUTE
Comment: Expected result is T where Plant ¡ wantal

- Property **P405**:RTP-ActualDifferenceReachedEXECUTE
Pattern: Existence
Scope: Global
P : RTP-ActualDifferenceReached.EXECUTE
Comment: Expected result is F for Scenario S_{004}

- Property **P406**:RTP-UppdatTurordningEnlPrioritetEXECUTE
Pattern: Existence
Scope: Global
P : RTP-UppdatTurordningEnlPrioritet.EXECUTE
Comment:

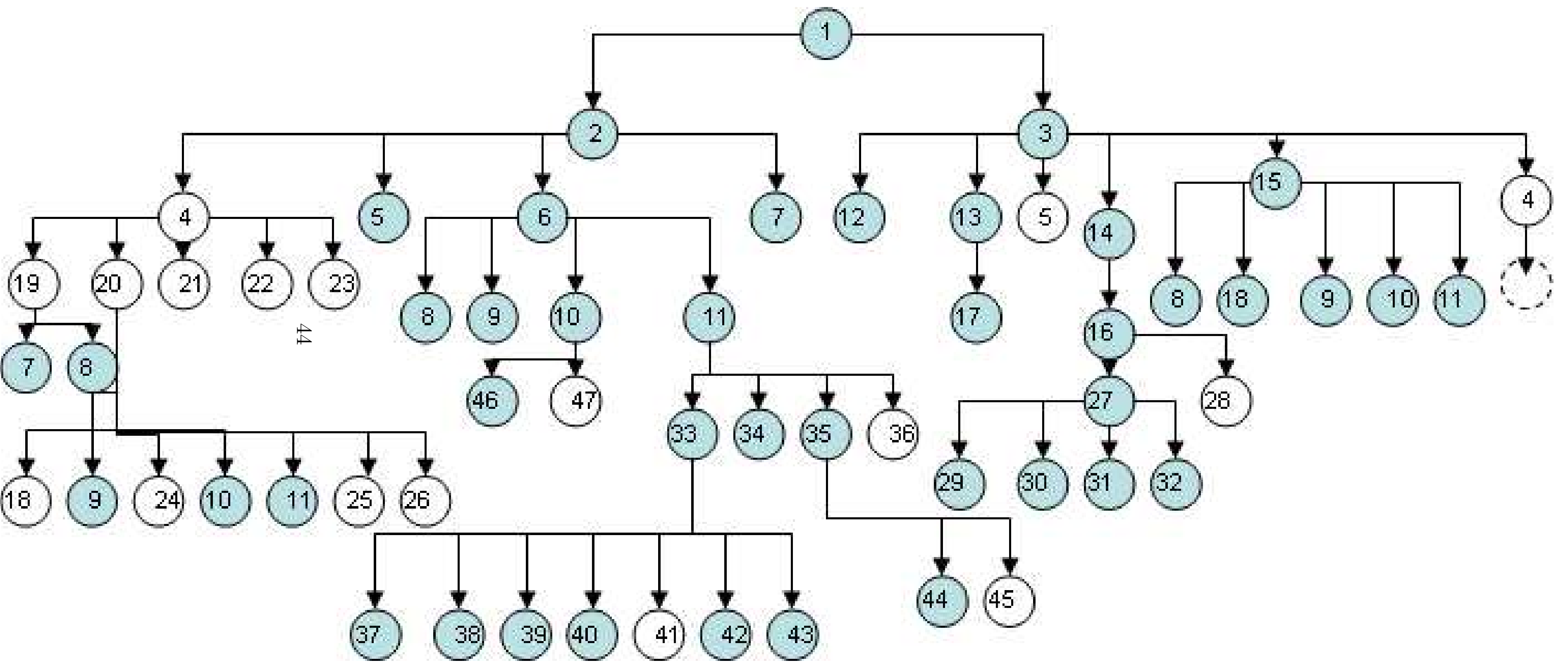
- Property **P407**:RTP-UppdatTurordningEnlPlanEXECUTE
Pattern: Existence
Scope: Global
P : RTP-UppdatTurordningEnlPlan.EXECUTE

- Property **P408**:RTP-ERMaxdiffEXECUTE
Pattern: Existence
Scope: Global
P : RTP-ER-MAXDIFF.EXECUTE

6.3 Running verification expressions on scenarios

S/ P	000	001	002	003	004	101	102	103	104	105	201	202	203	204	205	301	302	303	401	402
P_{001}	**F**	**T**	**F**	**F**	**F**	**T**	**T**	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{002}	**F**	**F**	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
P_{003}	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{004}	**F**	**T**	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{101}	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{102}	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**T**	**T**	**F**	**F**
P_{103}	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{104}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{105}	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{106}	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{107}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{108}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{109}	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{201}	**F**	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**T**
P_{202}	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**T**	**T**	**F**	**F**
P_{203}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{204}	**F**	**F**	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**T**	**T**	**F**	**F**	**T**	**T**	**T**	**T**	**T**	**T**
P_{205}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**
P_{206}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{207}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**
P_{208}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**
P_{209}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**
P_{301}	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**
P_{302}	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**
P_{303}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**
P_{304}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**
P_{401}	**F**	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{402}	**F**	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**T**
P_{403}	**F**	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{404}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**
P_{405}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{406}	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{407}	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
P_{408}	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**T**	**F**

Table 6.1: Result from running verification.



Bibliography

[1] 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.

[2] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 26, pp. 183–235, 1994.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] M. Berndtsson and B. Calestam, "Graphical notations for active rules in uml and uml-a," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 2, p. 2, 2003.