MOGENTES

Model-Based Generation of Test-Cases for Embedded Systems

Ontology Based Model Verification

Final Version

<table>
<thead>
<tr>
<th>Project</th>
<th>MOGENTES</th>
<th>Contract Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Id</td>
<td>3-14_1.4r</td>
<td>216679</td>
</tr>
<tr>
<td>Date</td>
<td>2011-03-25</td>
<td>Deliverable</td>
</tr>
<tr>
<td>Deliverable</td>
<td>3.3b</td>
<td></td>
</tr>
<tr>
<td>Contact Person</td>
<td>Balázs Polgár</td>
<td>Organisation</td>
</tr>
<tr>
<td>BME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>+36-1-4633579</td>
<td>E-Mail</td>
</tr>
<tr>
<td><a href="mailto:polgar@mit.bme.hu">polgar@mit.bme.hu</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Distribution Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Company</th>
<th>Department</th>
<th>No. of copies</th>
<th>Hardcopy/Softcopy</th>
</tr>
</thead>
<tbody>
<tr>
<td>All MOGENTES team members</td>
<td></td>
<td></td>
<td>1</td>
<td>SC</td>
</tr>
</tbody>
</table>

### Change History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason for Change</th>
<th>Pages Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2010-06-01</td>
<td>Initial version based on D3.3a</td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>2010-06-22</td>
<td>Chapter 2 is updated and Chapter 4 is added</td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>2010-06-25</td>
<td>Introduction and Chapter 5 is updated</td>
<td></td>
</tr>
<tr>
<td>1.0d</td>
<td>2010-06-30</td>
<td>Style improvement based on ETH feedback</td>
<td></td>
</tr>
<tr>
<td>1.0r</td>
<td>2010-06-30</td>
<td>Feedback from AIT included; Release</td>
<td></td>
</tr>
<tr>
<td>1.1w</td>
<td>2011-01-20</td>
<td>Chapter about model metrics included</td>
<td></td>
</tr>
<tr>
<td>1.2w</td>
<td>2011-03-21</td>
<td>Model metrics chapter improved, introduction updated</td>
<td></td>
</tr>
<tr>
<td>1.3r</td>
<td>2011-03-25</td>
<td>Document released</td>
<td></td>
</tr>
<tr>
<td>1.4r</td>
<td>2011-05-04</td>
<td>Release updated</td>
<td></td>
</tr>
</tbody>
</table>
Contents

1 Introduction ...........................................................................................................................................5
  1.1 Purpose ...............................................................................................................................................5
  1.2 Constraint Checking Solutions ...........................................................................................................6
  1.3 The Model Pre-Checking Verification Approach Applied in MOGENTES ............................................8
  1.4 Scope ..................................................................................................................................................9
  1.5 Related Project Materials ..................................................................................................................9
  1.6 Structure of the Document .................................................................................................................9

2 Verification of Application Specific UML Models ............................................................................11
  2.1 Verification of Class Diagram Elements ............................................................................................11
  2.2 Verification of State Machine Elements ............................................................................................15
  2.3 Verification of Behaviour Definition ...................................................................................................16
  2.4 Verification of OCL Constraints .........................................................................................................17
  2.5 Application Specific Verification .......................................................................................................19
  2.6 TCG Specific Verification ..................................................................................................................19

3 Ontology Based Verification ...........................................................................................................21
  3.1 Introduction .......................................................................................................................................21
  3.1.1 Ontologies .....................................................................................................................................21
  3.1.2 Description Logic ...........................................................................................................................21
  3.1.3 DL Families ....................................................................................................................................21
  3.1.4 Tools .............................................................................................................................................22
  3.2 Description Logic Formalism ..............................................................................................................22
  3.3 Reasoning Tasks ..................................................................................................................................24
  3.3.1 Inference Services Offered by RacerPro .......................................................................................25
  3.3.2 Concept Level Consistency Checks in RacerPro ..........................................................................25
  3.4 The Role of Ontologies in Model Checking .......................................................................................26
  3.5 Mapping UML to DL ............................................................................................................................26
  3.5.1 Metamodel (TBox) Level ...............................................................................................................26
  3.5.2 Model (ABox) Level ......................................................................................................................27
  3.5.3 Constraint Representation in RacerPro Syntax ..............................................................................27
  3.5.4 Transformation in VIATRA2 .........................................................................................................28

4 Implementation ....................................................................................................................................29
  4.1 Process of the Verification ...................................................................................................................29
  4.1.1 Transformation of UML Models ....................................................................................................29
  4.1.2 The Checker Component ..............................................................................................................31
  4.2 Execution of the Verification Process ..................................................................................................34
  4.3 Verification of the Constraints ...........................................................................................................35

5 Metrics Based Testability Assessment ..............................................................................................38
  5.1 Workflow of Model Assessment ........................................................................................................39
  5.2 The Computed Model Metrics ..........................................................................................................40
  5.2.1 Class diagram metrics .................................................................................................................40
  5.2.2 State machine metrics ..................................................................................................................42
  5.2.3 Computation of metrics for the selection of the test generation method ......................................42
  5.3 The Implemented Ontology Based Model Assessment Environment .............................................43

6 Verification of Process Models ........................................................................................................46
  6.1 Scope ..................................................................................................................................................46
6.2 Formalisation of the Requirements ................................................................. 46
6.3 Classification of Tools in the Tool Repository ............................................. 48
6.4 Modeling the Development Process ............................................................ 49
6.5 Assessment of Development Toolchains ..................................................... 49

7 Summary and Outlook ....................................................................................... 51

8 Abbreviations and Definitions ......................................................................... 52

9 References ......................................................................................................... 53

---

### Figures

- Figure 4-1: The process of the ontology based verification ........................................ 29
- Figure 4-2: The model transformation process ...................................................... 30
- Figure 4-3: Example configuration file for constraint checking ................................. 32
- Figure 4-4: Example output of constraint checking ............................................... 33
- Figure 4-5: The transformation process realized in the framework ......................... 34
- Figure 4-6: The checking process realized in the framework ................................... 35
- Figure 5-1: Workflow of metrics based orientation of TCG tools ............................. 40
- Figure 5-2: The process of the ontology based computation of model metrics .......... 43
- Figure 5-3 Architecture of the verification tool .................................................... 44
- Figure 5-4: Example output of metric based assessment ......................................... 45
- Figure 6-1: Verification and Testing methods for SIL4 (EN50128) .......................... 47
- Figure 6-2: Verification and Testing methods ontology from EN50128 .................... 48
- Figure 6-3: The assessment process ....................................................................... 50

---

### Tables

- Table 1-1: Comparison of model representation and their verification techniques ................................................................. 7
- Table 3-1: The DL notation and the RacerPro (KRSS) syntax of the SHIQ logic .......... 23
- Table 3-2: The DL notation and the RacerPro syntax of the concept axioms ............... 24
- Table 6-1: The Verification and Testing methods (EN50128) .................................... 47
1 Introduction

1.1 Purpose

The aim of the MOGENTES project is efficient model-based test case generation. Test case generation needs a sound basis in the terms of well-defined, consistent modelling languages and correct models that satisfy certain static constraints. These constraints can be for instance well-formedness rules, domain constraints or other modelling restrictions.

MOGENTES aimed at the reuse of existing engineering languages as front-end, thus no new modelling languages have been developed. Thus, widely used and accepted modelling languages (UML and Simulink) have been chosen as front-end.

Internally, action systems and timed automata have been selected as the direct back-end input for test case generation as these have formal semantics and form an appropriate basis for rigorous mathematical calculations.

The lower level, back-end models are generated by mapping the concepts of the front-end models to their respective low-level equivalent formalisms by automatically transforming the application models to the appropriate syntax.

However, this two-step approach also suffers of some drawbacks, as well. The majority of engineering modelling languages was constructed by taking the comfort of modelling, thus richness of constructs as primary objective. This way, a large number of modelling construct should be transformed to the low level of notations which exceeds the frames of the conceptual research. This way, in MOGENTES a core modelling sub-language was defined in order to keep the pure technical work at a reasonable level.

Testing has to decide on the correctness of a model and its implementation. A major problem related to engineering modelling languages (and especially UML) is the lack of an unambiguous formal semantics, thus a selected precise interpretation is needed.

Numerous checks are needed prior of running the rather costly test generation algorithms in order to avoid problems originating in ill-formed or practically untestable input test objects:

- Obviously, the model to be tested has to conform to all syntactic restrictions as defined in the meta-model possibly accomplished by other formal requirements.
  
  The test generation process is in general vulnerable to ill-formed models, for instance a model violating syntactic roles may lead to a crash in the TCG phase or prohibit the termination of the process.

- It is quite common in safety engineering to put additional syntactic well-formedness rules onto a general purpose modelling language in order to exclude the use of risky constructs.
  
  Such restrictions serve for instance the exclusion of such model parts which conform to the syntax constraints of the host language but lead to a non-deterministic behaviour in the implementation.

  A typical group of such requirements prescribes the completeness of the specification at the model level meaning that in each situation the behaviour has to be completely specified. Otherwise the behaviour may depend on the model to implementation transformation technology or what's even worse a non-deterministic run-time behaviour can be the result.

  From the point of view of testing, non-determinism clearly prohibits to bring up even approximate guarantees on the effectiveness of testing in the terms of coverage prediction (a successful run of a test pattern not indicating any faults is insufficient to state the absence of the faults detected by the test as due to the non-determinism the test generated are only potentially fault detecting ones).

  Similarly, non-determinism in such cases prohibits the reproduction of faults effects detected thus leading to a potentially broken error detection – fault diagnosis – correction – re-testing loop.

- Even a model conforming to all syntactic and semantic design constraints has no guarantee to be practically testable. Some hard to test parts in the model may lead to such long and memory intensive TCG runs which violate the usual practical run-time and storage complexity constraints of the TCG process.
Hardware testing realised this phenomenon already decades ago and extended the design paradigm and environment with testability requirements.

This way, a product from the design workflow has to be packed on the conformance of testability rules and metrics prior of entering any automated test generation process. A model violating testability requirements is simply rejected of TCG even if it is otherwise fully compliant with the design rules.

The same principle was gradually adapted by the software testing community, as well.

- Testability metrics are orienting in the selection of a particular candidate test program generation method. As usual, no single silver bullet like solution exist that would fit to all of the test generation problems. Testability metrics assessing for instance the sequentiality level of a program or the complexity of its potential control flow are important guidelines to select between test generation algorithms fitting more to either control or data intensive applications.

Accordingly, all the input models shall be checked on the conformance to all the syntactic and semantic design and testability constraints and restrictions in order to guarantee a successful application of the test case generation techniques.

A joint characteristics of the checks and calculation above is that they can be performed (or the majority of them) in a static way on the model.

- A part of them requires the checking of different complex constraints over an object-oriented model. The constraints here can be described best in the terms of the meta-model (for instance no two concurrent state transitions originating in the same initial state and having no trigger condition may occur in order to avoid non-determinism).

- Testability metrics access the complexity of the typed graph structure of the model thus they necessitate calculations after identification of the model structure.

The next sub-section shortly evaluates the main implementation guidelines of the implementation of the checking tasks sketched above.

### 1.2 Constraint Checking Solutions

The formulation and checking of the constraints and the calculation of the metrics can be performed by using different representations of the models. Regardless of the approach used two general alternative solution mechanisms exist:

- **Tight coupling**, in which case constraint checking is part of modelling.
  
  Constraints can be formulated as rules embedded into the syntax of the modelling language; in this case constraints are checked during model build time by the editor, thus inconsistent models are not allowed to be created at all.

- **Loose coupling**, in which case constraint checking is an additional follow-up to modelling.
  
  Given the representation format of models to be tested, the constraints can be formulated as queries in a corresponding formalism; in this case the constraint check is the execution of the query on a model instance resulting in the list of elements that violate some constraint.

