DE GRUYTER OPEN

The MODUS Approach to Formal Verification

Lukasz Brewka, José Soler, Michael Berger
DTU Fotonik, Denmark

Abstract

Background: Software reliability is of great importance for the development of embedded systems that are often used in applications that have requirements for safety. Since the life cycle of embedded products is becoming shorter, productivity and quality simultaneously required and closely in the process of providing competitive products **Objectives:** In relation to this, MODUS (Method and supporting toolset advancing embedded systems quality) project aims to provide small and medium-sized businesses ways to improve their position in the embedded market through a pragmatic and viable solution **Methods/Approach:** This paper will describe the MODUS project with focus on the technical methodologies that can assist formal verification and formal model checking. **Results:** Based on automated analysis of the characteristics of the system and by controlling the choice of the existing open-source model verification engines, model verification producing inputs to be fed into these engines. **Conclusions:** The MODUS approach is aligned with present market needs; the familiarity with tools, the ease of use and compatibility/interoperability remain among the most important criteria when selecting the development environment for a project.

Keywords: software quality, formal verification, embedded systems, translation tool selection

JEL main category: Economic Development, Technological Change, and Growth
JEL classification: O31
Paper type: Research article

Received: 15, July, 2013
Accepted: 12, January, 2014

Citation: Brewka, L., Soler, J., Berger, M. (2014), "The MODUS Approach to Formal Verification", Business Systems Research, Vol. 5, No. 1, pp.21-33.
DOI: 10.2478/bsrj-2014-0002

Acknowledgments: This research activity is funded under the EU Research for SME associations FP7 project, MODUS-Methodology and supporting toolset advancing embedded systems quality (Project No.286583).

Introduction

The MODUS project was initiated to provide a sustainable and pragmatic tool set that will allow small and medium-sized businesses to improve their ranking in embedded systems engeneering.

By the use of formal description Techniques, MODUS will develop and validate a set of technical methods, as well as an open and customizable toolsets enhancing embedded systems by providing (MODUS (2013a)) :

- Model verification by performing the selection of the available open-source model verification engines can be supplied, based on the automated analysis of system characteristics and the production inputs in these engines.
- Connection to standard simulation platform for HW / SW co-simulation.
- Software to optimize performance through automated design transformations.
- Customizable source-code generation in line with coding standards and conventions.

The project will also provide features and open interfaces to customize and extend MODUS toolkit for use with various formal specification techniques, modeling techniques, programming languages, platforms, etc. MODUS is not intended to be low competition with CASE tools used in embedded software engineering at the moment. On the other hand, you want the project to allow the adoption of quality strategies by supplementing these tools and allow existing investments in technical-know with continued use. In the next section, the tools for formal verification is described, followed by methods to control the selection of techniques and strategies for the selection tool.

Supported verification tools/languages and their properties

The selected model verification tools to be supported by the MODUS toolset are SPIN and RAISE. Both of the tools are LTL (Linear Temporal Logic) model checkers. The sections below present more details about the properties of these tools and the languages they use for model description.

SPIN

SPIN and PROMELA are focusing on the process interaction, i.e., describing how system components communicate with each other (Holzmann, 2003). As already, mentioned not much attention is given to internal computation processes. The process interaction can be modelled in a number of ways:

- rendezvous primitives (synchronous)
- asynchronous message passing through buffered channels
- access to shared variables
- any combination of the above

SPIN itself provides a methodology for matching the system design expressed in PROMELA language and the LTL formula describing desired/correct behaviour of the system.

PROMELA is a language crafted for describing models of distributed systems. It expresses the model description using a language similar to C with some notation from the guarded command language by Dijkstra and the CSP language from Hoare (particularly to describe interaction between processes). A PROMELA model consists of (Holzmann, 2003):

- variable declarations with their types
- channel declarations
- type declarations
- process declarations
- init process (optional)

In PROMELA a process is a basic building unit of the system It is defined by a so called "proctype" definition that contains the process' name, the process' list of parameters, its declaration of local variables, and the sequence of local statements. Meenakshi (2004) provides a following example of process definition:

```
proctype Sender(chan in; chan out)
{
bit sndB, rcvB;
do
:: out ! MSG, sndB ->
in ? ACK, rcvB;
if
:: sndB == rcvB -> sndB = 1-sndB
:: else -> skip
fi
od
}
```

Models written in PROMELA can (and usually will) contain more than one process. Multiple processes will run in parallel, communicating with each other using the interaction methods described at the beginning of this section. The state of the process is defined by its local variables and the process counter. Processes are invoked by using a run statement inside the init process or by adding to active keyword. The process creation can be placed in arbitrary places within the model. The variables in PROMELA require declaration defining the type and name of the variable, prior to its use. There are these five different types of variables available in PROMELA:

- bit
- bool
- byte
- short
- int

It is also possible to declare arrays and records. All variables (local and global) have the initial value of 0. Conflicts in type during value assignment are resolved at run-time.

