OMEGA

Correct Development of Real-Time Embedded Systems

IST-2001-33522

Title : Adapting and optimising untimed model checking tools to UML

Author(s) : OFFIS

Editor : Verimag

Date : 20/06/03

Identifier : IST/33522/WP 2.2/M2.2.3 (appendix)

Document Version : 2.0

Status : Final

Confidentiality : Restricted

Abstract :

This part of the report M2.2.3 is a revised version of the deliverable D2.1.1 describing the main structure of the verification tool with discrete time model. It gives an overview of the tool obtained by adopting an existing untimed model checking tool to the UML specifications. The current version of the tool allows to check a subset of UML models with the object-oriented features defined in the Omega kernel model. Covered aspects include in particular the concept of active and reactive objects of non-trivial multiplicity, different kinds of associations, synchronous and asynchronous communication between objects, inheritance with static and dynamic polymorphism as well as a detailed presentation of hierarchical UML statecharts. The main stages of the UML-model translation to the model-checker format – both for design-tool dependent and XMI-based sub-components – are pictured.

Table of Contents

Document history

Revision	Date	Author	Comments
1.0	23/12/2002	B. Josko, A. Votintseva	Tool description
2.0	23/06/2003	A. Votintseva	Implementation description: architecture of the OXF, data flow in the tool

Introduction

Today's designers are faced with heterogeneous integrated toolsets in order to be able to manage growing complexity of designing safety critical embedded control applications. Usually, these toolsets integrate high-level design and specification tools/languages (e.g. Statemate [9], Rhapsody [8], Scade [10], Matlab/Simulink [11], VHDL [12]) with powerful back-end technologies, like formal verification, prototyping, and testing. The integration of different front-end tools with these back-end technologies within toolsets provides the services offered by the technologies independent of the chosen specification style.

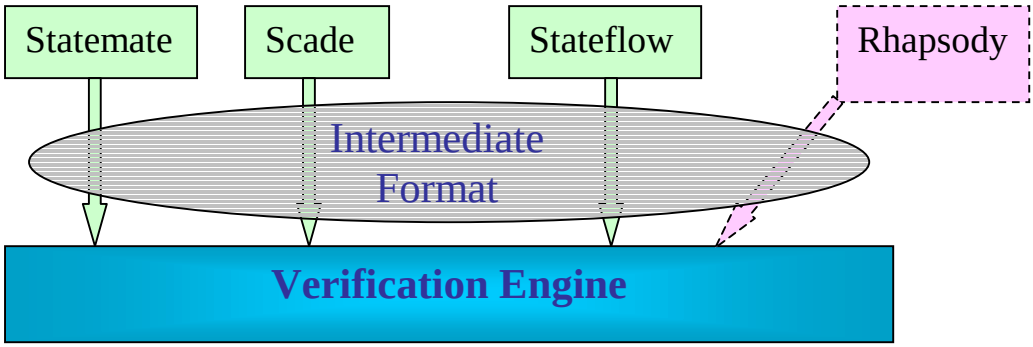


Figure 1. Tool Integration

As an example of a back-end technology, a verification environment was developed in OFFIS, which has been integrated with different high-level specification tools – Statemate, Scade, Stateflow – in such way, that a generalised intermediate format was introduced allowing to represent different models in a unified manner (shown in Figure 1). Using the tool interfaces developed for different CASE-tools, the models from these tools are translated into the common intermediate format which feeds the verification engine. The objective within the Omega project is to provide a powerful integration for UML [13] in this tool chain. As a starting point, a tool prototype for such integration was developed within another IST project, WOODDES [15]. This initial prototype treated simple Rhapsody models and very simple properties (like "drive to state"). For more complicated models and non-trivial properties, additional optimisation techniques are required at different stages of the translation from a UML tool to the common intermediate format and also over this format as well. Another task for the initial prototype (in the scope of the WOODDES project) was to provide a means for the property specifications within the Rhapsody tool, e.g. introduction of patterns with their parsing, which can be used further, possibly, in combination with other specification techniques, such as temporal logics, LSC [2], symbolic timing diagrams [7].

Thus, our task in the OMEGA project (work-package WP 2.1) is to efficiently translate advanced object-oriented features – dynamic object creation/destruction, inheritance, polymorphism, different kinds of inter-object communication, hierarchical statecharts etc. – from the Rhapsody representation into the internal common format used by the existing verification environment, applying different kinds of optimisation and developing better interface for the visualisation of the verification results. Another task – within WP 2.2 – is the integration of this verification tool with XMI representation of Omega UML models.

1. The Tool Overview

The task in the reporting period was the completion of compiling and elaborating object-oriented concepts of UML models from the Rhapsody representation to the model checker format called SSL/SMI. This format allows to represent the kernel language defined in D1.1.2 (part 1).
As a result of the implemented translation and elaboration techniques, the tool can verify a model with several communicating objects (related via associations of different kinds), covering such concepts as (multiple) inheritance, static and dynamic polymorphism, complete hierarchical statecharts etc. As requirement specifications the tool can now accept temporal logic formulas defined using patterns or written in general form (to be used by advanced users) as well as LSC specification, which are translated into observer automata and feed the model checker. These requirement specifications plays an important role for the elaboration of the SSL model.

1.1. Overview of SSL and SMI

SSL – System Specification Language [5] – is a high-level language that serves to specify the decomposition of high-level designs into components and their interconnections. SMI – System Modelling Interface [6] – is a simple imperative programming language that serves as a generalised intermediate format between high-level design tools and different validation environments. The dynamic behaviour of every subsystem of a whole design from high-level design tools is represented by one SMI program which feeds the model checker. SSL comprises concepts to describe the architecture of a system, its components, the interfaces of these components, and, as an important point, the interconnection of the components' interface objects. The language SSL allows to describe the structure of a system, as well as the dynamic aspects of the system's components (specified via methods with SSL-actions), making the translation of models from high-level design tools to the model checker input format – SMI – effective. In other words, SMI provides an operational semantics of the behavioural part of SSL. Note that it is not possible to translate automatically general SSL into SMI representation. For this purpose, an SSL model should be transformed into simpler SSL representation.

1.1.1. High-level language SSL

SSL meets the industrial requirement of supporting multi-formalism specifications: the designer can – dependent on the component of the embedded control application under consideration – choose to pick the modelling techniques best suited for a given application, e.g. either the more state-oriented modelling techniques of statecharts incorporated in Statemate, or the dataflow modelling style supported by the synchronous language Lustre incorporated in the Scade tool, or object-oriented technology with UML supported by

Rhapsody. To describe design hierarchies, the designers use the schematic entry mechanisms provided by the high-level design tools.
The added power coming from SSL rests in the capability to freely bind components in a design hierarchy to models expressed in any of the supported specification languages of a used toolset through the concept of *configurations*. This concept is exploited to express multi-formalism design hierarchies, and also to associate different views with one design component, such as providing for one component a behavioural model (expressed using, e.g. statecharts) and a set of requirements (expressed using, e.g. LSC or Symbolic Timing Diagrams – STD [7]).
SSL fulfils the following requirements:

- *Interface descriptions* are provided which support a uniform style of specifying visible design objects and their types independent of a used design tool;
- *Description of components* of an *architecture* can be seen together with their interconnections;
- *Configurations* bind components of a structure to SMI programs which represent modules of the supported specification languages.

In SSL a module can appear in three different ways: as an entity, architecture or configuration. The interface of a module is defined in its *Entity Declaration* by stating the ports of the interface, which can be of one of two sorts – *Inputport* or *Outputport* – and is characterised through its identifier, mode and the type of the message it can transmit. In a completed system, the ports of each entity are connected to so called *Signals*. The signals determine unambiguously the interconnections of the different modules with each other in a regarded system. An entity declaration is a separately compilable unit.
To an entity declaration, there can be one or more attached *Architecture Descriptions*. Each architecture defines one conceivable implementation of the module either as a behavioural or as a structural description. Behavioural modules are defined by SSL methods (allowing overloads), which can be derived automatically out of high-level behavioural specifications. Structural descriptions are defined using SSL's architecture entities. Syntactically, structural descriptions are constructed via the declaration of *Signals* and *Components* together with their interfaces, the *instantiation* of the components in an architectural context, and *wiring* of the component's ports through the declared signals. Component declarations can be multiply instantiated. The wiring of port instances is done by so called *port maps*. They bind the ports of a component instance either to ports of the entity for the upper architecture or to internal signals which constitute wires between components. One output port of a component can be connected to one or more input ports of other components.
A *Configuration* binds concrete modules to the component instances. These concrete modules are given by their entity and architecture declarations. This bindings can be done with architectures which are either behavioural or structural, i.e. a hierarchy of modules can be defined. Configurations bind component declarations to entity declarations, where the ports must be the same apart from renaming. The renaming of port identifiers are allowed by using *port maps*. The concept of binding component instances at a separate configuration step has the advantage making it possible to fix the structural description before a concrete implementation of the component. In particular, this approach supports a structured handling of proofs in a verification environment by allowing to formulate and handle the correctness of the structural composition.