In the following we summarize the most common solutions that can be used for representing models:

- The model can be represented as a set of runtime objects; the elements of the language are defined either by self-defined classes or by using a modelling framework. A widely used framework for it is EMF (Eclipse Modeling Framework, [EMF]),
- The models can be represented as an ontology describing the concepts and their relations using a formalism, which is developed natively for knowledge representation,
- Models can be represented as typed graphs,
- Models can also be stored in a database; the database can be either a traditional relational DB, or an object oriented database,
- The models can also be represented by other special formalisms needed by special tools.
<table>
<thead>
<tr>
<th>Model representation</th>
<th>Ad-hoc check on runtime objects</th>
<th>Framework support on runtime objects</th>
<th>Ontology</th>
<th>Typed graphs</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes in a specific programming language</td>
<td>Dedicated modelling language with class representation in a specific programming language</td>
<td>Knowledge base (concepts + their relations)</td>
<td>Typed nodes and their relations</td>
<td>DB tables and their relations</td>
<td></td>
</tr>
<tr>
<td>Constraint representation</td>
<td>Custom program code</td>
<td>Rules on models or model query (can be dedicated language or program code)</td>
<td>Knowledge base assertions or query</td>
<td>Pattern matching</td>
<td>DB query</td>
</tr>
<tr>
<td>Constraint verification</td>
<td>Loose</td>
<td>Tight / Loose</td>
<td>Tight / Loose</td>
<td>Loose</td>
<td>Loose</td>
</tr>
<tr>
<td>Advantage</td>
<td>• Rapid development</td>
<td>• Strong support for generic verification tasks</td>
<td>• Models have precise static semantics</td>
<td>• Efficient pattern matchers exist</td>
<td>• Large models can be supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Integrated with modelling framework</td>
<td>• Consistency check of constraints (assertions) is possible</td>
<td>• Can be the base for integrating different environments</td>
<td>• Technology is mature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Dedicated constraint/rule language</td>
<td>• Verification can be reused for several different modelling environments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drawback</td>
<td>• Programming is needed to add new constraint</td>
<td>• Mainly programming is needed for constraint checking</td>
<td>• Transformation is needed to map original models to the ontology based representation</td>
<td>• Mainly academic tools exist</td>
<td>• RDBMS is not well suited for models</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Everything shall be programmed manually</td>
<td>• Limited efficiency</td>
<td></td>
<td>• OODBMS is not efficient</td>
</tr>
<tr>
<td>Example</td>
<td>Own program</td>
<td>EMF + EMF Validation / EMF Query</td>
<td>OWL + RacerPro</td>
<td>VIATRA modelling and model transformation framework</td>
<td>Oracle, MS-SQL, MySQL</td>
</tr>
</tbody>
</table>

Table 1-1: Comparison of model representation and their verification techniques

Each model representation has its own technique for formulating the constraints and checking them:
- If the model is a set of runtime objects (either using a dedicated representation or a modelling framework), ad-hoc checks can be implemented by writing specific code (in the particular programming language). In this case the checking means the execution of the checker program on the runtime instance of the model.
If a modelling environment is used, a related validation framework can be used; in case of EMF e.g. the EMF Query can be used for retrieving model elements that violate certain constraints.

- If model is represented as an ontology, either assertions can be added to the knowledge base (tight coupling) or the constraints can be formulated as queries on the knowledge base (loose coupling), e.g. using nRQL. In the former case the constraints are checked as the model is built, in the latter case checking is executed in a separate step.
- If the model is handled as a graph, typically constraints are formulated as graph patterns and constraint checking is realized as pattern matching searching for patterns not conforming to the requirements.
- If the model is stored in a database the compliance to the constraints can be checked by means of a database query language (e.g. SQL) in the form of searching patterns violating the constraints.

The different approaches are compared in Table 1-1.

### 1.3 The Model Pre-Checking Verification Approach Applied in MOGENTES

Constraints and restrictions can be defined either on the front-end (i.e., in UML or Simulink) models or on the back-end (i.e., on actions systems or timed automaton) models. The advantage of the formulation at the higher abstraction level is to be closer to the domain thinking. Errors identified by checking can be directly interpreted on the models created by the modeller.

The advantage of the latter is that checking is applied on the direct input of the test case generation, thus

- Requirements which are hard to express at the high-level for instance such ones which prescribe the determinism of the run-time behaviour can be included.
- Less important in the designated final technology; however, of significance during technology development time, errors introduced by the complex chain of transformations in the MOGENTES tooling can be detected.

In the project we have decided to focus on the checking of the front-end (domain) models, as

i) this is the direct input of the test case generation,

ii) the problems found by the checks can easily be interpreted by the modeller and the related model elements violating the constraints can be accessed directly in order to correct the problem.

From the possible verification techniques we have chosen the ontology based verification approach, as

i) it is well suited to the semantic checking of modelling languages and model instances by means of standard query languages and off the shelf ontology reasoners,

ii) it can be reused in several modelling environments, i.e. it is possible to use the same technology for both the UML and the Simulink track,

iii) graph based tools are not as mature as COTS ontology reasoners,

iv) there is no standard constraint description formalism that can be reused across several modelling languages (with the exception of the still immature OCL).

Note, that the ontology based approach fundamentally exploits the correspondence between the meta-model language and the ontology describing its notions. In our approach we look more at the technical side by reusing the underlying query and reasoning technology for checking and testability assessment purposes and do not really enter the field of ontology engineering in the traditional sense.

The ontology based approach could be used as a uniform technology for both the UML and the Simulink tracks. We have chosen to apply it to the UML track as UML models (due to the flexibility of the language and the more loose semantics) tend to require more consistency checks than Simulink models. Note, however, that the checks developed for UML models demonstrate the high-level representation of the constraints which makes possible the reusability of the technology for other modelling languages as well.
1.4 Scope

The work of this deliverable is primarily related to the application models that are modelled using the UML formalism. Some features are selected and verified using the ontology based verification approach to provide well-formed and well-defined, sound models as inputs to test case generation.

In addition to assuring that constraints are satisfied in a model, the complexity of the model can also influence the effectiveness of a given TCG method, and even some properties could affect the applicability of a given approach. Properties of models (like complexity) can be characterized by model metrics, and the values of these metrics can provide hints for the test engineer how to configure the test case generation (which method to use, how to modify the model to be better suited for TCG, what parameter values of the TCG tool shall be used for the given model). Accordingly, the ontology based verification environment is extended to support the computation of model based metrics to be assessed by the test engineer before the configuration of the TCG methods.

In addition, the ontology based verification approach is also applied to process models that are defined and used in WP2 for the execution of transformations and test case generation. These processes are verified against requirements (constraints) defined in standards related to development processes. These typically require the existence of certain kinds of development or verification tools to achieve a certain safety integrity level (e.g. SIL 3 or SIL 4). This solution also provides means for tracing the reasons behind application of a given tool or methodology in the development, which is a goal of the tool integration framework. This extension demonstrates that the ontology based static checking can be re-used at the level of process engineering and – being in line with the objectives of the project – can contribute to the standards-based assessment and potential certification of modelling and test generation processes.

This document is the updated version of the deliverable D3.3a:

- the constraints are updated according to the changes in the modelling approach,
- the implementation of the ontology based verification approach is added,
- the integration of the approach to the tool integration framework is described,
- the metrics based model assessment approach is added,
- the description of the verification of process models is updated.

1.5 Related Project Materials

The following project materials are related to this deliverable:

- Deliverable D3.2 Modelling Languages contains details about the way UML is used for modelling applications.
- Deliverable D2.1 Framework Specification and deliverables D2.2a First Framework Implementation, D2.2b Pre-final Framework Implementation contains details about the tool integration framework to which the ontology based verification tool is integrated and information about test generation processes that can be verified against requirements in standards.

1.6 Structure of the Document

The deliverable is structured as follows:

- Sec. 2 provides a list of the properties or constraints that shall be verified or checked using the ontology based verification approach,
- Sec. 3 gives an introduction to the ontologies and description logics and discusses what type of verification can be performed in these domains and how UML models can be mapped to description logic languages,
- Sec. 4 describes the implementation of the ontology based verification tool, the integration of it into the framework and some considerations about the implementation of the constraints described in Sec. 2,
- Sec. 5 outlines the model metrics based assessment approach that can orient the application of the test case generation, describes some model metrics that are relevant from the testability point of view and describes the extension of the verification environment for the metric evaluation and assessment tasks,
• Sec. 6. discusses how requirements related to process models can be formalized, how development processes can be modelled and how the assessment of the tool-chains defined by the process models can be performed.
2 Verification of Application Specific UML Models

In this section we give a list of properties or constraints related to the application models that should be verified or checked.

2.1 Verification of Class Diagram Elements

<table>
<thead>
<tr>
<th>Goal</th>
<th>Metamodel is consistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_C1</td>
</tr>
<tr>
<td>Type</td>
<td>Consistency check</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>A metamodel is inconsistent if no instance model exists that satisfies the class descriptions (e.g. there is multiplicity conflict), i.e. it is not possible to create a model which satisfies all the constraints defined in the metamodel.</td>
</tr>
<tr>
<td>Output</td>
<td>List of inconsistencies in the metamodel.</td>
</tr>
<tr>
<td>Related elements</td>
<td>Classes and their relations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal</th>
<th>Conformance of an instance model to its metamodel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_C2</td>
</tr>
<tr>
<td>Type</td>
<td>Consistency check</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>The elements in the instance model shall instantiate only those types that are defined in the metamodel and the links between them shall also correspond to the relations defined in the metamodel.</td>
</tr>
<tr>
<td>Output</td>
<td>List of inconsistencies in the instance model.</td>
</tr>
<tr>
<td>Related elements</td>
<td>Classes with their relation and the objects with links between them.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal</th>
<th>Consistency of the instance model with respect to the metamodel ontology</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_C3</td>
</tr>
<tr>
<td>Type</td>
<td>Consistency check</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>The elements and their relations in the instance model shall satisfy the constraints defined in the metamodel.</td>
</tr>
<tr>
<td>Output</td>
<td>List of inconsistencies in the instance model.</td>
</tr>
<tr>
<td>Related elements</td>
<td>Classes with their relation and the objects with links between them.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal</th>
<th>Coherence of the instance model</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_C4</td>
</tr>
<tr>
<td>Type</td>
<td>Consistency check</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>The instance model shall contain no contradictions.</td>
</tr>
<tr>
<td>Output</td>
<td>List of inconsistencies in the instance model.</td>
</tr>
</tbody>
</table>
### Related elements
Objects with links between them.

### Goal
State machine diagram exists for all active classes

<table>
<thead>
<tr>
<th>ID</th>
<th>C_C5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Completeness</td>
</tr>
</tbody>
</table>

**Description / Rationale**

‘An active object is an object that […] commences to execute its classifier behavior’ [UML 2.1.2 p.436], thus it shall have a classifier behavior. In MOGENTES we define this as a StateMachine and it has to be checked whether it is not forgotten.

**Output**
List of active classes that do not have a state machine.

**Related elements**

Class where the isActive attribute is true has oneStateMachine as ownedBehavior

### Metamodel pattern

![Metamodel pattern diagram](image)

### Goal
Coverage of all non-output signals by at least one transition trigger

<table>
<thead>
<tr>
<th>ID</th>
<th>C_C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Completeness / cleanness</td>
</tr>
</tbody>
</table>

**Description / Rationale**

Existence of an input signal without triggering a transition in a state machine can have two causes:

1. It is forgotten to process the signal.
2. The signal is unnecessary.

In the former case one of the state machines shall be extended, in the latter the deletion of the signal is suggested.

**Output**
List of non-output signals that do not trigger any transition.

**Related elements**

A Transition contains triggers that reference Events (event property) which can be a SignalEvent. A SignalEvent references a Signal (signal property). This Signal is not received by an environment class, i.e., there is no Reception in a Class with stereotype <<environment>> that receives this Signal (appliedStereotype property).
<table>
<thead>
<tr>
<th>Goal</th>
<th>Coverage of all signals by at least one transition trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_C7</td>
</tr>
<tr>
<td>Type</td>
<td>Completeness / cleanness</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>In comparison to the previous constraint this one also forces processing of the output signals (i.e. to have a state machine where a transition is triggered). This is needed for the UPPAAL based test case generation, while the AS based TCG works without it.</td>
</tr>
<tr>
<td>Output</td>
<td>List of signals that do not trigger any transition.</td>
</tr>
<tr>
<td>Related elements</td>
<td>A Transition contains triggers that reference Events (event property) which can be a SignalEvent. A SignalEvent references a Signal (signal property).</td>
</tr>
</tbody>
</table>
**Goal**
Coverage of all non-input signals by at least one send signal AGSL statement

**ID**
C_C8

**Type**
Completeness / cleanness

**Description / Rationale**
Having an output or internal signal without a send signal statement can have two causes:
1. It is forgotten to generate the signal.
2. The signal is unnecessary.
In the former case one of the state machines shall be extended, in the later the deletion of the signal is suggested.