Communication. As indicated earlier channels are used to enable communication between processes. There can be the following two types of communications:

- Message-passing or asynchronous
- Rendezvous or synchronous

Channels are of a FIFO (First-In First-Out) type and is defined using array type defining the number of messages that can be occupied by the channel. The declaration also specifies the type of elements that can be passed using this channel.

Rendezvous communication is established using channels with dimension zero. If you send through a channel is enabled, and if there is a corresponding receiver that can be performed simultaneously, so both statements are enabled. Both statements will be able to perform a single transition.

Statements.
Promela statements are separated by a semicolon. One of the basic statements Promela projections (do something) to distinguish , printf , assert (expression) (Check whether the expressions property is valid in a state) if statements (executable if at least one of the options is non-blocking) and submit observations .

The feasibility of Promela statement depends on its type and value - expression is evaluated . Assignment statements , skip, pause, printf are statements that are always executable . Expressions are executable if it does not evaluate to zero. This means that if or executable statement is , if at least one guard evaluates to true. The communication settings are executable statements , depending on the status of the channel , a transmit executable for non- channels and a receiver statement executable for non-empty channels.

To the statements of the group for a specific process in a sequence in a single step , which does not include statements of other processes performed nested , one can use an atomic statement. The feasibility of the atomic statement depends executablility to the first survey. If one of the following statements is not executable , the indivisibility of atomic propositions are broken and other processes can be nested .

Another example of a one-step embodiment, a d- point . In contrast to the statement if the atomic d- step involves blocking mode , it causes a runtime error.

Finally timeout statement is a statement that is executed if no other statements contained in any of the other processes are executable (this function can not be used to be modeled timeouts that are involved in the system design.)

RAISE

RAISE (Rigorous Approach to Industrial Software Engineering) is a tool-set that consists of: a method for software development, RSL (RAISE Specification Language) specification language, and a computer tools for automated model checking, analysis, and translation (George, 2008).

The key for system verification using the RAISE tool-set is to obtain a specification in RSL. RSL, as a specification language, provides a wide range of opportunities for expressing the modelled system: it allows property and model-oriented styles, applicative and imperative styles, as well as sequentiality and concurrency.

Modules. A RSL specification is divided into modules. A module should capture the types, values and axioms that characterize the system or its parts. A generic module definition in RSL has the following form:

```
id =
 class
   declaration-1
   …
   declaration-n
 End
```

A declaration will start with a keyword identifying the kind of declaration (e.g. type, value, axiom). The following RSL specification of a database illustrates RSL type declaration. This is further clarified in the following subsections (George, 2008):

```
DATABASE =
 class
 type
   Person
   Database = Person-set
 value
   empty: Database
   register: Person × Database → Database
   check: Person × Database → Bool
 axiom
   empty ≡{}
   ∀p : Person, db : Database • register(p,db) ≡{p} □ db,
   ∀p : Person, db : Database • check(p,db)≡p ∈ db,
End
```

Type declaration. A type is a collection of logically related values. There exists a number of build in types that are predefined in RSL. Beside those, a RAISE user can also define his/her own types. One can create an abstract type like Person in the example above. Abstract types do not have any predefined operators for manipulating their value (beside the "=" operator, used for comparison of two values).