Within SSL any design hierarchy of finite depth can be expressed. At least for the leaves in the hierarchy tree a behaviour must be provided. For every internal node of a given design hierarchy there may exist a structural as well as a behavioural description.
The formal semantics of SSL is given in [5] in terms of fair synchronous transition systems from [14].

1.1.2. Low-level imperative language SMI

SMI is a programming language that was originally designed in the OFFIS as an interface between the graphical design tool Statemate and finite state machines that are used as an input for a model checker. Now, SMI serves as a generalised intermediate format between high-level design tools and different validation tools. Its main virtue is that it guarantees the necessary independence from different design tools. As such, SMI plays a central role for the integration of several design languages. It closes the gap between user-friendly design environments and sophisticated automatic validation and verification tools. As an example, the translation from Statemate and VHDL into SMI has been realised, whereas the integration with the modelling tool Rhapsody is the topic of the current development in the scope of the Omega project (WP2.1 and WP2.2). SMI allows to develop tools for code generation, testing, and verification, as well as to apply optimisations and abstractions on SMI level on a well-defined basis.
SMI is designed to describe the cyclic behaviour (not the structure) of, for example, an embedded controller. Typically, such systems perform certain steps that have to be represented. To this end, an SMI program describes all feasible steps of a particular system. Each run through this code corresponds to one step of the desired system. A step depends on input values from the environment and on the local state represented by variables. Computations from input and local values yield new output values, as well as a new state, that become visible to the environment at the end of a step. Local variables preserves their values from the end of one step to the next one. References to input values, to old and new values of local variables and to newly computed output values are possible within one step.
This simple cyclic step behaviour of SMI does not fit to all views of embedded systems. As an example, the behaviour of the Omega-models (defined in D 1.1.1) is described using the notion of run-to-completion step (RTC-step) consisting of several internal steps before engaging in a new synchronisation with its environment. The standard choice to represent such sort of behaviour in SMI is to map internal system steps to SMI steps, and to indicate completion of a RTC-step by evaluating a special predicate over the object attributes.
SMI describes the effect of steps using a simple imperative programming notation. As programming statements we have

- Assignment, which can modify local state variables or output variables;
- Deterministic choice: at most one of the alternatives (with Boolean conditions) specified in the statement may be chosen (only one condition may be evaluated to *true*);
- Non-deterministic choice: more than one condition could be evaluated to true, in which case one of the alternatives corresponding to these conditions will be chosen and executed non-deterministically;
- Sequential composition, simply achieved by writing SMI statements in a sequential order;
- Parallel composition, specifying that one or more statements are executed in parallel;
- While-loop

- Break, used to specify the point where the surrounding while-loop can be left immediately without evaluating the corresponding condition.
- Skip, allowing to specify an idle step, leaving the state of the SMI program unchanged.

Within SMI, two views are available on each variable, allowing to distinguish its old value (unprimed) from its new value (primed) inside one step.
Every SMI program is coupled to a *symboltable* containing declarations and definitions for all types, variables and constant used in the SMI program. The used objects are divided into the classes *inputs*, *outputs*, *locals*, and *constants*. Each variable or constant is assigned one of the predefined data types. Furthermore, we associate with every SMI program an interface description, which is consistent to the information within the associated symboltable. Also, it is syntactically compatible with the concepts of interfaces in the language SSL, allowing an integration of SMI programs into SSL specifications. The interfaces of SMI programs contain names of variables which are used for communication with the environment, mode of usage for each variable – input or output – and its data type. Interface descriptions are generated automatically together with the SMI programs and the associated symboltables.
In modelling tools, we can distinguish fine-step semantics and coarse-step semantics (in some tools it corresponds to synchronous and asynchronous semantics). Fine-step corresponds to taking one transition in a statechart, whereas coarse-step corresponds to run-to-completion computation in UML (computation step between two stable state). It is much simpler to represent models under the fine-step semantics within SMI, compared to a representation of models under coarse-step semantics. In the former case, each step of the system corresponds to exactly one time unit, time increases uniformly and environment can influence the state of the system variables (e.g. sending signals to object queue) at every step. In contrast, the RTC-semantics of UML models needs additional bookkeeping to indicate stability, which increases the complexity of the verification and, thus, requires additional abstractions.
The formal semantics of SMI is given in [6] in terms of symbolic transition systems.
The translation of SMI generates functional BDDs (binary decision diagrams) instead of relational ones, because they are more efficient during the generation of finite state machines (FSM) and while model checking. Every compiler produces overhead, resulting in unnecessary instructions and variables. But model checking can get impossible due to this overhead. A set of optimisers for SMI code were developed to cope with this problem with the following tasks:

- Make SMI more deterministic.
- Determine and eliminate unused inputs.
- Perform a data flow analysis and delete unused and constant variables.
- Delete code parts unused due to the values of variables.
- Reduce the SMI program to the set of variables sufficient to verify a given property (computing the “cone of influence”).
- Perform a safe abstraction by freeing variables (turn variables into inputs).

1.2. Tool Architecture

The current version of the verification tool requires the following:

- Rhapsody in C++ versions 4.0, 4.0.1 or 4.1;
- Operation System Windows NT 4.0 or Windows 2000;
- Cygwin 1.3.12-1 or later with additional packages (described in the tutorial).

The current tool translates the Rhapsody representation of UML models into the internal format SSL/SMI. At the both levels – SSL and SMI – optimisations and abstractions of the

model representation are applied, and the existing model checking tools are run (on SMI). To capture requirements, the tool supports a patterns library for the specification of LTL formulas, e.g. allowing to check invariants, drive to a property or drive to a state in a statechart. Simple kinds of LSCs – specified within ASCII-files – can also feed the verification engine. The results are visualised using a graphical tool with symbolic timing diagrams or LSCs.

Since XMI was set as the common format for the model interchange between different tools within the Omega project (M2.2.1), the tool architecture shown on Figure 2 is being implemented, where the translation between XMI and SSL/SMI is necessary to make the model checker available for the users of other UML tools (different from Rhapsody).

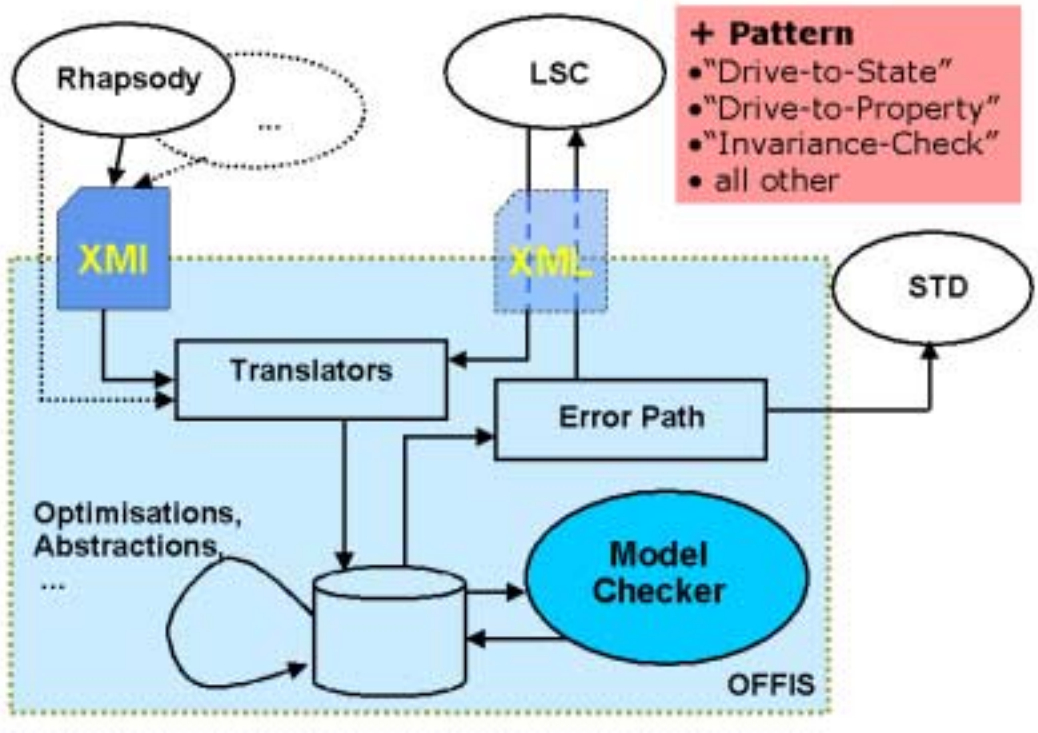


Figure 2. Tool Architecture

We are going to use the LSC editor developed and used within the Omega project for requirement specification. For the exchange between the LSC editor and verification tools, XML is intended to be used. To specify a simpler requirements, the mechanism of patterns can be also used with extended list of the pattern kinds.