**Output**
List of signals that are not sent by any AGSLSendStatement and are not received from the environment.

**Related elements**
Signal has stereotype <<from_environment>>, or is referenced by a Reception (signal property) that has a stereotype <<from_environment>>, or is referenced from an AGSLSendStatement (sentEventID property references an AGSLQualifiedIdentifier, the first element of its members property references an AGSLIdentifier, and its referencedElement property references the Signal).

**Metamodel pattern**
## 2.2 Verification of State Machine Elements

<table>
<thead>
<tr>
<th>Goal</th>
<th>Each state is targeted by at least one transition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_SM1</td>
</tr>
<tr>
<td>Type</td>
<td>Completeness</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>States without ‘in-transitions’ are not reachable. Note that initial states have an incoming transition from the initial pseudo-state.</td>
</tr>
<tr>
<td>Output</td>
<td>List of states that are not the destination of any transition.</td>
</tr>
<tr>
<td>Related elements</td>
<td>A State shall have at least one input Transition (incoming property) or a nested Region of the state (region property) shall contain a State (subvertex property) with at least one incoming transition.</td>
</tr>
</tbody>
</table>

### Metamodel pattern

![Metamodel Diagram]

<table>
<thead>
<tr>
<th>Goal</th>
<th>State machines should be deterministic (optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_SM2</td>
</tr>
<tr>
<td>Type</td>
<td>Consistency</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>In all states of the state machines the processing of events is deterministic, i.e., there are no multiple transitions starting from the same state that have the same trigger and guard. It means that there are no conflicts that are not resolved by the priority scheme of UML. Namely, conflicts among different hierarchy levels are always resolved as the transition with source state at the lowest level fires. Conflicts not resolved by the priority scheme, thus requiring non-deterministic choice, occur only at the same hierarchy level. In this constraint the problem can be determining whether two guards are equivalent or not since the constraints can be expressed in several ways (see the metamodel pattern below). Thus a warning will be given if transitions of a state are triggered by the same event (without comparing their guards).</td>
</tr>
<tr>
<td>Output</td>
<td>Sets of transitions for all states that are triggered by the same event.</td>
</tr>
<tr>
<td>Related elements</td>
<td>All output Transitions of a State (outgoing property) shall be compared if both have a Trigger (trigger property) that is related to the same Event (event property).</td>
</tr>
</tbody>
</table>
### 2.3 Verification of Behaviour Definition

<table>
<thead>
<tr>
<th>Goal</th>
<th>Sufficient method definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_BD1</td>
</tr>
<tr>
<td>Type</td>
<td>Completeness</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>The effects of all methods within the tested system should be sufficiently described. This means that the behaviour defined by an operation shall be defined. In MOGENTES this is defined in the AGSL language.</td>
</tr>
<tr>
<td>Output</td>
<td>List of operations the behaviour of which is not defined.</td>
</tr>
<tr>
<td>Related elements</td>
<td>All Operations shall have an OpaqueBehavior as defining method with language “AGSL” and with a nonempty body.</td>
</tr>
</tbody>
</table>
### Metamodel pattern

<table>
<thead>
<tr>
<th>ID</th>
<th>Description / Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_BD2</td>
<td>Where the effect of a transition in a state machine is defined, it shall use the AGSL language formalism.</td>
</tr>
</tbody>
</table>

### Goal
- Sufficient effect definition

### Type
- Completeness

### Output
- List of Transitions and related effects, where the effect is empty or not in the AGSL language.

### Related elements
- If a Transition has effect, it shall be an OpaqueBehavior with language “AGSL” and with a nonempty body.

### 2.4 Verification of OCL Constraints

<table>
<thead>
<tr>
<th>Goal</th>
<th>Validity of guards in state machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_OCL1</td>
</tr>
<tr>
<td>Type</td>
<td>Consistency</td>
</tr>
</tbody>
</table>

### Description / Rationale
- In the application models, guards of transitions are formulated as OCL constraints. The Papyrus UML tool used for UML modelling, has some support for validating OCL constraints, but only those which can be evaluated in the context of the associated element. Guards shall be evaluated in the context of the classifier of the state machine, thus there is no tool support for it in Papyrus.
- However, it shall be validated that the OCL constraint is valid (e.g. syntax is correct and does not reference not existing model elements, for instance properties or states).

### Output
- List of invalid OCL constraint guards with the owning transition and with a description of the error.
Related elements

Condition:
- Transition has a Constraint as guard (guard property)
- the Constraint is expressed as an OpaqueExpression (specification property)
- language of the OpaqueExpression is OCL

Consequences:
- the body of the OpaqueExpression is valid in the context of the Class that is the context of theStateMachine which contains the Region (either directly or indirectly) that contains the Transition (transition property). The Region containing the Transition can be contained
  - directly by theStateMachine (region property), or
  - indirectly by a State (region property) that is contained by a Region (subvertex property) which is contained by theStateMachine either directly or indirectly.

Metamodel pattern
### 2.5 Application Specific Verification

<table>
<thead>
<tr>
<th>Goal</th>
<th>Existence of a marked up singleton class representing the system itself</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_APP1</td>
</tr>
<tr>
<td>Type</td>
<td>Completeness</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>The main class of the system that should be tested shall be marked with the stereotype &lt;&lt;system_under_test&gt;&gt; (see D3.2 for details)</td>
</tr>
<tr>
<td>Output</td>
<td>True if exists, false if not</td>
</tr>
<tr>
<td>Related elements</td>
<td>Class with a Stereotype</td>
</tr>
<tr>
<td>Metamodel pattern</td>
<td>–</td>
</tr>
</tbody>
</table>

### 2.6 TCG Specific Verification

<table>
<thead>
<tr>
<th>Goal</th>
<th>Check of elements in the UML model to be in the subset which is the basis of the TCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_TS1</td>
</tr>
<tr>
<td>Type</td>
<td>Restriction</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>Test Case Generation to be developed in MOGENTES will be based on a well-defined subset of the UML models. It should be checked if the model to be used in TCG contains other elements or not. If there is additional information needed for documentation or clarity, but not intended for TCG, then a sufficient markup or tagging of these parts is needed.</td>
</tr>
<tr>
<td>Output</td>
<td>List of elements that are neither included in the 'TCG subset' of UML nor marked as 'not intended to use in TCG' (&lt;&lt;ignore&gt;&gt;).</td>
</tr>
<tr>
<td>Related elements</td>
<td>Details can be found in deliverable D3.2 Modelling Languages.</td>
</tr>
<tr>
<td>Metamodel pattern</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal</th>
<th>Restriction of OCL constructs in guards to the subset supported by the transformation into the TCG input.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>C_TS2</td>
</tr>
<tr>
<td>Type</td>
<td>Restriction</td>
</tr>
<tr>
<td>Description / Rationale</td>
<td>Test Case Generation to be developed in MOGENTES will support only a subset of the OCL language in the guards of the transitions. It shall be checked if other elements that will not be included in TCG are used or not.</td>
</tr>
<tr>
<td>Output</td>
<td>List of OCL constraints that includes not supported OCL constructs.</td>
</tr>
<tr>
<td>Related elements</td>
<td>Details can be found in deliverable D3.2 Modelling Languages.</td>
</tr>
<tr>
<td>Metamodel pattern</td>
<td>–</td>
</tr>
</tbody>
</table>
3 Ontology Based Verification

3.1 Introduction

3.1.1 Ontologies
Ontologies are widely used as knowledge management mechanism to capture knowledge about some domains. An ontology is a data model that represents a domain and is used to reason about the objects in that domain and the relations between them.

Ontologies generally describe the following items:

- Individuals / Objects / Instances
  - Basic or "ground level" elements of the domain.
- Concepts / Classes / Types
  - Sets or collections of objects sharing certain characteristics.
- Relations / Properties / Roles
  - Sets of pairs (tuples) of objects.
  - Determine the ways in which objects can be associated to each other.
- Attributes
  - Special relations where the concept is related to a concrete domain (e.g. integer, real, string).
  - Properties, features, characteristics, or parameters that objects can have.

3.1.2 Description Logic
In order to automate the analysis of an ontology, reasoners are used that are based on the description logic formulation of an ontology [Moeller03]. Description logic (DL) languages are a family of logic based knowledge representation formalisms that have formal semantics. It provides a way to represent knowledge bases in a well-structured and expressive way. DL reasoners provide some kind of inference services thus an important aspect of DL languages is decidability.

Description logics consist of terminologies (TBoxes) and model instances (ABoxes). A terminology is a metamodel-like 'dictionary' to describe a model while model instances store the knowledge of an object.

One of the main services offered by a reasoner is the so-called subsumption test, checking whether a concept is a 'subconcept' of another concept. By performing such tests it is possible to compute the inferred ontology class hierarchy, i.e., a class hierarchy where the subsumptions between classes are explicitly represented. In addition, a standard consistency check can be carried out using a reasoner: a class is inconsistent if it cannot have any instances according to the class descriptions in the description logic.

DLs can be considered as the decidable fragments of first order logic languages:

- objects are equivalent to constants in FOL,
- concepts are equivalent to unary predicates in FOL,
- relations are equivalent to binary predicates in FOL.

3.1.3 DL Families

Description logic languages are classified according to their expressivity, i.e., according to the elements contained by the language.

The smallest propositionally closed DL is ALC. It contains concepts constructed using boolean constructors (u, t, ;) and restricted quantifiers (9, 8)\(^1\). It can contain only atomic roles.

The following letters are used in the name of DL languages as a reference to a certain property:

\(^1\) The exact meaning of the constructors will be discussed in details in the next section.
• S is often used for $\text{ALC}_{R\!,\!,\!}$, i.e. for $\text{ALC}$ extended with transitive roles (e.g., hasPart can be defined as a transitive role)
• H denotes that role hierarchies can be defined (e.g., hasDaughter v hasChild)
• O denotes that nominals/singleton classes are allowed in the language (e.g., {Hungary})
• I denotes that the inverse of roles can be defined
• N denotes that number restrictions for roles can be used (e.g. a car has at least four wheels)
• Q stands for qualified number restrictions (e.g. a car has exactly four wheels that are not spare wheels)

### 3.1.4 Tools
A very commonly used ontology development tool is **Protege** [Protege] that is developed at the Stanford University. It facilitates the use of several reasoners to check ontologies.

One of the most powerful reasoners is the **RacerPro reasoner** [Racer] [Haarslev01], [Haarslev04]. RacerPro implements a highly optimized tableau calculus for very expressive description logic and offers reasoning services for multiple terminologies and model instances. The system implements the description logic SHIQ (see [Horrocks00]). RacerPro also provides facilities for algebraic reasoning including concrete domains and can even handle multiple definitions or cyclic definitions of concepts. It also offers a functional API for querying a knowledge base.

RacerPro supports the XML based formalism of the OWL DL language developed by the Ontology Working Group of the World Wide Web Consortium [OWL] but also supports the concrete syntax of the older Knowledge Representation System Specification (KRSS) and implements most of the functions specified in it. Since KRSS is the native RacerPro syntax and it is more compact and better for human reading this syntax will be used in this document.