Another kind of type declaration is using the "=" operator. Using "=" expresses that the new type is representing the expression on the right side of the operator. Back to the previous example, using "=" declares the Database as a set of people i.e., the type containing all finite subsets of the set of values in Person. RSL Atomic types with their values and operators are listed below (Haxthausen, 2010):

- Bool values: true, false operators: =, ≠, ∧, ∨, ⇒, ~, ∀, ∃
- Int values: ..., -2, -1, 0, 1, 2, ... operators: =, ≠, +, −, *, /, ↑, \, <, ≤, >, ≥, abs, real
- Nat values: 0, 1, 2, ... operators: =, ≠, +, −, *, /, ↑, \, <, ≤, >, ≥, abs, real
- Real values: .., -4.3, ..., 0.0, ..., 1.0, … operators: =, =, +, −, *, /, ↑, <, ≤, >, ≥, abs, int
- Char values: 'a', . . . operators: =, ≠
- Text values: "Alice", . . . operators: =, ≠

For declaration of composite types it is possible to choose amongst:

- A product - an ordered collection of values, not necessarily distinct, of some given types (possibly different).
- A list (sequence) - an ordered collection of values, not necessarily distinct, of the same type.
- A set - an unordered collection of distinct values of the same type.
- A map (or table) - an unordered collection of pairs of values.

Value declaration. Values can be named in the value declaration. The simplest example of value declaration is in the form "id : value" and can be seen in the provided example of the RSL declaration of a database (see empty). The actual value that is identified by empty is described in one of the axioms (described later).
The next value example (register) represented in that example, declares a function that adds a person to the database, it represents a database after performing the registration i.e., when the person was added.

Finally, the third type of value definition in that example, defines a function check. When check is applied to database and a person, depending on whether particular person is part of Person-set defined by Database, a Boolean true of false is returned.

Similarly like for previous value expression the axiom contains a detailed characterisation of this value.
One can distinguish 3 three forms of value definition in RSL (Haxthausen, 2010):

```
o  explicit function definition:
value
is_in : Person × Database → Bool
  is_in(p,db) ≡ p ∈ db

o  implicit function definition:
value
 square_root : Real → Real
 square_root(r) as s
  post s ∗ s = r ∧ s ≥ 0.0
  pre r ≥ 0.0

o  axiomatic definition:
value
 is_in : Person × Database → Bool
axiom
 ∀ p : Person • is_in(p, empty) ≡ false,
 ∀ p : Person, db : Database • is_in(p, register(p, db)) ≡ true
```

Axiom declarations. Axioms are used to bring the property value names for expression. Let's go back to the database, for example, the first axiom is an expression of type bool and declares that the value expression is empty, and {} are the same. All axioms are Boolean expressions evaluate to true. The second axiom is presented in our example suppresses the register function. It uses level expressions show that for all people and all databases, the data for a couple of person p and database db is used, is equivalent to a union {p} □ db (ie, a new series that both subsets). Axiom defines Check the function that will be registered if it belongs to the group hosting the database.
Module extension. RAISE also allows for module extension where one can declare a new module that adds type, values, and axioms to existing module. For that purpose one should use extend <module_name> with expression.

Mechanisms for guiding the verification tool selection

Relevant model properties

The most important feature that should be considered when deciding on the formal methodology tools is the possibility of representing the properties of the system in the formal language selected for performing the verification and validation. Different formal description languages have different set of features that they can express. During validation it is important to identify the crucial system properties and ensure that these properties can be expressed using a particular formal description. Otherwise the verification makes no sense since, when important parts of specification are lost; essentially it is a different system that is validated.

UML and SysML diagrams are the sources of the model descriptions, as far as MODUS tool-set is considered, as presented in previous deliverables (MODUS 2013a, 2013b, 2013c, 2013d). As such, the set of the features contained in these diagrams will depend on the types of the diagrams that are integrated in the model.

Definitely system is described in different degree of details depending if it contains structural or behavioural modelling data. This is clarified in the following:

The most commonly used diagram during the software development is a class diagram. These types of diagrams present the system structure by showing classes with their properties and relations between them.

On the other hand, behavioural diagrams can express a dynamic nature of the system. They describe states of an object during its lifetime

Two translation tools selected for the integration with the MODUS tool-set, i.e. UML2RSL and Hugo/RT have different capabilities when it comes to translation model. The following sections present the capabilities of the tools and describe certain translation possibilities and examples.

UML2RSL

UML2RSL used class diagram, to formulate RSL specification. As a result of the translation, in order to obtain a specification of a plurality of modular RSL files.

A top-level module is stored in the file S.rsl. It contains a specification of the model with the whole class diagram. S.rsl uses one set of modules containing a specification of one of the classes from the diagram. The name of these modules is given to match the class name in capital letters, followed by "S_". Each RSL module for a class is created with a lower level module corresponds to an object of the given class, named after the class in capital letters, followed by "_". Each of these lower level modules used TYPES.rsl module where all the abstract types defined in the illustration (George,2008).