2. Formal Verification of UML Statecharts in Rhapsody

At the current stage, the following object-oriented features, which were formally specified in the Omega kernel model (D1.1.2 [1]) are supported by the tool (can be generated into SSL/SMI representation and model-checked):

- Multiple communicating objects of different (reactive) classes;
- Communication between objects via signals (asynchronous type of communication) and operation calls (synchronous type) with parameters;
- Dynamic object creation/destruction (for objects of bounded multiplicity);
- Inheritance and polymorphism with the statechart inheritance restricted by Rhapsody;

- Hierarchical statecharts: AND- , OR-states, connectors (only defined in the Omega kernel model), exit- and entry-actions;
- All kinds of associations defined in the kernel model with bounded multiplicity and dynamic links between objects;
- Actions from the action language of the Omega kernel model (in C++ syntax or via XMI integration).

2.1. Architecture of the OXF Implementation

One major benefit of the OO paradigm is the inherent support for abstraction-centric, reusable, and adaptable design. In particular, it is common to construct complex systems using pre-defined *frameworks*. The Rhapsody fixed predefined framework for code generation is called OXF, where all classes from the C++ framework are prefixed with "OM".
In this subsection we describe the architecture of our implementation of Rhapsody models structured in the following three layers shown on Figure 3:

- **OM-layer** comprises classes taken from the Rhapsody OXF-framework. In our implementation, these classes have only those methods and attributes that are needed for verification. They do not have verification or model dependent methods and attributes. Extensions, e.g. external events, are introduced by overriding virtual methods in derived classes.
- **FW-layer** comprises the framework defining the Kernel Model, inheriting from the OM-layer classes. These classes have all model-independent methods and attributes needed for verification, e.g. `FW Thread` implements the event dispatch loop in method `execute` and has all variables needed by this method as attributes.
- **MD-layer** comprises classes of a particular model:
  (i) specialisation of classes from the FW-layer, e.g. a special event queue that handles external events,
  (ii) the class of the root object that owns in particular an MD-queue and an MD-thread to provide a starting point for crystallisation of one object of each of these classes. *Crystallisation* is a pre-processing which comprises analysis for the required memory space (necessary for the static memory representation within SMI) and binding of possible object's components (under inheritance and composition relation), which reduces the state space exploration during model-checking.

  There is the model that comprises in particular the event classes introduced in the model.

Note that in Figure 3 and in the description we left out the attributes for "memory management", namely \`allocated`\, \`constructed`\, the identity \`this`\, that are present in every class, and the "uplink" that links parts of objects of a derived class and that is only present in derived classes.

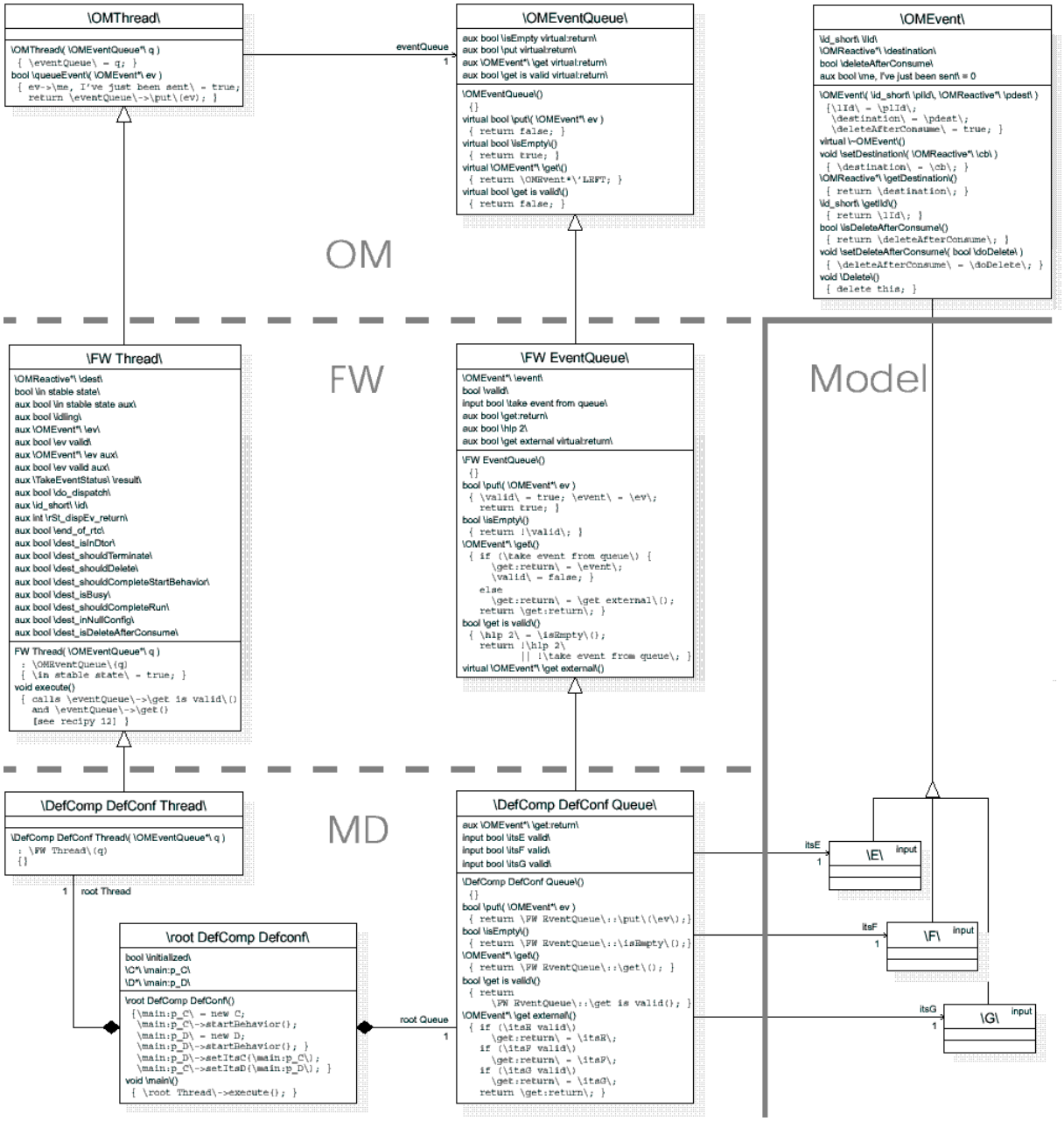


Figure 3. Three-layers tool architecture.

The description of the implicit attributes and operations is given by using C++ syntax (for readability and brevity) where identifiers are shown as SSL identifiers.
Attributes are stereotyped as "`aux`" if they become auxiliary variables in the SMI and as "`input`" if they become inputs in the SMI. (both are also visible in SSL as the attribute MODUS).

1.1.3. The OM-Layer

For our implementation we take into account the following classes from the OXF framework:

`\OMThread\`: From Rhapsody's class `OMThread` we take only the constructor that sets up the link `\eventQueue\` and the queuing method `\queueEvent\` that puts events into `\eventQueue\`. We use the method `\queueEvent\` to set a flag `\me, I've just`

`been sent`\ of \`OMEvent`\ to provide for observability of event sending. Strictly speaking, this functionality belongs in layer FW, i.e. \`queueEvent`\ should be virtual and be overridden in \`FW Thread`\.