### 3.2 Description Logic Formalism
The basic elements and constructors of the SHIQ logic implemented by RacerPro are the following (these are summarized in Table 3-1 showing both the description logic notation, the corresponding RacerPro syntax and the First Order Logic interpretation) [RacerUG]:

• **atomic** concepts: a set, e.g., a class as the set of its instances in the metamodel
• **top concept**: the set of all objects in the logic
• **bottom**: empty set
• **roles**: relations that relate concepts to each other
• **negation of a concept**: the complementary set of the concept
• **intersection of concepts**: the set of concepts that consists of the joint elements of two or more concepts
• **disjunction of concepts**: the disjunction of two or more concepts (two or more sets of objects) is a concept that is a set containing all and only the objects of the two or more concepts
• **existential quantifier wrt. a role and a concept**: a constraint on the set of objects that are related to an object of the given concept through a given role: this set cannot be empty
• **value restriction of roles**: this constructor yields a set of objects fulfilling some constraint on the range of the relation
• **qualified number restrictions**: the cardinality of the set of objects that are related to another concept by the given role is restricted (e.g., lower or upper bounds)
• **role hierarchies**: roles may have several subroles, i.e., the concepts that are related by a subrole are also connected by the superrole. The domain and the range of a role are the set of all the instances by default however they can be restricted to certain concepts.
• **inverse roles**: for instance the inverse role of the role ‘parent-of’ is the role ‘child-of’
• **transitive roles**: if concepts C1, C2, and C3 are related by a transitive role then C1 and C3 are also related by the role.
In ontologies a concept is a set of individuals that have the same characteristics. Thus a complex concept can be defined using the set constructions and restriction in the table: the negation of a concept is also a concept (i.e. the set of individuals that are not in the concept set), the intersection and the union of concepts define also a complex concepts, where an individual is an instance of an intersection if the individual has all the characteristics defined by the concepts in the intersections while an instance of a union of concepts has to have the characteristics of at least one concept in the union. In addition, restriction construction defines a set of individuals, e.g. \( \geq n \ R \ . \ C \) defines a set of individuals that are connected to at least \( n \) instances of concept \( C \) by role \( R \). The rest of the restriction constructions are straightforward.

RacerPro supports several kinds of concept axioms. Axioms are used to define the relationship between concepts (the DL notation and the RacerPro syntax is given in the following table):

- **general concept inclusions** state the subsumption relation between two concept terms, i.e. all instances of the subsumed concept are instances of the other concept (the subsumee)
- **concept equations** define the equivalence between two concept terms, i.e. both concepts have the same set of instances
- **concept disjointness axioms** declare pairwise disjointness of several concepts, i.e. disjoint concepts do not have common instances

<table>
<thead>
<tr>
<th>Concept Type</th>
<th>DL notation</th>
<th>RacerPro syntax</th>
<th>FOL syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic concept</td>
<td>A</td>
<td>:atomic-concepts (A)</td>
<td>A(x)</td>
</tr>
<tr>
<td>top concept</td>
<td>T</td>
<td>&quot;top&quot;</td>
<td></td>
</tr>
<tr>
<td>bottom concept</td>
<td>( \bot )</td>
<td>&quot;bottom&quot;</td>
<td></td>
</tr>
<tr>
<td>negation</td>
<td>( \neg C )</td>
<td>(not C)</td>
<td>( \neg C(x) )</td>
</tr>
<tr>
<td>conjunction</td>
<td>( C_1 \cup \ldots \cup C_n )</td>
<td>(and ( C_1 \ldots C_n ))</td>
<td>( C_1(x) \wedge \ldots \wedge C_n(x) )</td>
</tr>
<tr>
<td>disjunction</td>
<td>( C_1 \vee \ldots \vee C_n )</td>
<td>(or ( C_1 \ldots C_n ))</td>
<td>( C_1(x) \vee \ldots \vee C_n(x) )</td>
</tr>
<tr>
<td>exists restriction</td>
<td>( \exists R.C )</td>
<td>(some ( R \ C ))</td>
<td>( \exists y. R(x,y) \wedge C(y) )</td>
</tr>
<tr>
<td>value restriction</td>
<td>( \forall R.C )</td>
<td>(all ( R \ C ))</td>
<td>( \forall y. R(x,y) \rightarrow C(y) )</td>
</tr>
<tr>
<td>at-least restriction</td>
<td>( \geq n R )</td>
<td>(at-least ( n ) ( R ))</td>
<td>( \exists^n y. R(x,y) )</td>
</tr>
<tr>
<td>at-most restriction</td>
<td>( \leq n R )</td>
<td>(at-most ( n ) ( R ))</td>
<td>( \exists^n y. R(x,y) )</td>
</tr>
<tr>
<td>exactly restriction</td>
<td>( = n R )</td>
<td>(exactly ( n ) ( R ))</td>
<td>( \exists^n y. R(x,y) )</td>
</tr>
<tr>
<td>qualified at-least</td>
<td>( \geq n R.C )</td>
<td>(at-least ( n ) ( R ) ( C ))</td>
<td>( \exists^n y. R(x,y) \wedge C(y) )</td>
</tr>
<tr>
<td>qualified at-most</td>
<td>( \leq n R.C )</td>
<td>(at-most ( n ) ( R ) ( C ))</td>
<td>( \exists^n y. R(x,y) \wedge C(y) )</td>
</tr>
<tr>
<td>role name</td>
<td>( R )</td>
<td>:roles(R)</td>
<td>( R(x,y) )</td>
</tr>
<tr>
<td>inverse role</td>
<td>( R^{-1} )</td>
<td>(inv ( R ))</td>
<td></td>
</tr>
<tr>
<td>transitive role</td>
<td>( R^+ )</td>
<td>(( R ) :transitive)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3-1: The DL notation and the RacerPro (KRSS) syntax of the SHIQ logic

Version: 1.4
Status: released
MOGENTES_3-14-1.4r_D3.3b_OntoModelVerif-final
Table 3-2: The DL notation and the RacerPro syntax of the concept axioms

Since we can define in RacerPro special roles like attributes with a concrete domain (integer, real, cardinal, complex and string) the following constraints on concepts can be defined (we give only the RacerPro syntax in the form of a grammar where CN is a concept name, AN is an attribute name, and CDC stands for concrete domain concept):

\[
\begin{align*}
CDC & \rightarrow (min \ AN \ integer) \\
& \rightarrow (max \ AN \ integer) \\
& \rightarrow (equal \ AN \ integer) \\
& \rightarrow (equal \ AN \ AN) \\
& \rightarrow (divisible \ AN \ cardinal) \\
& \rightarrow (not-divisible \ AN \ cardinal) \\
& \rightarrow (> aexpr \ aexpr) \\
& \rightarrow (\geq aexpr \ aexpr) \\
& \rightarrow (< aexpr \ aexpr) \\
& \rightarrow (\leq aexpr \ aexpr) \\
& \rightarrow (= aexpr \ aexpr) \\
& \rightarrow (string= \ AN \ string) \\
& \rightarrow (string<> \ AN \ string) \\
& \rightarrow (string= \ AN \ AN) \\
& \rightarrow (string<> \ AN \ AN) \\
\end{align*}
\]

\[
\begin{align*}
string & \rightarrow "letter*" \\
aexpr & \rightarrow AN \\
\rightarrow real \\
& \rightarrow (+ aexpr1 \ aexpr1*) \\
aexpr1
\end{align*}
\]

In addition, RacerPro is able to retrieve an instance for a given definition that satisfies the conditions of the definition over a given ABox (a set of individuals, where individuals are instances of the concepts in the terminology TBox). Thus if we were able to define a consistent TBox and a corresponding ABox according to the requirement specification, constraint satisfaction problems can also be solved by RacerPro retrieval.

Please note that in order to perform consistency checks on metamodels and the models, the constraints for the metamodels and models have to be transformed into the above formalism.

### 3.3 Reasoning Tasks

Based on the description of classes (set of concepts) as an extended SHIQ logic, the RacerPro reasoner serves to test the satisfiability, subsumption, consistency and the validity of a metamodel.

Moreover, it enables the correctness check of a given model (a set of instances) according to its metamodel, i.e., whether the model conforms to the metamodel. This kind of validation is essential if a model of a metamodel was developed using tools that do not automatically maintain the consistency between the metamodel and the model (e.g., the tool allows the creation of inconsistent instances and links between them that do not have an appropriate representative in the metamodel), or we want to check the correctness of the output of the tool. This check is especially useful if some additional restrictions are given, for instance textually, that cannot be taken into account by the modelling tool (e.g. as listed in Sec. 2). However, the satisfaction of most of such criteria can be tested using ontologies where these requirements are expressed using description logic statements.
### 3.3.1 Inference Services Offered by RacerPro

Using RacerPro the following type of inference tasks can be performed:

- **On the TBox level (i.e., considering the metamodel ontology):**
  - **TBox classification**
    Classification is the process of computing the most-specific subsumption relationships between all concept names mentioned in a TBox. The result is often referred to as the taxonomy of the TBox, or the inferred hierarchy of the concepts and gives for each concept name two sets of concept names listing its “parents” (direct subsumers) and “children” (direct subsumees).
  - **TBox coherence check**
    Checking the consistency of all concept names mentioned in a TBox (without computing the parents and children) in order to determine all concept names which are unsatisfiable, i.e. that cannot have a valid instance model.

- **On the level of knowledge base (TBox + ABox, i.e., considering the model ontology as an instance of the metamodel ontology):**
  - **Conformance of the model to the metamodel** (e.g., if it references a concept that is not part of the metamodel)
  - **Consistency of the instance model with respect to the metamodel ontology** (e.g., if a constraint defined in the metamodel is violated in the model)
  - **Retrieval inference:** a way to find all individuals mentioned in an ABox that are instances of a certain concept

- **On the ABox level (i.e., considering the model ontology in itself):**
  - **Coherence of the instance model** (e.g., if it contains contradictions in the concrete domain constraints)

### 3.3.2 Concept Level Consistency Checks in RacerPro

The following tests can be performed in RacerPro in order to find inconsistencies in a metamodel, i.e., in a TBox:

- **Inconsistent classes:** The (tbox-coherent-p) query returns TRUE if there is at least one inconsistent class in the ontology, while (check-tbox-coherence) delivers the atomic concepts (classes) that are inconsistent.
- **Extendable classes:** Using the (concept-equivalent-p C1 C2) query it can be tested whether a concept is equivalent to a complex concept that can be defined for instance as a disjoint union of classes. This way classes that cannot be extended with additional subclasses can be determined.
- **Cycle detection:** (tbox-cyclic-p) query checks whether there is a loop in the concept hierarchy.
- **Subsumption test:** (concept-subsumes-p Concept1 Concept2) returns if Concept1 subsumes Concept2, i.e., if Concept2 is always true when Concept1 is true (e.g., SafeSystem subsumes DeadlockFreeSystem).
- **Inferred subsumption test:** (concept-descendants Concept_name) lists all classes in the hierarchy below the given concept.
- **Inferred superclass test:** (concept-ancestors Concept_name) is the inverse query of concept-descendants.
- **Subclass test:** (concept-children Concept_name) delivers all the subclasses (i.e., the direct descendants) of the given concept.
- **Superclass test:** (concept-parents Concept_name) is the inverse query of concept-children.
- **Role subsumption test:** (role-subsumes-p Role1 Role2) returns true if Role1 role is refined to Role2.
- **Role domain and range test:** The domain and the range of roles are queried using the (role-domain Role_name), and (role-range Role_name) queries.
3.4 The Role of Ontologies in Model Checking

In MOGENTES ontologies are used to validate application models that are developed in the demonstrators. As an ontology can handle only a single metamode -l model relation (i.e., multiple level metamodelling is not supported), ontology-based verification can be applied on multiple levels in order to verify the properties listed in Sec. 2. It can be applied in the following cases:

- In order to formulate the constraints that are referencing elements of the UML metamodel, a part of the UML metamodel containing the referenced elements (e.g., class diagram and state machine related elements) has to be transformed into the DL notation to compose a TBox. In this case the application model is transformed into an ABox, i.e., elements of the application model are considered as instances of the concepts defined in the UML metamodel.
  
  In this case the knowledge base level inference services can be used for verification purposes (see Sec. 3.3.1). This way the application model as a UML model is verified if it is modelled correctly or not, considering the formalism and constraints of the UML metamodel and some additional user defined constraints.

- During modelling other concepts than those defined in the UML metamodel can also be used especially if we have an own domain model. These can be formulated as UML profiles (see e.g. the MARTE profile). In this case the profile (i.e. the domain metamodel) can be transformed into a TBox and the application model shall be transformed again to an ABox.
  
  This way the application model as an instance of a domain model is verified if it is modelled correctly or not, considering the formalism and constraints defined in the domain model.

- Another application way is when the application model is mapped to a terminology box: in this case the TBox level inference services can be used (see Sec. 3.3.1 and 3.3.2).
  