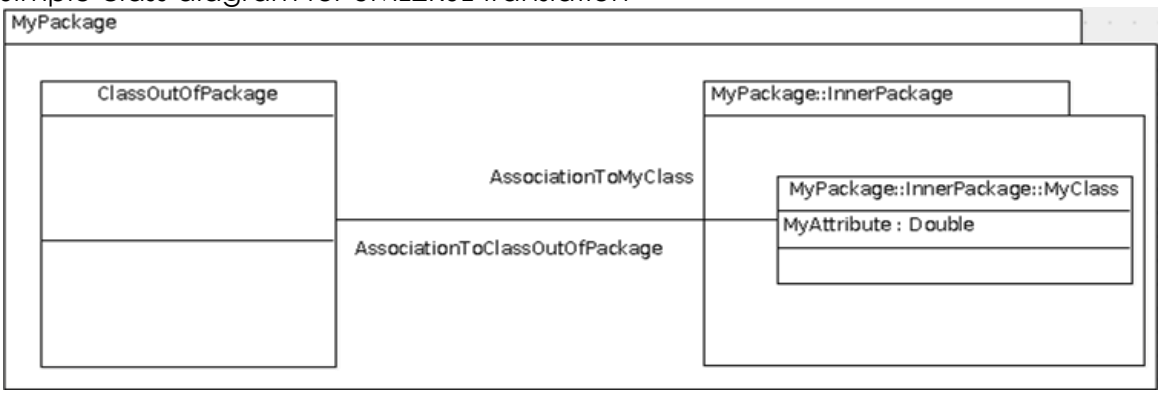
Considering a simple example of a class diagram presented below six files will be created (Figure 1):

- CLASSOUTOFPACKAGE_.rsl
- CLASSOUTOFPACKAGES_.rsl
- MYCLASS_.rsl
- MYCLASSS_.rsl
- S.rsl
- TYPES.rsl

They can form the following dependency diagram presented in Figure 2.

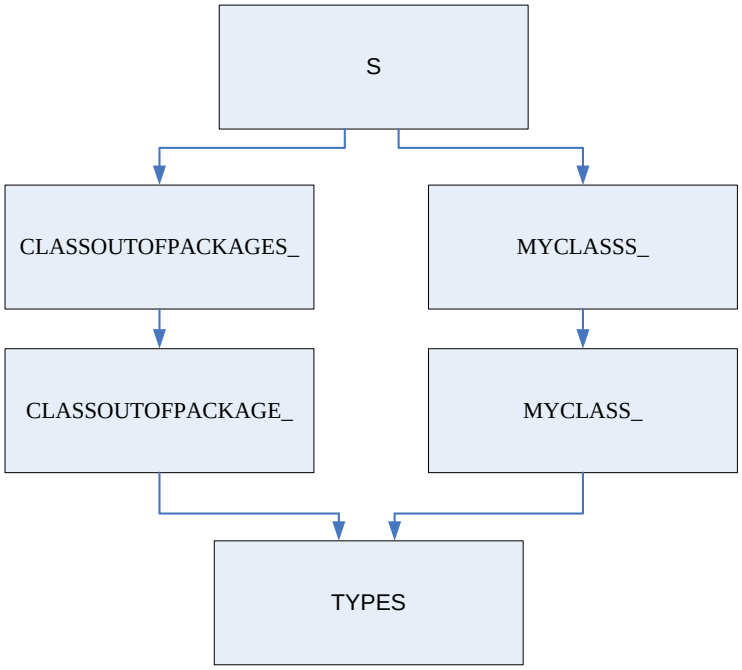
Figure 1

Simple class diagram for UML2RSL translation



Source: Authors

Figure 2
RSL output dependency diagram



Source: Authors

As a result of the translation the MYCLASS_.rsl will take a form: TYPES (George, 2008)

```
object MYCLASS_ :
  with TYPES in
  class
    type
        MyClass
    value
        MyAttribute: MyClass -> MyAttribute,

        update_MyAttribute: MyAttribute >< MyClass -~-> MyClass
        update_MyAttribute(at, o)  as o' post MyAttribute(o') = at
            pre preupdate_MyAttribute(at, o),

        preupdate_MyAttribute: MyAttribute >< MyClass -> Bool,

        Association: MyClass -> ClassOutOfPackage_Id,

        update_Association:
        ClassOutOfPackage_Id >< MyClass -~-> MyClass
        update_Association(a, o) as o' post Association(o') = a
            pre preupdate_Association(a, o),

        preupdate_Association:
          ClassOutOfPackage_Id >< MyClass -> Bool,

      consistent: MyClass -> Bool
  end
end
```