\`OMEventQueue` is actually an abstract class that is not meant to be instantiated, because it does not provide the storage for the queue but only the interface. The concrete queue storage and management is implemented in derived classes. The operations \`put`\ and \`get`\ are virtual in Rhapsody's implementation, but \`isEmpty`\ is not. The method \`get is valid`\ is a new virtual method that complements \`get`\ to avoid using `NULL`.

\`OMEvent`\: From Rhapsody's class `OMEvent` we take the attributes \`destination`\, \`lId`\, and \`deleteAfterConsume`\ and the methods to get and set them. As in Rhapsody's implementation, the destructor is declared virtual and there is a method \`Delete`\ to delete any object of a class derived from \`OMEvent`\, e.g. inside the event dispatch loop. The method bodies are transcriptions of Rhapsody's implementation.

To allow observation of event sending we introduce an auxiliary variable that gets its initial value *false* at the beginning of each (SMI-)step and should be set to *true* only in those steps, where the event is sent. According to our implementation, “being sent" means “being visible in the queue for the first time". This new attribute belongs conceptually into layer FW. We didn't introduce a class \`FW Event`\ there, since the attribute would be the only contribution in layer FW and since we would have to introduce the memory management bits \`allocated`\ and \`constructed`\ that would become “state bits" in SMI.

1.1.4. The FW-Layer

\`FW Thread` provides the event-dispatch-loop in the method \`execute`\. Its attributes are in fact local variables of \`execute`\, except for \`dest`\ and \`in stable state`\ which become “state bits". The constructor has to set the initial value of \`in stable state`\ as required by \`execute`\.

\`FW EventQueue`\: In the FW-layer, we want to provide the possibility for different implementations of a queue, e.g. as a ring-buffer, as a bag, etc. and we want to provide the model independent part of handling external events. \`FW EventQueue`\ is one particular implementation of a queue, that manages a queue implemented by an event pointer and a valid flag. Methods \`put`\, \`isEmpty`\, \`get`\, and \`get is valid`\ are correspondingly overridden. Furthermore \`FW EventQueue`\ introduces a new virtual method \`get external`\ that abstracts the “guessing" of external events and has to be overloaded in model specific derived class. Returning an external event is already implemented in the method \`get`\ using the input attribute \`take event from queue`\.

Once a statechart has completed a possible initial run-to-completion step, its object then starts dispatching a new event as follows:

(i) takes event `ev` from a corresponding queue;
(ii) calls `ev->dest->takeEvent`, which in turn
(iii) calls `this->consumeEvent(id)`, where `this=ev->dest` & `id=ev->id`, which in turn
(iv) calls `this->rootState_dispatchEvent(id)`;
(v) if the object is currently in unstable state, then calls `this->rootState_dispatchEvent(NullId)`.

For a triggered operation call, we introduce an event and methods for injecting this event into a queue and dispatching it. Note that there is a separate subclass of \OMEvent\ for every triggered operation in the system.
If two orthogonal states of a statechart accept the same event, they do it at the same RTC-step within this statechart, but the order is defined by the Rhapsody implementation (whereas it is considered non-deterministic in the Kernel Model WP1.1/D1.1). This semantic difference is going to be resolved in the next version of the (XMI-) tool.

1.1.5. The MD-Layer

\DefComp DefConf Thread is the model dependent class in the *Thread* hierarchy. A virtual method \in stable state\ from a higher layer has to be overridden here in a model dependent way since it needs to know the number of OMReactive objects. It can be left in the architecture since most variables can be crystallized such that they don't contribute unnecessary “state bits".

\DefComp DefConf Queue is the model dependent queue class that has the purpose to provide external events by overriding method \get external\. Class \DefComp DefConf Queue\ has a link to an input object of every class of events in the model. Many values in these input objects are in fact constants, e.g. the memory management information, and only \destination\ and \lId\ from \OMEvent\ and possible event parameter attributes are variable. All methods call the methods of the base class.

\root DefComp DefConf is the class of the “root”-object from the Rhapsody framework. It carries the local variables of the main function, i.e. the top level objects, the initialisation code from \main\, that is, everything between OXFInit and OXFStart in the constructor, and a call of the event dispatching method \execute\ in a method \main\. In the SSL representation, the construction of the root object takes into account that the root object is always crystallised, so it uses attribute \initialised\ to test if the initialisation branch has to be taken. It owns an object of class \DefComp DefConf Queue\ which is used to construct the object of class \DefComp DefConf Thread\, which functions as the global default thread.

2.2. Data Flow in the Tool Implementation

Figure 4 gives an abstract view of the data flow of the model generation process and an overview of how to carry out the model generation manually, using the *sllif* tool, a command line interface to the SSLlib library. Although a tool for automatic model generation should use the functions provided by the SSLlib directly, the internal data flow and the set of used libraries remain to be the same.

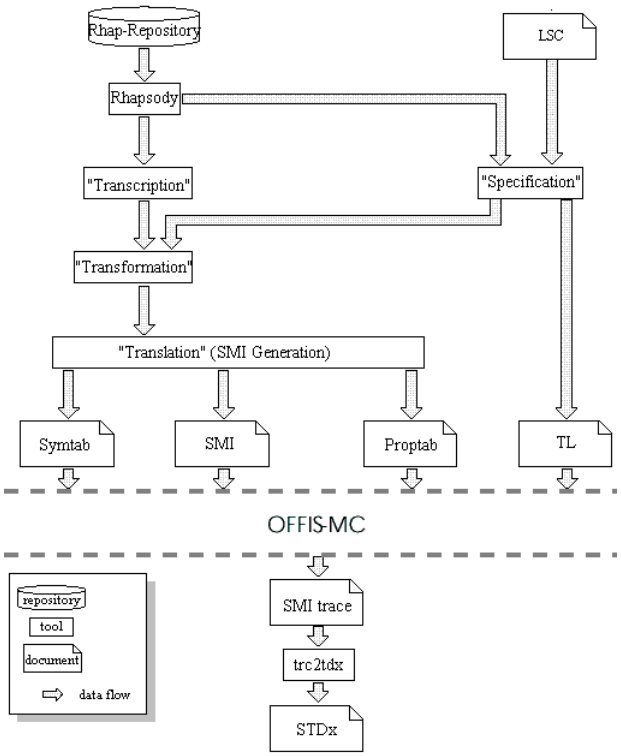


Figure 4. From Rhapsody to SMI

The main phases in the model generation process are the following:

- **Transcription**, that aims at a “one-to-one" conversion from the source language (e.g. C++ or XMI) into SSL, i.e. it keeps all information like inheritance relations, kind of associations, etc. that are contained in the Rhapsody repository, the generated C++ code or XMI.
- **Transformation**, that comprises a number of sub-phases that do the actual transformation from an SSL code which is similar to the high-level language C++ down to an SSL code which is similar to the low-level language SMI.
- **Translation**, or SMI generation, that takes the SSL code, now with a single global scope and no remaining function calls, and translates it into SMI together with the corresponding *symtab*.

The rest of this section is basically a detailed caption for these phases with some examples of OO feature handling.

2.2.1. Transcription

We start our walk through at the Rhapsody-Repository. Figure 5 gives a refined description of the first phase in the model generation, called *transcription*. The Rhapsody-Repository is Rhapsody's internal representation of the model. The tool *rhap2ssl* uses the COM API of Rhapsody to navigate the repository and extracts the following:

- type definitions,
- classes,
- inheritance relations,
- associations,
- attributes,
- signatures and bodies of primitive operations,

- triggered operations,
- events.

but not the implementation of statecharts. The output of *rhap2ssl* is SSL source code.