  This way the application model as a domain model is verified in itself if it is consistent or not.

- When considering an application model as a domain model modelled on a UML class diagram, an instance of it can be modelled on a UML object diagram. In this case an instance model with respect to the application model as a domain model can be verified.

In all cases we consider the application models as UML models. The UML metamodel itself as well as the profile models are also defined as UML models, thus a transformation from UML models to description logics is needed.

Finally in MOGENTES no own modelling language is developed, but UML is used as one of the front-ends for test case generation, and constraints listed in Sec. 2 are about how UML models shall be constructed to be the well-defined basis for transformation and test case generation. Thus, in MOGENTES the first approach is applied for the static verification of demonstrator models.

3.5 Mapping UML to DL

The relation between UML and ontologies is discussed in several papers, e.g. [Straeten03], [Gasevic04], [Berardi01]. These papers can be categorized into two classes. One direction (e.g., [Gasevic04], [Djuric04], [ODM]) is to investigate how UML can be used in the ontological engineering as a front-end modeling tool (borrowed from the software engineering field). The other approach (e.g., [Berardi01], [Berardi02], [Brockmans06]) is the translation of UML diagrams into ontologies (or semantic web languages like RDF, OWL, etc.) in order to give a more precise semantic for the modeled system or to analyze model correctness and completeness. Our solution is related to this later approach.

3.5.1 Metamodel (TBox) Level

The ontology based representation of the metamodels (either if it is the UML metamodel, or a profile model or the application model as a domain model) corresponds to the following scheme:

- A class in the metamodel is represented by a concept in the ontology.
- Generalization is naturally provided in the ontology as subsumption.
- **Aggregation, composition and association** relationships between two classes are considered as roles between concepts.
- **Attributes** of a class are represented as data properties of concepts (or in other words binary relations where the range of the relation refers to a concrete domain concept, e.g. integer, real, string) assigning a data value to a class in the ontology.
- **Association cardinalities** are modelled as number restrictions of the corresponding relations in the ontology.
- **Additional constraints**: only those constraints can be included into the ontology that are concrete enough to express these in the DL form (e.g. numerical constraints).

The additional features of DL languages facilitate the more precise modelling of the metamodels:
- It can be checked whether the subclasses of a given class are disjoint, i.e., there is no (sub)class which is inherited from two disjoint subclasses of the given class or there are no individuals that are instances of both concepts.
- Defining a class as the union of its subclasses refers to completeness. In other words, if it is stated that a class is the union of its subclasses there cannot be any additional object that is an instance of the superclass but it is not an instance of one of its subclasses. This way the direct instantiation of abstract classes can be avoided.
- **Necessary and sufficient conditions** can be defined for classes: necessary conditions impose what conditions have to be satisfied by all subclasses (subclass instances) of the class while sufficient conditions determine the subclasses (subclass instances), i.e., a class that satisfies the sufficient conditions must be a subclass of the given class. Conditions are logic expressions on properties.

### 3.5.2 Model (ABox) Level

The mapping of instance models to ABoxes depends on how it is represented:
- If the instance model is a UML model and its relation to the UML metamodel is considered, then
  - all elements are mapped to individuals that are instances of their UML types (e.g. Class, Association, State, Transition, TimeEvent),
  - the links between them are defined as role fillers of the roles defined in the UML metamodel ontology.
- If the relation between a domain instance model and the domain (meta)model represented as a UML profile is examined, then
  - again all elements are mapped to individuals, but in this case these are instances of concepts defined as their stereotype,
  - the links between them are defined as role fillers of the roles defined in the profile ontology.
- If the instance model is defined on a UML object diagram, then the
  - UML objects are mapped to individuals, that are instances of their types, which are defined as UML classes,
  - values of attributes are mapped to concrete domain objects, that are related to the individuals as attribute-fillers,
  - the links between them are defined as role fillers of the roles that correspond to the associations between UML classes.

### 3.5.3 Constraint Representation in RacerPro Syntax

In order to capture the additional constraints for the metamodels and models into the ontology, they have to be defined in a standard way. One way is to use directly the DL notation. However, the drawback of this notation is that in this case the designer should know the structure of the ontology. The other way is to find an appropriate constraint representation that is convenient for UML models. This representation could be a part of the **Object Constraint Language** (OCL) [OCL]. Since OCL has the expressiveness of first-order logic, while the SHIQ logic is a lower-order logic, only a subset of OCL expressions can be used for constraint definition. The expressions that can be translated into the RacerPro syntax are the following (we assume that the reader is familiar with OCL):
- the context of an expression can be a class (a concept)
- the comparison of two objects
- numerical or string restrictions for attributes of a class
- subsumption between classes
- instance retrieval according to numerical or string constraints
- set and bag functions can be translated as instance retrieval or defining new complex concepts and testing their emptiness
- Boolean operators (and, or, not, implies) can be used also in the ontologies for concepts (classes)
- navigation is represented by the relations in the ontology.

### 3.5.4 Transformation in VIATRA2

To implement the mapping we use the VIATRA2 model transformation framework [VIATRA2]. This framework contains components for both model representation and model manipulation. The model representation is based on a very simple metamodel, the so called VPM metamodel, thus it enables the formal representation of a wide variety of domain metamodels. The models in VIATRA2 are represented as graphs and the model manipulation component is an effective graph transformation engine. (For more information on VIATRA2 see Deliverable D1.2 Part b).

To implement the transformation from UML to the DL formalism the following components are needed:

- source metamodel in the VIATRA2 modelspace,
- importer for the source models (this is a Java component),
- target metamodel in the VIATRA2 modelspace,
- transformation code in the VIATRA2 transformation language,
- exporter for target models (this is also written in the transformation language).

Currently the UML2 and an ontology metamodel are available as VPM models, the importer for UML2 models is also available, and the transformation from UML class models to DL formalism has a prototype implementation. Further work is needed on improving the UML to DL transformation including e.g. OCL constraints and profiles.

Since the models to be verified together with their metamodels are imported into the VIATRA2 modelspace, the fulfillment of some properties can also be checked in the VIATRA2 framework. However, this approach can only be used when simple graph pattern matching is sufficient (as the counterpart of instance retrieval in RacerPro) and inferences between concepts are not needed.
4 Implementation

This chapter describes how the ontology based verification tool was implemented, how it can be used, i.e., how it is integrated into the tool integration framework, and how the constraints described in Sec. 2 are verified.

4.1 Process of the Verification

The verification process consists of two main steps:

- First the UML model is transformed into the ontology domain; this involves the creation of an ontology in the OWL syntax [OWL] from the application model. For that purpose the VIATRA model transformation framework is used.
- Next the checking of the constraints is performed using the RacerPro reasoner. Constraints are formulated as queries on the ontology.

An overview of the process is depicted in Figure 4-1. The two main components are discussed in more details in the following subsections.

4.1.1 Transformation of UML Models

The process of creating a transformation in VIATRA contains the following steps:

1. The metamodel of the source modelling language is needed as a model in the VIATRA model space.
2. An importer for source models are needed that reads the models which are in some external format (for instance in XMI serialized format) and creates the model in the VIATRA model space as an instance of its VIATRA metamodel.
3. The metamodel of the target language is needed as a model in the VIATRA model space.
4. The mapping shall be defined from the elements of the source metamodel to the elements of the target metamodel, and this mapping shall be implemented as a transformation in VTCL (Viatra Textual Command Language), which creates an instance of the target metamodel in the VIATRA model space from an instance of the source metamodel.
5. An exporter shall be written for the target language that serializes the instance of the target metamodel created in the VIATRA model space into an external format (into a given concrete syntax of the language).
In case of the UML-to-OWL transformation the following components are required (Figure 4-2):

1. **The source metamodels: UML + OCL + AGSL**
   The UML 2.1 metamodel was already part of the VIATRA environment, but the OCL and the AGSL metamodels had to be created and linked with the UML metamodel.
   OCL and AGSL are constraint and action specification languages that can be used in the context of UML models and which are used in the project for specifying guards and actions in UML state machines.
   For the remainder of this section, when UML metamodel is mentioned, depending on the context it also includes OCL and AGSL related parts, as these are integral parts of it.

2. **The importer for the source metamodel**
   Correspondingly to the previous, an importer for UML 2.1 models was already available but OCL and AGSL importers had to be created to replace textual representation of OCL and AGSL expressions with their abstract syntax tree representation in the VIATRA model space.

3. **The target metamodel: OWL-DL**
   The OWL2 metamodel was created in the VIATRA model space as part of this implementation work.

4. **The UML-to-OWL transformation**
   As the constraints in Sec. 2 restricted the UML models in the context of the UML metamodel (i.e.), the transformation consisted of two steps:
   a. The UML metamodel had to be transformed into an ontology TBox.
      Although this step was executed only once during the development phase (as the UML metamodel remains the same) a transformation has been developed because (i) the UML metamodel is extensive and manual transformation would be time consuming and error prone, and (ii) this transformation is independent of the UML metamodel (as the subject of the transformation) thus can be reused for an arbitrary language.
   b. The UML instance models had to be transformed into ontology ABoxes.
      This step shall be executed for each application model to be verified. It transforms models as instances of the UML metamodel and creates ABox elements that are instances of TBox concepts representing UML metamodel elements.

5. **The OWL exporter**
   This component was developed as a VIATRA "transformation" (using only pattern matching capabilities) that traverses the OWL model (TBox+ABox) and creates the serialized XML representation of the OWL model.

---

*Figure 4-2: The model transformation process*
4.1.2 The Checker Component

The checker component is implemented as an Eclipse plug-in in Java language and it provides an API for checking models. First, it establishes a connection with a local or remote RacerPro server using its JRacer component that provides a Java API for communication with RacerPro server. Then the ontology can be uploaded to the server and consistency and the constraints can be checked on it. It provides two mechanisms for constraint checking:

A. Constraints can be defined in an xml configuration file if the constraint is simple and it can be checked with a simple query on the ontology.

B. Constraints can be defined as custom code if multiple queries and further specific processing are needed.

The definition of constraints in configuration file

In the configuration file a list of queries can be defined that shall be checked during model verification (see an example in Figure 4-3). For a query the following information shall be defined:

- **queryID** – The ID of the constraint
- **version** – The version of the implementation of the constraint checking
- **description** – A short description (name or summary) of the constraint
- **longDescription** – A detailed description of the constraint
- **queryHead** – The head of the query in the nRQL syntax. nRQL is the query language of RacerPro.
- **queryBody** – The body of the query in the nRQL syntax. Together with the queryHead this is forwarded to the RacerPro server.
- **queryType** – Describes the content of the result list. This gives information for how to process and interpret the elements of the list returned by RacerPro that contains information about the model elements violating the constraint.
<queries xmlns="http://ontology.inf.mit.bme.hu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ontology.inf.mit.bme.hu
http://toolintegration.inf.bme.hu/ontology/queries.xsd">

<query>
  <queryID>C_C5</queryID>
  <version>1.0</version>
  <description>Checks that every class has at least one StateMachine.</description>
  <longDescription>
    Checks that every active class has at least one StateMachine. A class is a Class as in the UML metamodel (i.e. a StateMachine is a Class too). The result list contains the UML Class elements that has no associated StateMachine.
  </longDescription>
  <queryHead>
    (?x)
  </queryHead>
  <queryBody>
    (and (?x #!:Class)
      (?x (boolean= #!:Class_isActive #T))
      (neg (project-to (?x) (and (?y #!:StateMachine)
          (?x ?y #!:BehavioredClassifier_ownedBehavior)))))
  </queryBody>
  <queryType>
    <patternMatching>
      <elementType>Class</elementType>
    </patternMatching>
  </queryType>
</query>

...</queries>

Figure 4-3: Example configuration file for constraint checking
The output of constraint checking

The checker component provides its result as an XML structure containing the results of the checked constraints (Figure 4-4). The result of a single constraint check contains the following items:

- **queryID** – The ID of the constraint
- **version** – The version of the constraint implementation that is used for the verification
- **description** – A short description (name or summary) of the constraint that is checked
- **longDescription** – A detailed description of the constraint, which is checked
- **resultList** – List of items that violate the constraint.
  - An item can contain multiple related elements (this is related to the `queryHead` in the constraint definition).
    - An element contains
      - type of the element, and
      - an instance of the model that violate the constraint.
- **conclusion** – A textual description of the result of the constraint checking.
- **boolConclusion** – A compact summary of the result. Its value can be OK or NOK.

```xml
<check>
  <result>
    <queryID>C_C5</queryID>
    <version>1.0</version>
    <description>Checks that every class has at least one StateM..</description>
    <longDescription>Checks for stateMachine existence of ...</longDescription>
    <resultList>
      <item>
        <element>
          <type>Class</type>
          <instance>
            uml2_models_AlarmSystem_6.uml_AlarmSystem_OpticalAlarm
          </instance>
        </element>
      </item>
      <item>
        <element>
          <type>Class</type>
          <instance>
            uml2_models_AlarmSystem_6.uml_AlarmSystem_AlarmArmed
          </instance>
        </element>
      </item>
      <item>
        <element>
          <type>Class</type>
          <instance>
            uml2_models_AlarmSystem_6.uml_AlarmSystem_AcousticAlarm
          </instance>
        </element>
      </item>
    </resultList>
    <conclusion>There are some faulty elements. Check the resultList.</conclusion>
    <boolConclusion>NOK</boolConclusion>
  </result>
  ...
</check>

Figure 4-4: Example output of constraint checking
4.2 Execution of the Verification Process

Both the transformation and the constraint checker components are available in the framework thus the execution can be managed from the framework.

As the UML-to-OWL transformation is implemented as a VIATRA transformation its integration into the framework did not require further work, but the steps of the execution needed to be composed into a transformation process. The steps of the transformation are depicted in Figure 4-5, as captured from the Process Execution User Interface component of the framework.

A connector for the checker component had to be implemented, which exposes the functionality of the component to the framework. A process is also created for the execution of the verification; see Figure 4-6.

In both processes it is also included that the input (the UML model or the OWL file) is downloaded from the Data Repository and the result (the OWL file or the result of the check) is uploaded there.

Figure 4-5: The transformation process realized in the framework
4.3 Verification of the Constraints

In this section details are given concerning the checking of the constraints described in Sec. 2.

Metamodel consistency (C_C1). This is about checking the consistency of the modelling language. As finally we have used UML extended with OCL and AGSL the scope of this check is limited to the checking of the correctness of the transformation of the metamodels (as we expect that the metamodels in themselves were correct), i.e., the faults introduced by the transformation can be discovered with this check.

To perform the check the "(tbox-coherent?)" RacerPro function shall be used.

Model consistency (C_C2, C_C3, C_C4). These are partially checked during the transformation (if the model is defined in terms of the metamodel), or with the "(abox-consistent?)" RacerPro function.

Structural checks (C_C7). This can be easily constructed in nRQL (the New RacerPro Query Language), because it can be described with the terms of the metamodel. For example the query that retrieves the faulty signals not covered by at least one transition trigger (C_C7) can be written as follows:

```
{retrieve
 (?xSignal)
(and (?xSignal #:!:Signal)
  (neg (project-to (?xSignal)
    (and (?xTransition #:!:Transition)
      (?xTrigger #:!:Trigger)
      (?xSignalEvent #:!:SignalEvent)
      (?xTransition ?xTrigger #:!:Transition_trigger)
      (?xTrigger ?xSignalEvent #:!:Trigger_event)
      (?xSignalEvent ?xSignal #:!:SignalEvent_signal) )))}
```

The literals starting with "?” are the variables, literals starting with “#!:” are the metamodel elements. The head of the retrieve "(?xSignal)" describes that it will return the faulty signals. The body describes the pattern that binds the ?xSignal variable. The inner and in the code above defines the pattern matches when
a transition has a trigger, which has an event, which has a signal. These correct signal elements are projected with the project-to operator, and the complement is created with the neg operator. Finally, faulty signals that are not matched to the inner pattern are returned.

**Checks with datatype properties (C_C6, C_C8, C_TS2, C_BD1, C_BD2).** In case of C_C6 (compared to C_C7) it is an exception if the signal is received by a class with stereotype `<environment>`, i.e. this condition joins to the previous with or relation. The previous query is extended with the following pattern:

```prolog
(neg (project-to (?xSignal))
  (and (?xClass #!:Class)
    (?xStereotype #!:Stereotype)
    (?xClass ?xStereotype #!:Element_appliedStereotype)
    (?xStereotype (string= #!:NamedElement_name "environment"))
    (?xReception #!:Reception)
    (?xClass ?xReception #!:Class_ownedReception)
    (?xReception ?xSignal #!:Reception_signal)))
```

The stereotype's name is stored as an OWL DatatypeValue, which can be checked with the help of the `string=` concrete domain concept. C_C8, C_TS2, C_BD1 and C_BD2 can be checked similarly including OCL and AGSL expressions.

**Check using constraint query atom (C_SM2).** In C_SM2 constraint the so-called query atom is used to compare the language and body of the OCL expressions.

```prolog
(?xOpaqueExpression1 ?xOpaqueExpression2
 (constraint (#!:OpaqueExpression_body) (#!:OpaqueExpression_body) string=))
```

The query atom describes, that the two bounded opaque expressions have a string typed datatype relation named “body” and these are constrained to be equal.

**Checks using lambda expressions for greater expressivity (C_C5, C_TS1).** In the case of C_C5 the query body matches active classes, and the ?x variable bounded to the matched classes is given as a parameter to the so-called lambda expression in the query head. Lambda expressions are nameless function definitions, in which MiniLisp functions can also be called. With the help of this extension, aggregate functions can be used, complex queries can be written, or even we can format the output. In the case of this query each class's state machine is retrieved connected through the ownedBehavior relation. The retrieved list's elements are counted with `length`, then stored to the variable numRelations with the `let` expression. Next it is checked: if less or more than one relation exists the result is constructed, else rejected. The whole query is as follows:

```prolog
(retrieve
  
  (lambda (xClass) (let ((numRelations (length (retrieve `(?y) `,(xClass ?y #!:BehavioredClassifier_ownedBehavior))))
    (if (or (> 1 numRelations) (> numRelations 1))
      `(?x ,xClass)
       :reject
    ))) ?x)
  
(and (?x #!:Class)
  (let (x (boolean= #!:Class_isActive #T))
    )))
```

C_TS1 can also be described with lambda expressions, utilizing list functions: if every direct-type is the member of the supported subset of the metamodel, it is correct, else returned.
Recursive check using transitive role (C_SM1). nRQL with the MiniLisp extension is a termination safe language, which means that recursion cannot be written. Recall the metamodel structure of C_SM1. In that case a recursive query should be written, because a state can contain a region, and a region can contain a state. This query can also be written in Java language, but it is simpler to write in nRQL using transitive roles. We can introduce a new transitive role called subState, which is a subrole of Region.subVertex and State.region. It greatly simplifies the query by reducing the checks to whether the state has an incoming transition, or the state has a subState which has an incoming transition.

Customizing correctness evaluation (C_APP1). The C_APP1 check is simple, but the correctness evaluation is different from the others. Most queries are evaluated to NOK, when the retrieve returned at least one (faulty) instance. In this case exactly one class must be stereotyped «system_under_test», so when one element is returned, it means that the model is correct, in other cases it isn’t. For this purpose a custom analyze script can be written in ECMAScript:

```ecmascript
<analyzeScript>
correctResultHolder.set(false);
if(queryResult_.size() == 1) {
  correctResultHolder.set(true);
  conclusionHolder.set("The only system_under_test stereotyped class exists.");
} else if(queryResult_.size() == 0) {
  conclusionHolder.set("The system_under_test stereotyped class is missing.");
} else if(queryResult_.size() &gt; 1) {
  conclusionHolder.set("There are " + queryResult_.size() + " SUT classes.");
}
</analyzeScript>
```

Validity of OCL and AGSL expressions in the UML model (C_OCL1). This constraint is checked before the transformation is executed as the expressions are parsed during the import: for this the existing parsers are used. A drawback of the parsers is that they do not resolve the identifiers by default for OCL guards in state machines and for AGSL expressions at all. However, this missing resolving functionality is implemented in the importers, thus identifiers in the expressions that cannot be interpreted in the context of the model are filtered during the import.
5 Metrics Based Testability Assessment

Checking of constraints in UML models ensures that only correct models are used as input for test case generation. However, the consistency and correctness of a model guarantee neither the applicability nor the effectiveness of a particular test generation method.

Since the very beginning of automated test generation the prediction of the success of a particular approach for a unit to be tested is a core problem. Initially, testability assessment and checking became an integral part only in the field of hardware ATPG, but during the last one-two decades testability assessment place an increasingly important role in software and system testing, as well.

The core idea of testability assessment is as follows:

- Testability metrics are estimated at a high-level model of the object to be tested in order to predict the testability of the implementation. Such a high-level model can be derived:
  - either by abstraction of the actual implementation (for instance a typical representative is the extraction of a control flow graph from the run-time code and the use of some graph complexity metrics),
  - or extracted from an intermediate high-level design model of a workflow based on gradual refinement (an example is using complexity metrics measured at the register-transfer level model of a hardware circuit).

In both cases the assumption on a strong correlation between the high-level abstract model used for testability metrics estimation and the target implementation is fundamental. Model-based testing in the sense as MOGENTES aims at it is still in the initial phase of its development as a mature paradigm. However; several facts argue for the validity of using testability metrics even in this context, as well:

- Typical code generators weave the target code in a template-based way by substituting smaller model fragments by means of code templates and integrating them into the gradually generated target code fabric. This way, the strong structural correspondence between the model and its target implementation is maintained.
- Similarly, debugging such code generators relies primarily on code inspection which makes the correlation in the form of bidirectional traceability to a primary design objective in the development of code generators (and all model transformers).
- The model to high-level language code generation problem heavily resembles to the high-level language to machine-level code compilation. For the later one, rich industrial experience proves the usefulness of testability metrics generated from the high-level code.

This way, the experimental techniques described in the current chapter are still research ideas as contributions of the evolving field of model-based testing. Naturally, the crystallization of a practically useful set of testability metrics requires a broad basis of experiences exceeding the frames of a research project.

Model based metrics provide support for the test engineer to make decisions in the TCG and testing process in the following way:

- On the basis of model based metrics a quick assessment of the applicability conditions of a given TCG track can be performed. Different TCG tracks may have different limitations, strengths and weaknesses depending on the characteristics of the specific model. In MOGENTES, the models that represent time related behaviour (thus include one or more time events) shall be handled by the UML/UPPAAL TCG track; however, if there are no such model elements then the UML/OOAS TCG track has better complexity handling capabilities and thus the application of UML/UPPAAL track is not efficient. As another example, models consisting of components that interact using complex data structures are amenable to qualitative fault modelling based error propagation analysis to optimize test generation.
- The high value of static structural complexity metrics of a component may indicate that the test generation will suffer from state space explosion and thus complexity handling measures shall be taken (e.g., model slicing or abstraction) before test generation. Since static checking can not precisely predict the dynamic state space of a component, these complexity indicators shall be manually checked.
- Metrics help to identify components that are more error-prone than other components. Recent studies found that there is a correlation between certain complexity metrics (like Number of Methods Local, Cyclomatic Complexity, and Coupling Between Objects) and the error-proneness of components [Siket]. This way metrics that are able to predict error-proneness can contribute to the
optimization of testing activities by identifying the critical components that shall be in the focus of testing.

It is important to emphasize that the static metrics are computed to support the decisions of the test engineers and not to take decisions in an automated way. The metrics (except from clear cases of applicability conditions) provide only prediction about the error-proneness of components or about the conditions for reaching/exceeding the complexity limits of specific TCG tracks, this way manual assessment and revision is required. Note that there is still a debate in the practice about determining how the metrics can be precisely used, although it is commonly agreed that even imperfect measuring is better than none ("you can’t control what you can’t measure") and measuring has positive impact.

In the following we outline some model metrics that can help in characterizing models and supporting decisions. We also demonstrate that the computation of metrics can be implemented using the already available ontology based machinery described in previous sections by determining the selected characteristics using ontology based queries.

In given application area the predictive power of certain metrics can be determined statistically by evaluating the capabilities on a wide range of models. In the scope of the project this was not possible, however, we demonstrate the model assessment approach by using simple existence relations, and provide an open ontology based evaluation environment that can be easily re-used and further extended with new assessment relations. In addition, we outline some more model metrics that can be the base for further assessments, and also provide the calculation of these metrics.

5.1 Workflow of Model Assessment

A generic workflow of how determination of model metrics can support the decisions of test engineers is depicted in Figure 5-1. We call cardinal metrics of an application model those metrics that has influence on the applicability of the available TCG tools on the given application. After determining these cardinal metrics, the TCG tool shall be selected which is suitable for the given model and seems to be the best within the suitable TCG tools according to some criteria (e.g. generation time, coverage of generated test cases, execution time of generated test cases).

When the TCG tool is selected, some complexity metrics can be computed on the model that predict those characteristics which can influence the efficiency of the test case generation. Based on the metric values, the quality of the model is evaluated. If the model shall be changed (e.g. by decomposition), then the application developer gets the hints about where are the critical points of the model. After modifications, the evaluation of these complexity metrics is performed again in a next iteration until it is considered that the model reaches a required quality.
5.2 The Computed Model Metrics

In software development it is a common practice to measure the characteristics of the source code using quantitative values, called software metrics (e.g. lines of source code, number of ancestors of an object). Several studies demonstrated explicit correlation between certain metrics and extra-functional attributes of software like the testability. According to a recent study [Siket], several coupling metrics (e.g., Coupling Between Objects, Response Set of a Class, Number of Outgoing Invocations, and Number of Foreign Methods Accessed) and several size metrics (Number of Methods Local, and Logical Lines of Code) were found to be effective for predicting the error-proneness of software components. Several model-based metrics (i.e., metrics that are defined in model terms) were also proposed in the literature: Error-proneness is predicted by using the cyclomatic complexity of UML state machines [Goseva-Popstojanova03].

Based on these results, we interpreted relevant software metrics on the model and calculated upper (worst case) estimates of the corresponding source code level metrics. Some new metrics were also defined, that can be used to help selecting test generation methods.

The current section defines the metrics that we applied to assess models, and describes their implementation.

5.2.1 Class diagram metrics

Model metrics in this category are based on existing source code metrics. Two size metrics (NML and WMC), and four coupling based metrics (CBO, NOI, NFMA and RFC) are defined. For all metrics, higher value indicates a more complex model.
5.2.1.1 Number of Methods Local (NML)

Number of Methods Local counts all the methods of a class defined locally, i.e., the methods of the ancestors are excluded.

5.2.1.2 Weighted Methods per Class (WMC)

This metric is similar to NML, but a weight is assigned to the methods and these weights are summarized. There are multiple methods for calculating the weight, one of the most popular is the McCabe’s cyclomatic complexity (McC) [McCabe]. This complexity value must be calculated for the body of a function.

5.2.1.3 Coupling Between Object Classes (CBO)

Coupling Between Object Classes measures connectivity between instantiated classes. An object class is connected to another (the so called couple class), if the other class’s method can be called or an attribute can be used by the object class.

For a class (excluding generalization) there are three ways to get couple:

- The class has a property, the type of which is the couple class.
- The class has an operation, the parameter type of which is the couple class.
- The class has an operation, the return value of which is the couple class.

The CBO for a class is the sum of the above defined couples, plus all of the ancestors’ couples. The couple cannot be the actual class, or its ancestor.

Note that visibility is not handled in the current implementation, all of the operations and properties are assumed to be public. This is usually acceptable for application models focusing on the definition of the interface of the class, and hide implementation details.

5.2.1.4 Number of Outgoing Invocations (NOI)

Number of Outgoing Invocations is the sum of methods that the actual class can call directly. The value is the sum of the following four cases:

- The number of methods of the couples.
- The number of methods of the couples’ ancestors.
- The number of methods of the actual class.
- The number of methods of the actual class’s ancestors.

5.2.1.5 Number of Foreign Methods Accessed (NFMA)

Number of Foreign Methods Accessed is the number of methods that can be called from the class, excluding the methods of the actual class, or its ancestors. This can be calculated like NOI, excluding the last two cases from the list.

5.2.1.6 Response Set For a Class (RFC)

Response Set For a Class is the set of methods, that can be called when a class operation is issued. This includes the methods counted by NOI, and also transitive function calls. This can be computed easily using ontologies, because a relation can be created using RacerPro to determine the couples and ancestors of the class. The following code defines the canAccessMethod relation, which connects two classes (inspectedClass and couple class) when inspectedClass can call couple’s method.