The remaining files that are output from the UML -to- RSL transition, listed in Appendix A: Translation results UML2RSL. A class, in addition to the normal function of returning the value of an attribute, very often contains operations change the attributes. It could be an event that occurs, this attribute modification in a particular state. This is handled by the compiler to generate the RSL functions for this purpose. Their claim is to be completed by the user (see e.g. preupdate_Association).

According to this example of the type described MyClasss is the set of all possible states of the class MyClasss, representing a number of groups of articles or MyClass all possible sets of objects that can be observed at a given time. It can be described as the class of container.

For each class in the class can create new objects and damaged or altered existing objects. This is the existence of some typical features (empty, add part is_in, sheep, and update) that can be on the set of instances of each class and the work of a translator resist. Seen an example of these features in CLASSOUTOFPACKAGES_ and MYCLASSS_ files (see Appendix A: Translation results UML2RSL).

To check the consistency of the whole system in order to verify that keep all constraints are a number of axioms defined top-level. This makes it possible to check whether the system is done in a consistent state before and after each change in status. For this reason, a number is generated by similar functions. They are included in the top level module S, but also a set of Boolean functions is formed in the lower level modules the. Lower level texture features, the user to check the consistency of objects and classes. The top feature using features lower level checks the consistency of the whole system.

This simple example also shows the association, which is translated as two RSL functions between the classes involved (or, if the association is navigable in only one direction).

The UML2RSL translation tool also accepts and executes test: Composition, aggregation, generalization (but only translates single inheritance) abstract, root, leaf and template classes. Dependencies ignored (details can be found in (George 2008).

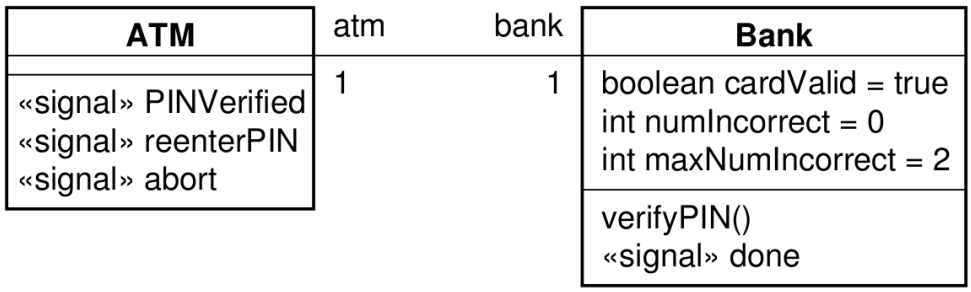
HUGO/RT

A translation performed by Hugo/RT goes beyond the basic class diagram translation (Knapp, 2008). Hugo/RT can translate the UML models that contain classes with state machines, collaborations, interactions, and OCL (Object Constraint Language) constraints.

The state machines describing the state the objects can be in, can be complemented by another dynamic view of the system, namely, the sequence and collaboration diagrams, describing the interactions between different objects in the system. Hugo/RT can be used to verify whether these complementary views of dynamic properties of the system are coherent. In other words it allows verifying whether the system described by the state machines can fulfil the interaction described in the collaboration.

The UML state machines should be described in the context of a UML class. The classes need to declare all events that its state machine can handle (Knapp, 2008). Hugo/RT translation for every state machine creates a separate PROMELA process.
Considering an example borrowed from (Schäfer, Knapp, Merz, 2001). It is possible to define a system that is composed of an ATM and a Bank, presented by a simple class diagram in Figure 3.