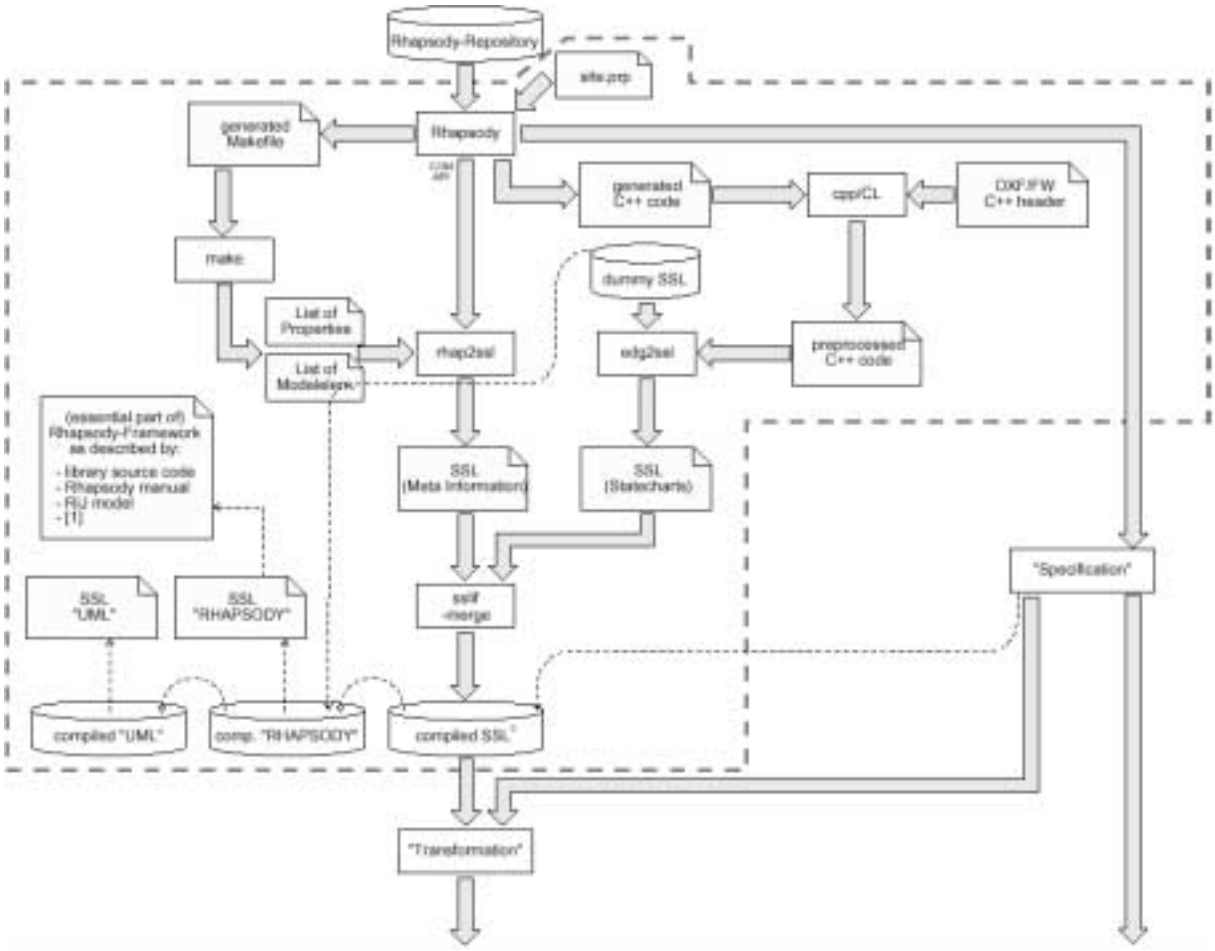


Figure 5. From Rhapsody to SMI – transcription

The SSL code generated by *edg2ssl* is different to the SSL generated by *rhap2ssl* in two ways:

(a) it does not contain information about associations, since associations are shown up as plain attributes in the C++ code, but
(b) it does contain the implementation of statecharts.

The SSL texts with and without statechart information are merged to a single repository employing the *sslif* tool. This yields a first version of compiled SSL or "compiled SSL^1" that is the starting point for an iterative process (within the transformation phase) that finally leads to a "compiled SSL^N" very similar to SMI.

The precompiled SSL libraries "*UML*" and "*RHAPSODY*" provide type definitions and signatures that are used by the design. The SSL library "*UML*" provides means to encode the kind of model elements like associations, classes, events, etc. utilising SSL attributes and names from the UML metamodel. The SSL library "*RHAPSODY*" contains classes used by the Rhapsody framework like `OMThread`, `OMReactive`, and `OMEvent` and signatures of C++ "built-in" functions like `new` and `delete`.

Both SSL libraries are compiled from SSL source code that is based on information from the implementation of the Rhapsody framework and additional documentation.

Example: object creation and destruction.
The call of operation "`new C(id)`" does a number of things:

(i) it allocates memory (enough such that an object of class C with all its inherited attributes fits in),
(ii) it calls the appropriate constructor to initialise the newly allocated memory, and
(iii) it returns a pointer to the newly allocated, initialised memory.

The call of operation "`delete(cp)`" (for `cp` pointing to an allocated object of class C) does the following:

(i) it calls the appropriate destructor (that operates on a fully functional object), and
(ii) it frees the memory.

For the SSL model generation procedure we assume that the operators `new` and `delete` are not overloaded for any class in the model.
The handling of the *new*-operator in SSL begins already in the transcription where we have to translate C++ expressions like "`cp = new C( p1, ..., pn )`". At the first phase of the model generation process, this expression will be simply transcribed to "`\cp\ = \C\\::\\operator new\( p1, ..., pn )`" with a declared function `\operator new\` in global scope.
The SSL implementation is very similar to the C version, but we provide a second version of every operator `new` of a class that allows to implement crystallisation. At the next phase, transformation, the `malloc` function will be "classified" for every class. The "interface" to our "memory management" for the operator `delete(cp)` is the function `free` that has to be "classified" for every class just like the mentioned function `malloc`.
Note that in general we also have to call the "allocating" version of a superclass' `malloc`. If analysis finds that the link is crystallisable or if the relations between object parts are crystallised by decision, then this call can be replaced by a call of the "crystallised" version.

2.2.2. Transformation

This phase of the model generation process iteratively implements a pre-compilation from a UML model (represented in SSL) into the kernel model, which is still represented in SSL, but contains very simple constructs, and thus is similar to SMI. This step comprises the following sub-phases:

- Establishing of structural relations, that introduces means to make the model elaboration configurable in number of objects and to allow crystallisation;
- Normalisation, that converts the "address view" into an "array view", i.e. method calls of the form "`p->f`" are translated into "`f(p)`", and normalised nested function calls, to prepare later inlining;
- Shift in packages, that basically transforms existing local scopes (one `ENTITY` per class) into a global scope (`PACKAGE`);
- Model elaboration, takes a configuration and computes the memory model, i.e. it basically determines how much "storage" the memory model has to provide to hold the model determined by the configuration;
- Inlining and simplification, that flattens existing function calls and simplifies expressions that can be computed at compile time.

If the transformation is carried manually, it consists of a number of iterations in the form

(i) apply an *sslif* command to the repository "*compiled SSL*^{i}",
(ii) get a representation "*SSL*^{i}" of the result, and
(iii) recompile the "*SSL*^{i}" into a repository "*compiled SSL*$^{i+1}$".