```prolog
(firerule
  (and (or (?class #!:Class) (?class #!:Interface))
    (or (?couple #!:Class) (?couple #!:Interface))
    (or (?inspectedClass #!:Class) (?inspectedClass #!:Interface))
    (or (same-as ?class ?inspectedClass)
      (same-as ?class ?inspectedClass #!:ancestor)
    )
    (or (project-to (?class ?couple))
      (and (?operation #!:Operation)
        (?class ?operation #!:Class_ownedOperation)
        (?parameter #!:Parameter
          (?operation ?parameter #!:BehavioralFeature_ownedParameter)
      )
    )
  )
)
(\(\text{parameter \ ?couple \#!:TypedElement\_type}\))
(project-to \(?\text{class \ ?couple}\)
(and \(?\text{operation \#!:Operation}\)
 (\(?\text{class \?operation \#!:Class\_ownedOperation}\)
 (\(?\text{operation \ ?couple \#!:Operation\_type}\))
(project-to \(?\text{class \ ?couple}\)
(and \(?\text{property \#!:Property}\)
 (\(?\text{class \?property \#!:StructuredClassifier\_ownedAttribute}\)
 (\(?\text{property \ ?couple \#!:TypedElement\_type}\))
(same-as \?\text{class \ ?couple}\)
 (\(?\text{class \ ?couple \#!:ancestor}\)
)
)
)((\?\text{related \?inspectedClass \ ?couple \#!:canAccessMethod})
)

Making this relation transitive, the reasoner automatically connects classes that can be called transitively. Finally, the number of operations is counted the same way, as it was counted in the NOI case.

### 5.2.2 State machine metrics

State machine level metrics provide further characterization of the execution complexity of models.

#### 5.2.2.1 Cyclomatic complexity

The following formula defines the cyclomatic complexity:

\[
\text{complexity} = \text{transitions} - \text{vertexes} + 2
\]

Vertexes in the state machine include states and pseudo-states. Transitions are the edges between vertexes.

#### 5.2.2.2 Maximum length of state disjoint paths

This metric first converts the hierarchical state machine into a flat state machine, preserving the original semantics. Lowest level vertices of the original state machine form the vertices in the flat one. In the flat state machine there is a directed edge between two states, if they can be reached within one step in the original state machine. Edges are weighted; they have weight 1, if the edge was a transition in the original state machine; otherwise the weight is 0. Zero weighted edges can occur between a state, and its initial pseudo state. Finally a depth first search (DFS) is performed to determine the longest path, according to the weights. This metric returns the maximum number of steps.

The flat state machine structure can be created with the reasoner, because structures (concepts, individuals and relations) are handled easily in the ontology domain. The DFS is implemented in an external Java function.

### 5.2.3 Computation of metrics for the selection of the test generation method

The metrics in this category are binary ones as they simply check the existence of specific model elements that determine the applicability of given TCG methods.

#### 5.2.3.1 Timing parameters

The computation of this metric checks for the existence of "time events" described in the UML standard. If time events are used in the model then the application of the UMOL/UPPAAL TCG track can be considered.

#### 5.2.3.2 Data flow with non-trivial data type

We call non-trivial data types those data types that allow a wide range of possible values for a property (e.g. integers). These types may heavily contribute to the explosion of the state space during test case generation, so in these cases qualitative abstraction should be applied which deals with equivalent partitions of the value range.
For simplicity we consider all data types non-trivial that are others than Boolean and enumeration types. The first part of the computation of this metric searches for a class and a couple, where object can flow between them. This is done as described in Section 5.2.1.3. Then a property, operation parameter or return value of the couple is identified, that has public visibility. This can be the object that flows between the class and its couple. Finally it is checked whether it has a non-trivial primitive type.

5.3 The Implemented Ontology Based Model Assessment Environment

The process of the ontology based computation of model based metrics (Figure 5-2) is similar to the process of the ontology based verification (Figure 4-1).

Accordingly, the tool developed for the evaluation of metrics is implemented as a new module of the existing ontology based verification tool. Figure 5-3 shows the relation between the modules. The common OntologyQueryEngine component contains the general query execution functionality using the JRacer interface of RacerPro. The modelMetrics component (similar to OntologyQueries) defines the queries in the format described in Section 4.1.2, the additional logic is programmed in Java. The assessment is based on the calculated metric values. The toolIntegration.ontologyBasedMetrics component contains the framework connector of the tool, which makes it available in the SDE Tool Manager component.
The assessment rules in the `modelMetrics` component have two types:

- In case of metrics described in sections 5.2.1 and 5.2.2 a predefined threshold can be given, and the components characterized by values above this threshold are reported.
- In case of the metrics described in section 5.2.3.1 the assessment is a simple indication (hint): if there are elements with the given property, a given TCG method should be considered.

The result of the assessment is provided in similar XML format like described in Section 4.1.2, containing the description of the metric, the assessment statement and the list of elements on which the assessment is based (elements exceeding the threshold or elements with the given property). An example is shown in Figure 5-4.

```xml
<check>
  <result>
    <queryID>AnalyzeLongestPath</queryID>
    <version>1.0</version>
    <description>Find longest path in a state machine.</description>
    <longDescription>Find longest path in a state machine.</longDescription>
    <resultList>
    </resultList>
    <conclusion>No long running statemachine found</conclusion>
    <boolConclusion>OK</boolConclusion>
  </result>
  <result>
    <queryID>AnalyzeSMCComplexity</queryID>
    <version>1.0</version>
    <description>Analyze cyclomatic complexity of state machines.</description>
    <longDescription>Analyze cyclomatic complexity of state machines. If the complexity is greater than 10 then it is reported as a complex state machine.</longDescription>
    <resultList>
      <item>
        <element>
          <type>StateMachine</type>
        </element>
        <instance>/uml2_models_AlarmSystem.uml.AlarmSystem.StateMachine_0</instance>
      </item>
    </resultList>
  </result>
</check>
```
<result>
  <queryID>AnalyzeFindTiming</queryID>
  <version>1.0</version>
  <description>Find timing parameters.</description>
  <longDescription>Search for timing parameters, because if found, timed automata based testing is advised.</longDescription>
  <resultList>
    <item>
      <element>
        <type>UML Entity</type>
        <instance>uml2_models_Alarmsystem_alarmsystem_TimeEvent_0</instance>
      </element>
    </item>
  </resultList>
  <conclusion>Timing parameter found, a timed automata based testing is advised.</conclusion>
  <boolConclusion>NOK</boolConclusion>
</result>

...
6 Verification of Process Models

The principles used for model-verification as described in Chapter 3 and which is used for the verification of application models, can also be applied to process models that define tool-chains. This will be explained in this chapter.

6.1 Scope

Our everyday life depends on software to a considerable extent and therefore the reduction of the risks of design and implementation faults is of utmost importance. Rigorous software development processes and model driven development with (automated) verification are two key approaches proposed for the development of safety critical systems.

Software development processes are more and more subject to regulations fixed in (general and domain-specific) standards that define criteria for the selection of appropriate development methods. Accordingly, if software is deployed in a critical environment then an independent assessment is needed to certify that its development process is compliant to the criteria stated in the relevant standard.

In this section a static analysis method will be proposed for the assessment of development processes and toolchains. It shows that ontology based verification is a viable approach to support the certification of development toolchains, e.g. that included in the MOGENTES Framework. The tasks and tools in the toolchain are modelled and then classified using an ontology that is constructed on the basis of the development standards, and a reasoning tool is applied to verify whether the criteria defined in the standards are satisfied.

The assessment of development tasks and toolchains is supported by a formal verification technique that allows the automated checking of the compliance to standards. Following the analogy of classical model checking (that is applied to examine whether a formal design model satisfies some temporal requirements) we represent the development process and tools in a process model (as introduced in the MOGENTES Framework) and use an ontology reasoning tool to check whether the criteria originated from the standard are satisfied.