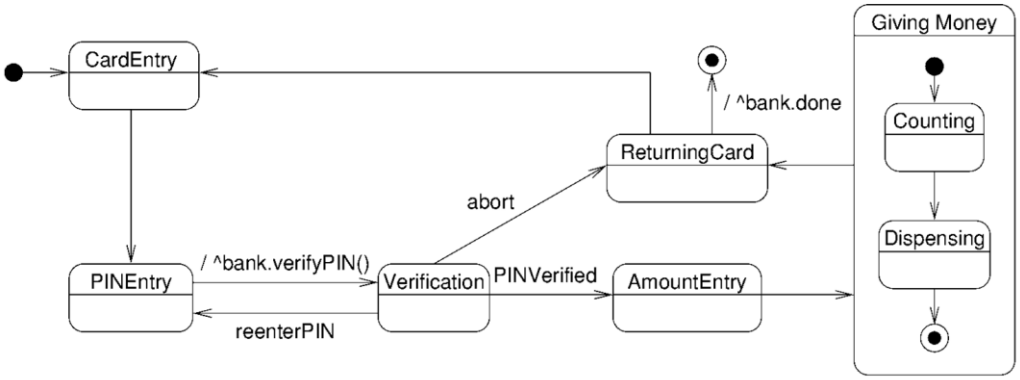
Figure 3
ATM - Bank class diagram



Source: Schäfer, Knapp, Merz, 2001

Each of the classes contains a state machine. For instance let's consider a state machine of the ATM module (see Figure 4). This state diagram contains simple and composite states together with guarded and unguarded transitions.

Figure 4
ATM state machine



Source: Schäfer, 2001

Strategies for selection of the formal description tools

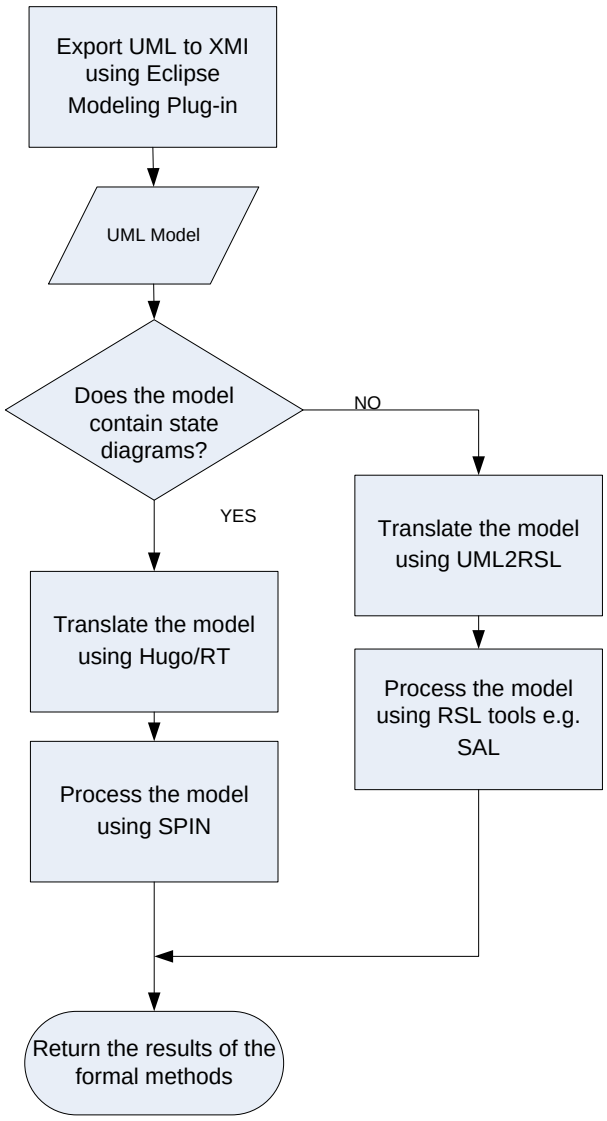
This section covers the possible strategies for selecting a proper formal verification/ validation path. This includes a selection of the UML to formal language translator and as such also the model checker. The strategy included in this deliverable is taking into account only the tools considered for the first stage of the MODUS tool-set development.

As described earlier, the two model checkers that are identified as first priority tools to be integrated within MODUS tool-set are SPIN and RAISE (SAL). This means that Hugo/RT and UML2RSL will be the tools that perform the translation between a UML model and a formal description. Considering different properties of the aforementioned translators, the strategy for the selection of the tools for the formal verification and validation is rather simple (at least at this stage of the development). Since UML2RSL handles only UML diagrams containing class diagrams it can only be considered for this basic types of models.