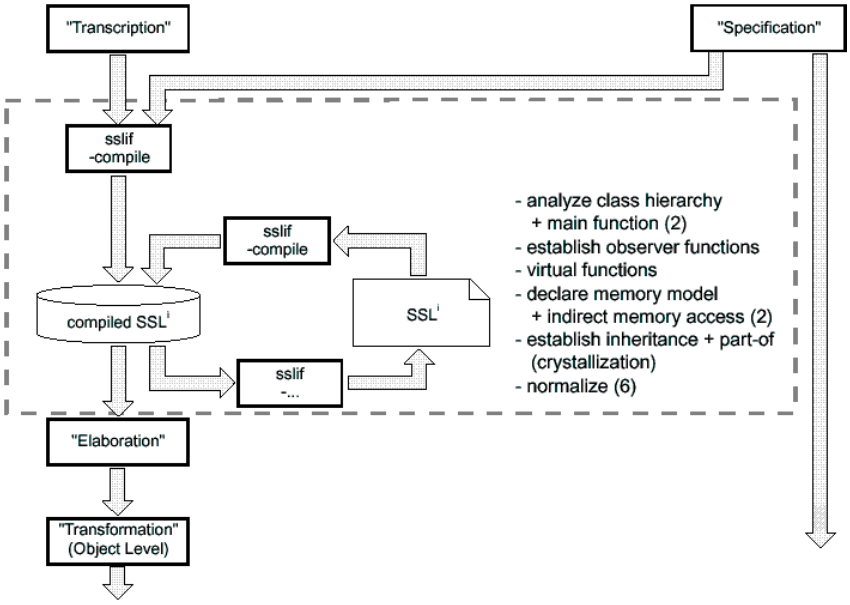


Figure 6. Transformation on the class level

Example: Memory Model. During the transformation phase, the UML model information will be analysed class by class and transformed into a set of parameterisable and generic memory model schemes. Each memory model scheme consists of an entity declaration, an architecture declaration and a configuration declaration. This schemes will be further used during the elaboration of a concrete memory model. The focus of the analysis of the classes lies on the two structural relation kinds – generalisation and strong aggregation (composition). If a class has no such relations it is called simple, otherwise structural.
Classes without any relation of kind generalisation can be viewed as if they are root classes of a class hierarchy. The hierarchy may only include the root class. During the creation of an object of such a root class there is no need to create inherited parts of a more general aspect of the object. The consequence for the declaration of the memory model is, that there is no need to declare structural parts to be able to deal with inherited state space or inherited behaviour. If such a class don't have any strong aggregation relation too, then there is also no need to build a structural part that represents the "parts-of" relation of that class. The memory model scheme for such a class, called simple, consists of the entity declaration that represents the class in SSL, a new created architecture declaration and a new created configuration declaration. Then the architecture declaration is empty. The architecture will be transformed during the normalisation phase. The created configuration does not configure anything inside of the architecture and/or entity, but serves as a reference point for other configurations. The configuration also determines the address width of the configured substructure.
The consequence from the definition of the composition relation for the declaration of the memory model schemes is that for each class with this (“part-of”) relations there is a need to declare a second scheme, beside the behavioural scheme, for the creation of all of the parts of the object. Thus the result of the transformation of a class with composition relations are two memory model schemes. One scheme is the behavioural scheme and the other is called structural and describes which parts and how many of them should be created beside the behavioural part of the composite class. The structural scheme consists of an entity, an architecture and a configuration. The architecture declares as many component declarations as composition relations, plus one component declaration for the behavioural part of the

composite class. Furthermore for each such a component declaration there is a component instantiation statement.
The configuration consists of as many configuration items as component instantiation statements and generate statements in the architecture. Each configuration item that configures a component instantiation statement binds a configuration declaration to that statement. The meaning of such a binding is that during the elaboration of the model each such component instantiation statement will be replaced by the result of the elaboration of the bounded configuration. Beside this binding there is also a mapping of the actual address value to the formal generic \`this`\ of the "part-of" entity.
The possibility to parameterise which objects should be represented in the concrete memory model has an impact on the mechanism that computes the address for an object.
If analysis finds, that a relation between a subclass-part and its superclass-part is fixed, then it sets the calls to the crystallised version of \`malloc C`\ with the (crystallised) *uplink* as parameter. This has to be established on a by-class basis, i.e. the relations between some parts of an object may be crystallised while the relations between some other parts is dynamic, for example since only the top part is shared between multiple objects.

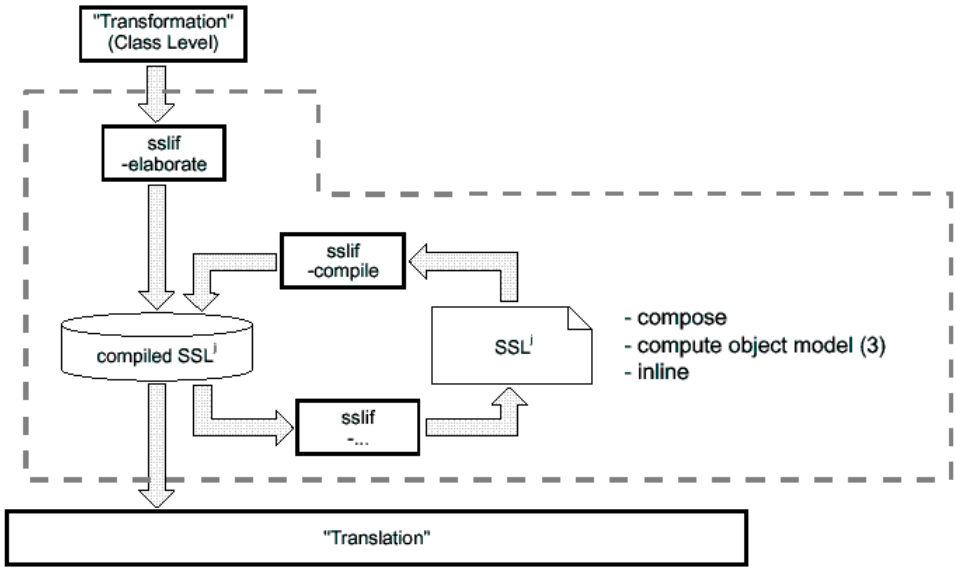
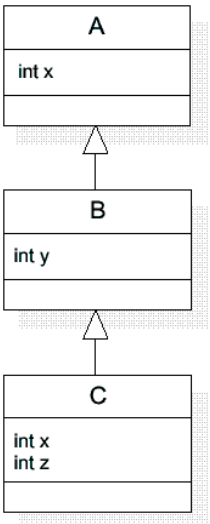


Figure 7. Elaboration and transformation on the object level

Inheritance in the object memory model. A naive SSL/SMI representation of the class hierarchy of the example from the right-hand figure could introduce records similar to C constructs. Then the object arrays for a system with one A, B, and C would be declared like (in C):

```
A   As[1];
B   Bs[1];
C   Cs[1];
```

Recall that attribute access has to be rewritten from "`ap->x = 0`" to a construct like "`As[void_pointer_to_A_index(ap)].x = 0`". Now, if `ap` actually points to an object of class B, we have to write "`Bs[void_pointer_to_B_index(ap)].x = 0`" to access array `Bs` instead of `As`, i.e. one would have to dynamically determine the type of the object the pointer actually points to and access the right array.



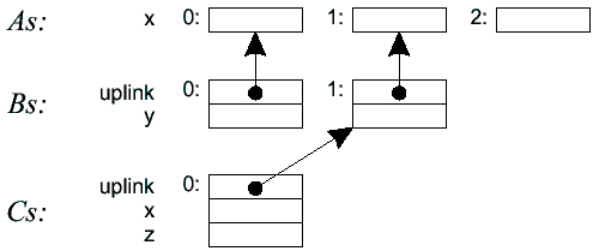


Figure 8. A fragment of a memory state

We need one A-part for an object of A, another A-part and a B-part for an object of B, and one of each part for an object of C. The operation for the objects creation has to be defined so that it allocates all parts needed for the particular class and that it initialises the *uplink* pointers. In a run of the example system, the memory may look like depicted in Figure 8 after all three objects have been created.
Note that the relation between the "object fragments" are not a priori fixed, the implementation may choose to set up the links in Figure 8 differently if we first destroy all three objects and then create them again. But we explicitly make the uplink relations fixed as part of the crystallisation.
Now to access attributes of objects we have to employ a more complicated translation in the normalisation that builds an expression using the *uplink*.
For example, an expression "`A::x = 0;`" inside a method of class `C` finally becomes "`As[Bs[Cs[void_pointer_to_C_index(ap)].uplink].uplink].x = 0`".
Such implementation can ease LSC verification, where an instance labelled with class A means "for all objects of class A and of all derived classes ..."

2.2.3. Translation

Finally, the model generation process ends up with an SSL code that is very close to an equivalent SMI representation, i.e. that has a single global scope and no function calls. The commands *sslif* from *ssl2smi* and *ssl2symtab* employ the library *libssl2smi* to translate this SSL code into SMI and its associated *symtab* (see Figure 9). The translation basically consists of a simple translation from the SSL statement and expression language to SMI and encoding SSL names into SMI names.