This vision necessitates the following tasks:

- Formalisation of the requirements (criteria) in standards that concern the selection of methods and tools.
- Definition (or adaptation) of modelling techniques to describe the relation of methods, the capabilities of tools, and the construction of (domain-specific) development processes.
- Creation of techniques that check the compliance of concrete development processes (constructed by process designers) to the requirements.

The formalisation of the requirements and the model-based description of tools and methods allow the synthesis of processes and toolchains that are compliant to the standard. The process designer can be assisted by

- identifying missing methods and tools,
- proposing alternative solutions,
- offering a library of toolchain patterns,
- optimizing processes from the point of view of costs, time, safety etc.

6.2 Formalisation of the Requirements

Formalisation is a prerequisite of both formal verification and synthesis support. This work focuses on the development processes for safety critical applications, in particular to the EN50128 standard for railway applications [EN50128]. This standard defines five safety integrity levels (SIL) for development processes and describes methods that can be applied during the process. For each development step the mandatory (M), highly recommended (HR), recommended (R) and not recommended (NR) methods are described in a tabular form.

The main challenges during the requirement formalisation are the following:

- The development methods are refined hierarchically, i.e., several high level methods are decomposed into alternative combinations of lower level methods.
For each SIL different requirements are described. Accordingly, this introduces a new dimension into the requirement formalisation.

The sufficient conditions for every SIL are formulated using various combinations of the applied methods.

Table 6-1: The Verification and Testing methods (EN50128)

<table>
<thead>
<tr>
<th>Technique/Method</th>
<th>SIL1</th>
<th>SIL2</th>
<th>SIL3</th>
<th>SIL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Formal Proof (HR)</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2. Probabilistic Testing (HR)</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>3. Static Analysis (HR)</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>4. Dynamic Analysis and Testing (HR)</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5. Metrics (R)</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>6. Traceability Matrix (HR)</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>7. SW Error Effect Analysis (HR)</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>

In the following, the Verification and Testing step of the development process described in the EN50128 standard is presented as a simple example in order to demonstrate the mentioned concepts.

In Table 6-1 the recommendation level of some methods is shown, while the hierarchy of the methods is depicted in Figure 6-1. The combinations of the required methods are expressed as follows: “For Software Integrity Levels 3 and 4, the approved combinations of techniques shall be (1 and 3) or (3 and 4) or (4, 6 and 7).”

During the requirement verification step one of the sufficient conditions regarding the combinations of the methods should become true on the process model and none of the “not recommended” methods should be used in the process.

These requirements can be represented as Boolean expressions. If these expressions become true on the input process model, then this process model is standard compliant. Because of the challenges in the requirement description (e.g., hierarchical refinement of methods) ontologies are used to characterize the methods and the tasks in the input process models and finally reasoning is used to verify the requirements.

The formal representation of the hierarchical structure of methods can be provided by defining their ontology and describing it using description logic. The XML based representation of ontologies (Web Ontology Language [OWL]) is used in this work.

In this use case, concepts refer to the development methods and their relations include the refinement. This way the hierarchy of methods is expressed using concept hierarchy in the ontology TBox. In Figure 6-2 part of the graphical representation of the ontology is depicted that is constructed using the Protégé ontology modelling tool [Protégé].
6.3 Classification of Tools in the Tool Repository

The next step of the formalisation process addresses the tool repository. This repository is a collection of tools that can be used (in a given company) for the construction of the toolchains. Each available tool is classified on the basis of the concepts defined in the ontology constructed in the previous step, i.e., for each tool the supported methods are given.

Typically, the following items can be found in this repository:

- **Simple tools**, that realize a specific method described in the specification. For example, the SPIN model checker [SPIN] and the PVS theorem prover can be classified as tools supporting Formal Proof. The PolySpace can be classified as a tool supporting Symbolic Execution which is a Static Analysis method.

- **Toolchain patterns** that are formed by tools that must be executed in a predefined sequence, to support a given method. For example, Structure-Based Testing (which is a Dynamic Analysis and Testing method) can be supported by the following toolchain:
  - Model transformation from UML2 statechart model to the input format of the UPPAAL model checker.
  - Test generation for a given test goal using the VerifyTA tool (part of UPPAAL tool set).
  - Mapping abstract test cases to executable test cases.
  - Execution of test cases and measuring coverage by DECOS Test Bench (see D2.2b).

- **Abstract development steps** that are also allowed in order to support high level design of development processes. For example, “model checking of the reachability of hazardous states” can be classified as Formal Proof. In the next refinement steps tools like SPIN or SAL can be assigned to this abstract step.

These tools are represented on the **ABox** level of the ontology, they are instances of the concepts defined using the **TBox** described in the previous section.
### 6.4 Modeling the Development Process

The (domain-specific) development process is formalised using a process model which describes the tasks, input and output artefacts, the roles and tools involved in the development process (see Deliverable D2.1, the specification of the MOGENTES Framework).

The OMG's Software Process Engineering Metamodel (SPEM) specification defines a formal representation of business processes (including development processes). The Eclipse Process Framework (EPF) supports this specification and can be used to model the processes.

Using this framework the process designer

- can construct the specific development process, and can assign the available tools to the tasks of the process,
- can choose from the available toolchain patterns.

The tasks of the process implement particular methods that are classified using the ontology that describes the method hierarchy.

In order to support the checking of models with logical reasoning, the development process is represented using ontology. W3C's OWL-S ontology supports the description of service composition as well as business processes. In the following the process description capability of this ontology will be used to describe the development processes [OWL-S].

The OWL-S ontology defines the Process concept that can be an Atomic Process or Composite Process. Atomic processes correspond to single steps of the development processes while composite processes are decomposable into other processes. Their decomposition can be specified using control constructs such as Sequence and Choice.

The tasks of the process implement particular methods that can be classified by the standards, and on the basis of this classification the assessment can be supported. A formal representation of the hierarchical structure of methods can be provided by defining a new ontology. Here concepts refer to the development methods and their relations include the refinement. The OWL-S ontology is specialized in order to support this method classification and this extension is called methods ontology. In this extension new concepts are defined as subclasses of the Atomic Process concept (e.g., Fault Tree Analysis, Probabilistic Testing).

The process model constructed can be transformed into a OWL-S based process ontology. The required SPEM to OWL-S mapping is implemented using the VIATRA model transformation framework [Viatra]. To do this, the metamodel of the SPEM process modelling language and the metamodel of the OWL ontology language is constructed in the VIATRA model space. Then the "SPEM to OWL-S" model transformation (based on these metamodels), an importer for the SPEM models and an exporter for OWL ontology format are implemented.

The SPEM model is constructed using the SPEM UML profile, so the UML metamodel (extended with UML profile support) is used to represent the input model of the transformation. The OWL metamodel is constructed based on the OWL-DL Metamodel developed by the University of Karlsruhe [Haase]. This metamodel is easily integrated into the Meta Object Facility (MOF) architecture and can be applied in the VIATRA model transformation framework.

### 6.5 Assessment of Development Toolchains

Following the analogy of classical model checking (that is applied to examine whether a formal design model satisfies some temporal requirements) we represent the development process and the corresponding design tools in a process model by means of ontologies, and we use a reasoner to check whether the criteria originating in the development standard are satisfied.

The assessment of development processes is implemented by an assessment toolchain in order to support automatic execution of the steps starting with the SPEM model transformation into ontology based models and concluding with the reasoning (Figure 6-3).
First the process model is constructed by domain experts using the EPF Composer tool. This input model is transformed into an OWL-S based ontology, then the tools to be used and atomic tasks are classified using the methods ontology. The output model is a process ontology. Finally, the conformance of the development process to the standard can be checked using the reasoner tool, which is executed on the process ontology.

Using the approach described above, all of the tasks, the tools and thus the development toolchains in the process model are characterized using the concepts represented in the ontology.

Using the concepts defined in the ontology, the necessary and sufficient conditions for the selection of methods and the dependency on the safety integrity level should be represented. The conditions can be described using an ontology query language, for example the New RacerPro Query Language (nRQL) [nRQL] or the RDF Query Language [RQL]. The query should check whether the required combinations of the methods are included in the process model. Moreover, the ordering of the methods can also be formulated using these query languages.

Accordingly, the conformance of the selection of methods and their supporting tools in the process model to the standard can be checked using an ontology reasoner. We use the Protégé [Protégé] ontology modeling tool and the Racer Semantic Web Reasoning System [Racer]. This way the assessment can be supported by reusing existing formal methods and checker tools.
7 Summary and Outlook

Ontology based model verification.
In the MOGENTES project efficient test generation methods are developed. The inputs of the test generation process are high level models using UML or Simulink. As the quality of the generated tests is strongly correlated to the quality of the original models, it is essential to verify these models beforehand. In this deliverable we have presented a verification method that is based on a common semantic representation format suitable for a wide range of modelling languages.

We have demonstrated how constraints can be verified in general using the ontology based representation of models. We have defined how UML models can be mapped to the ontology models as a specific workflow, and implemented this transformation using the VIATRA2 model transformation framework. We have also defined constraints in the context of UML models that are relevant for MOGENTES and implemented the checking of these in the ontology domain using the RacerPro reasoner. The whole verification process is integrated into the tool integration framework developed in WP2.

As another application, the ontology based verification approach is also applied to process models that are defined and used in WP2 for the execution of transformations and test case generation. In this case the sources of constraints are development process related standards and regulations. This extension demonstrates that the approach can be re-used at the level of process engineering and, in accordance with the objectives of the project, can contribute to the standards-based assessment and potential certification of modelling and test generation processes.

Ontology based derivation of complexity and testability metrics.
Ontology based representation of our models also allows further analysis of the models using the capabilities of the existing reasoners. It is possible to calculate those structural complexity metrics of the models that are widely used as indicators of fault proneness (helping to identify those software components which need more thorough testing). The testability metrics can also guide the model developer, for example, in how to modify the models to be less prone to faults and well testable with a given testing approach. Several metrics can also be used to give guidance about which test generation method is appropriate or efficient for the given application.

Model Validation and Verification in the UML/OOAS track.
In addition to model verification on the UML-level, MOGENTES employs several other classic static verification techniques in the UML/OOAS track. Firstly, after translating the UML models to OOAS, the resulting OOAS is syntax and type checked. This step ensures that the given OOAS is well formed. Note that in addition to 'normal' type-checking, the following static validation checks are also being carried out: (1) range checks on assignments, (2) checks whether all instantiated classes are mentioned in the system assembling block, (3) whether only a finite number of objects is created, and (4) whether visibility and call constraints are being fulfilled.

Secondly, any well-formed OOAS can be validated with the tools offered by the CADP toolbox. The model checker allows easy checks of for example, deadlock states, or whether there is a loop of internal actions. Besides model-checking, we can also use a bisimulation checker to prove the inclusion-relation between different models. This has been used in the car alarm system case study, where we showed that two different UML models shared the same basic functionality. Obviously, besides techniques that statically validate a given model, dynamic model animation is also a useful validation technique.
# 8 Abbreviations and Definitions

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABox</td>
<td>Assertion Box</td>
</tr>
<tr>
<td>FOL</td>
<td>First Order Logic</td>
</tr>
<tr>
<td>KRSS</td>
<td>Knowledge Representation System Specification</td>
</tr>
<tr>
<td>MOGENTES</td>
<td>MOdel-based GENeration of Tests for (dependable) Embedded Systems</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>ODM</td>
<td>Ontology Definition Metamodel</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>SPEM</td>
<td>Software Process Engineering Metamodel</td>
</tr>
<tr>
<td>TBox</td>
<td>Terminology Box</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VIATRA</td>
<td>Visual Automated Model Transformations</td>
</tr>
</tbody>
</table>
9 References


[BPMO] Dieter E. Jenz: Defining a Private Business Process in a Knowledge Base


[EMF] Eclipse Modeling Framework
http://www.eclipse.org/modeling/emf/

[EN50128] CENELEC EN 50128: Railway applications - Communication,signalling and processing systems - Software for railway control and protection systems.
http://www.cenelec.eu

[EODM] Eclipse Ontology Definition Metamodel
http://wiki.eclipse.org/MDT-EODM

[EPF] Eclipse Process Framework project
http://www.eclipse.org/epf/