UML diagrams containing state machines, within their classes, cannot be translated into RSL, because the translator is not processing behavioural diagram types. As such, large part of the system description would be lost.
For diagrams that contain state machines describing the behaviour of the objects of a certain class type, Hugo/RT should be used in the currently planned MODUS tool-set formal verification block. The resulting PROMELA model produced by the Hugo/RT translation can later be fed to SPIN for formal verification according to collaboration diagrams and OCL constraints.

If class diagrams that do not contain any state diagrams are translated using Hugo/RT, the tool will prompt a warning indicating that no behaviour description was detected and almost empty file (containing only idle process) will be generated.
This strategy for selection of the formal verification tool to be used in each case can be implemented as a simple algorithm as depicted in Figure 5.

Figure 5
Formal methods selection



Source: Authors

Conclusions

MODUS is targeting the market of tools for embedded software engineering. The project will develop a toolset advancing embedded systems quality that will target the growing group of SMEs (and bigger companies as well) specialising in the development of embedded systems in different industrial sectors (e.g. avionics, automotive systems, consumer electronics, telecommunications systems, etc).

It should be emphasized that MODE does not aim to compete with the major suppliers of CASE tools currently used in embedded software engineering to become. On the contrary, the project aims to facilitate the implementation of quality strategies by preserving existing investments in technical know-how and tools. The MODUS approach is aligned with present market needs; the familiarity with tools, ease of use and compatibility/interoperability remain among the most important criteria when selecting the development environment for a project. Specifically, this paper has focused on the formal verification part of the MODUS toolsets, but the uniqueness of the MODUS toolsets lies in the combination of formal verification, HW/SW co-simulation, SW performance tuning and customizable source code generation.

References

1. George, C. (2008), "RAISE Tool User Guide", available at: http://www.iist.unu.edu/newrh/III/3/1/docs/rsltc/user_guide/html/ug.html (12 September 2012).
2. Haxthausen, A. (2010), Lecture notes, 02263 Formal Aspects of Software Engineering, available at http://www2.imm.dtu.dk/courses/02263/F14/index/ / (10 September 2012)
3. Holzmann, G. J. (2003), The Spin Model Checker: Primer and Reference Manual, Boston: Addison-Wesley.
4. Knapp, A. (2008), "Hugo/RT", available at: http://www.pst.ifi.lmu.de/projekte/hugo / (12 September 2012).
5. Meenakshi, B. (2004), A tutorial on SPIN, Bangalore: Honeywell Technology Solutions Lab.
6. MODUS (2013a), Deliverable D2.1 "State-of-the-art review and identification of technological requirements", Internal documentation.
7. MODUS (2013b), Deliverable D2.2 "MODUS functional and technical specifications", Internal documentation.
8. MODUS (2013c), Deliverable D3.1 "Methodological framework for LNR-based model transformation and code generation", Internal documentation.
9. MODUS (2013d), Deliverable D3.2 "LNR-based model transformation and code generation modules", Internal documentation.
10. Schäfer, T., Knapp, A., Merz, S. (2001), "Model Checking UML State Machines and Collaborations", Electronic Notes in Theoretical Computer Science, Vol. 55, No. 3, pp. 357-369.

About the authors

Lukasz Brewka received his M.Sc. degree in 2008 and his Ph.D. degree in 2012 both from Technical University of Denmark, Department of Photonics Engineering. He was involved in European projects ICT-ALPHA and MODUS. His research interest includes quality assurance in telecommunications and software systems - from network QoS to software QA. Author can be contacted at **brewka111@gmail.com**

José Soler earned his PhD in Electrical Engineering (2006) by DTU (Denmark), and his MSc in Telecommunication Engineering (1999) by Zaragoza University (Spain). Currently he is Associate Professor on Telecommunication Networks at DTU Fotonik. His research interest include heterogeneous networks integration, telecommunication related software and services. Author can be contacted at **joss@fotonik.dtu.dk**

Michael S. Berger was born in 1972 and received the M.Sc. EE and Ph.D. from the Technical University of Denmark in 1998 and 2004. He is currently Associate Professor at the Technical University of Denmark within the area of switching and network node design. Currently, he is leading a project on next generation IP and Carrier Ethernet networks partly funded by the Danish National Advanced Technology Foundation. Author can be contacted at **msbe@fotonik.dtu.dk**