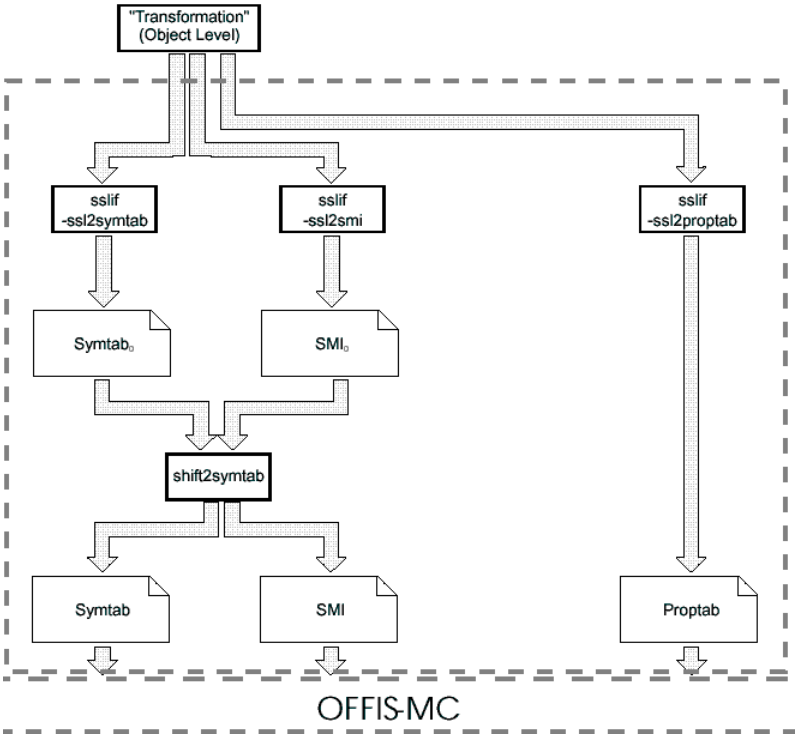


Figure 9. Translation phase: from SSL to SMI

2.2.4. Specification

Sending and reception of events , method calls, and conditions referenced in LSCs have to be interpreted and observed in the SSL/SMI model. The observation in the SSL implementation comprises, on the one hand, an “infrastructure” of regularly named functions established in the SSL model and, on the other hand, a specification of the exact meaning of sending and reception of events, method calls, and conditions in terms of variables in the SSL implementation that provides the bodies of these functions.

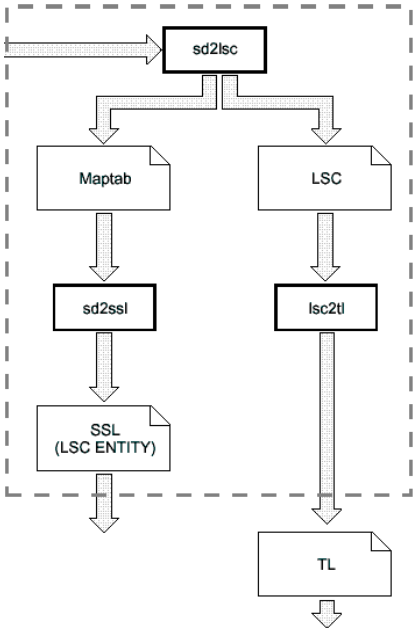


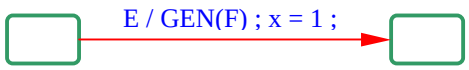
Figure 10. From Rhapsody to SMI – specification

The basic idea in the SSL specifications is to have the following:

- For every event (class) `E` two Boolean SSL functions – `\ES E\(SENDER, RECEIVER, P1,…,Pn)` and `\ER E\(SENDER, RECEIVER, P1,…,Pn)` – that yield *true* iff an event of class `E` (or event instance) has been sent or received, respectively, from `SENDER` to `RECEIVER` with the corresponding values of the event parameters `P1,…,Pn`.
- A tool *sd2ssl* (see Figure 10) that takes *Maptab* (that talks about properties `prop1, … , propn`) and generates an SSL `ENTITY` of metaclass LSC that has attributes `prop1, ..., propn` that are set to
  - `\ES E\( ... )` if `propi` denotes sending of an event `E`,
  - `\ER E\( ... )` if `propi` denotes receiving of an event `E`,
  - `\MC C::f\( ... )` if `propi` denotes calling of a method `C::f`, and

  conditions are simply translated from C++ to SSL according to the *Maptab*.

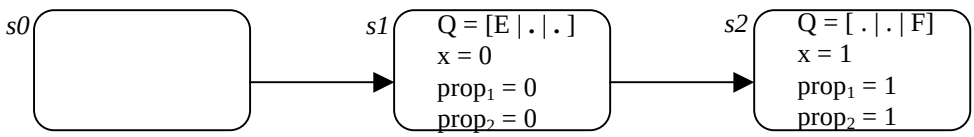
Event sending. The concept of the asynchronous event sending and reception from Rhapsody has been re-built in the SSL model, where even sending consists of calling the insert-method of the receiver's queue, thus implementing event communication via synchronous calls.
We decided to consider an event sent, if it is inserted in the queue of the receiver. An alternative is to observe the call of the `GEN()` function. They are equivalent when `GEN()` and the insertion into the queue are executed in a single step, e.g. when the sender and the receiver belong to the same thread (called activity group in the Omega kernel model).
In Rhapsody, events don't carry links to their senders, so to observe the `SENDER` we would need to introduce another attribute. The type of the attribute could be the address of the most specialised part or of the part of a common class "Object", because any object can send events, not only reactive. The common "Object" would introduce multiple inheritance, so we would go for the most specialised part.
Event reception. We decided to consider an event received, if it is taken out of the queue and dispatched to the receiver (where it might be ignored when no transition is activated). An alternative is to observe the insertion into the queue of the observer and introduce a third name that denotes dispatching. Note, that two alternatives are equivalent when the sender and the receiver belong to the same thread. We also consider an event received when it is ignored by the `RECEIVER` since there is no transition enabled with such trigger.
Example. Consider a statechart containing the following transition



When receiving an event `E`, the object sends an event `F` to itself and sets a value to its attribute. Assume that we have installed the observers

`prop1 = \ER E\(...),`
`prop2 = \ES F\(...),`

and that `E` is the first event in the queue and "`x = 0`". Then we will see the following transition in the FSM (finite state machine) of the model checker:



To get from the system state *s1* to *s2*, we take `E` from the queue, dispatch it, and have the corresponding action executed. Thus the effect of the action is visible in the next step and so is $prop_1$. Furthermore, `F` is first visible in the queue in the system state *s2* and so is $prop_2$.
Conditions. We currently consider conditions in LSCs as C++ Boolean expressions without side-effect that talk about object attributes where the objects are absolutely navigated, i.e. relative to the *root*, e.g. "`root->p_C->a == 0`", but not "`a == 0`".
The conditions are translated to SSL expressions that are assigned to properties $prop_i$, e.g. `prop1 := \root\->\p_C\->\a\ = 0`.
A property is true in a state *s* iff the starting values of variables in *s* (those have not been yet modified at the corresponding SMI-step) fulfil the property. For example, if we have the statement "`a = 0, a = 1, a = 2, a = 3`" as the action of the last transition and attribute `a` is not modified elsewhere, then the property `prop1` defined above will not be true at this transition.
Quantification in LSCs. The semantics of a universal LSC `L` with an instance line named `n` without navigation expression is defined by the quantification as follows:

```
M |= L <==> forall c in C_IDs : M |= L[n/c]
```

i.e. all occurrences of name `n` are substituted by object identifiers `c`.
An existential LSC is read as "there exists a binding of objects to instance lines such that these objects adhere to the specification".
The navigation expression is integrated into the activation condition such that the substitution also applies to occurrences of free variables in navigation expressions. For instance, lines which does not represent instances created during the run of the LSC, i.e. instance line beginning right at the top of the LSC, there is the additional condition that object `c` has to be alive initially. If there are multiple instance lines of type `C` which are initially alive, there is the additional condition that the instance lines are bound to different objects in the system.
Object creation and destruction is handled by extending the corresponding instance lines to the top and bottom of the LSC and transforming the creation into the observation of a special message with the condition that the object should not be alive before and the destruction to the condition that the object is not alive afterwards.
Within the model-generation, the activation condition (which has a name from the *maptab*) is defined such that the navigation expressions and aliveness requirements are respected. Thus the input is still a symbolic LSC, the LSC itself is not expanded and unfolded.
At this point there is a "verification driver" which selects -- one after another -- one of the observer objects, starts a verification task, and keeps track of the result. If all results are true, the overall result is true (universal case).
Optimisation. There are several possibilities to optimise the verification of LSCs already at this stage of the generation process (specification translation and analysis).
(i) *Smart verifier*. The "verification driver" can derive from the activation condition that some combinations need not be tried since the activation condition is always false. For example, if there are two initially alive lines of the same class then the activation condition requires that they are not bound to the same object. Thus combinations which bind the same instance to different lines are trivially true and need not be started.
(ii) *Symmetry reduction*. The idea of query reduction is not to verify all the observer objects but only a representative subset. If the types the quantifiers range over a scalar-sets (symmetric) – which is typically the case for object IDs and link array indices – then the example of a `C` instance and a `D` instance reduces to a single verification task of one of the observer objects. It has to be verified that the types of IDs are actually scalar-sets. This can be

done syntactically, or more elegant by using SSL's concept of types (a priori there are no operations, in particular no conversions, defined for a new type, so a scalar-set type could be introduced which does not get operations like "+").
(iii) *Variable freeing* is a kind of abstraction, where if one concentrates on a particular index *i* in a specification, then all other indices can be subsumed in a special index *NaN* ("Not-a-Number") which represents "different from *i*". If an LSC has two instance lines of types `C` and `D`, then all class identifiers except for `C` and `D` can be reduced to *NaN* and the class identifiers for `C` and for `D` each have one regular and one *NaN* index. It remains to be verified on the SSL side the "scalar-set" property, the possibility to modify ranges, and the extension of the read/write functions to respect the special value *NaN*.

2.2.5. Visualising results.

One of the essential concepts for verification is visualisation of the model-checker's error paths in a rather non-technical style. Currently our tool supports two ways: displaying paths in form of multiple waveforms with STD-tool, and showing event communication via LSCs. A third possibility – which is not yet implemented – would be to drive the Rhapsody model via the animation API with respect to the information of the LSC.
Timing Diagrams. Symbolic timing diagrams (STD) are already used in our verification environment to specify requirements and to visualise graphically the verified property. STDs are a user-friendly notation with a well-defined formal semantics given by temporal logic.
The tool chain for the generation of readable timing diagrams contains the following:

- Filtering of uninteresting parts of the trace (like framework objects or *uplink* attributes) and flattening the trace so that each object attribute becomes an own waveform. It also back-substitutes the symbolic values of variables of enumeration type and the values *true* and *false* for Boolean variables (instead of showing "1"and "0") inside the waveform.
- Conversion of the SMI-trace format to the format handled by the STD tool. Also at this step, the "basic state" waveforms of each object are collapsed into one "statechart" waveform per object.
- Conversion of SMI-names which occurs inside the waveforms to Rhapsody names. The waveforms names itself must keep their SMI-encoding in order to be accepted by the error tracing tool.

In addition to these pre-processing steps already presented in the verification environment, we extend the timing diagram viewer by a C-function which converts the SMI-encoded waveform names back to Rhapsody names. This function is executed whenever waveform names are displayed to the user. Two further functions enable users to dynamically and incrementally filter and search for interesting trace elements.
The timing diagram view is quite good for checking the state of statecharts and values of attributes in different (SMI-)steps, but is hard to exactly figure out the event communication of the model.
LSC Generation. The LSC displays the events (signal and call events), communication in the model and the reception of events from the outside. We use the framework information for determining what is happening at each step. More specifically, we discover:

- *event sending*: Looks at the corresponding attribute of objects (derived) from `OMEvent`, which indicates that someone has just sent an event of the corresponding type. The destination, a pointer to an object derived from `OMReactive`, can be found in the \`destination`\ parameter, the sender is stored in an observer.

- *event receiving*: An observer is set appropriately in the event dispatch-loop. We can lookup this `OMEvent` pointer in a tool-internal list of pending messages and determine the corresponding sender and destination.
- *method calls*: SSL observer functions are needed which detect the beginning and the return of each method call and which log the information about caller, callee and parameters into a global stack. The size of this stack has to be determined in advance for the specific model. The stack would provide sufficient information to discover the call chain at each step.

To conclude from the `OMEvent` and `OMReactive` pointers to the real objects addresses, we use the *memory model tables* like “most specialised” and “void” to index, what can be found in the symbol table.

3. Integration with XMI

A preliminary version of the XMI-integration for the model checking tool is now available. The current state of the verification tool handles a restricted subclass of the Omega kernel models represented in XMI format. Instances of the following UML meta-classes can be translated from XMI to the internal model-checker format SSL/SMI:

- Model_Management.Model: There should be exactly one named instance, as child of XMI.content.
- Model_Management.Package: There should be exactly two packages, as ‘ownedElement’ of the Model. One Package named “PredefinedTypes” has to contain all basic datatypes (like int, bool, char), which are used in the model (They are instances of the UML meta-class ‘Foundation.Core.Datatype’). The other package, named by the user, contains the rest of the model.
- Foundation.Core.Class: (Named) classes as ‘ownedElement’ of the user-defined Package are supported (but for example no classes in classes). Exactly one class should have a tag ‘CPP_CG.Class.isRootClass’ set to ‘True’.
- Foundation.Core.Association: The user-defined package may contain (as ‘ownedElement’) Associations between classes. The ‘connection’ of each Association has to consist of exactly two AssociationEnd-instances.
- Foundation.Core.AssociationEnd: ‘isNavigable’ may be ‘true’ or ‘false’ – the AssociationEnd should be named, if ‘isNavigable’ is ‘true’. The ‘aggregation’-kind may be ‘none’ or ‘composite’.
- Foundation.Data_Types.Multiplicity: The ‘multiplicity’ of an AssociationEnd should consist of exactly one MultiplicityRange.
- Foundation.Data_Types.MultiplicityRange: The ‘lower’- and ‘upper’-bound should be set to 1.
- Foundation.Core.Attribute: A named attribute can be a ‘feature’ of a class. All kinds of ‘visibility’ (public, private, protected) and the ‘ownerScope’-value ‘instance’ are supported. The ‘type’ of an Attribute has to be a basic type defined in the “PredefinedTypes” package.
- Foundation.Core.Operation: A named operation can be a ‘feature’ of a class. All kinds of ‘visibility’, all kinds of ‘ownerScope’ and the ‘concurrency’-value ‘sequential’ are supported.

- Foundation.Core.Method: For each operation there should be exactly one defining method, which is also a ‘feature’ of the corresponding class. The definition of the method is located in the ‘body’-meta attribute.
- Constructors and the destructor of a class may be modeled as Operations resp. Methods. The have to be named like the corresponding class (for constructors) or like the corresponding class prefixed with ‘~’ (for destructors).
- Foundation.Core.Parameter: Operations and the corresponding methods may have parameters of kind ‘in’ and should have at least one parameter of kind ‘return’. The type of a Parameter has to be a basic type, defined in the “PredefinedTypes” package.
- Behavioral_Elements.State_Machines.StateMachine: Each class may have at most one Statechart as ‘behavior’. The ‘top’-state of a StateMachine has to be a CompositeState. A StateVertex may be a PseudoState of kind ‘initial’, a CompositeState (‘isConcurrent’ may be ‘true’ or ‘false’) or a SimpleState.
- Behavioral_Elements.State_Machines.Transition: Transitions may be ‘incoming’ and ‘outgoing’ of state vertices. Each transition may have a at most one ‘guard’ (of meta-class “Foundation.Data_Types.BooleanExpression”), a ‘trigger’ (of meta-class “Behavioral_Elements.State_Machines.SignalEvent”) and an ‘effect’ (of meta-class “Behavioral_Elements.Common_Behavior.UninterpretedAction”).
- Behavioral_Elements.State_Machines.SignalEvent: The ‘signal’ of a SignalEvent should be named.
- Behavioral_Elements.Common_Behavior.UninterpretedAction: The ‘script’ of an UninterpretedAction contains the ‘body’ of the action
- The action language in method bodies, transition effects and transition guards may be either the Rhapsody action language (subset of C++ with library functions corresponding to the actions defined informally in D1.1.2), or the Omega Action Language defined in M2.2.1.

4. Concluding Remarks

The current version of the tool resolves restrictions from the previous version, such that supporting triggered operations, complete inheritance, parameters for events, higher multiplicity for composition association, higher length of event queues, pseudo-states in a hierarchical statechart and some other features of UML models as well as more complicated properties – general kinds of patterns and restricted subset of LSCs.

References

[1] W. Damm, B. Josko, A. Votintseva, A. Pnuelli, *A Formal Semantics for a UML Kernel Language*, Deliverable WP1.1/D1.1.2 of the Omega project, January 2003, available at http://www-omega.imag.fr/doc/d1000009_6/D112_KL.pdf .

[2] W. Damm, D. Harel, LSCs: Breathing Life into Message Sequence Charts, *Formal Methods in System Design*, 19(1), pp.121-141, 2001.

[3] Object Management Group *Unified Modelling Language Specification (UML),* v 1.5, March 2003, http://www.omg.org/cgi-bin/doc?formal/03-03-01 .

[4] Object Management Group. *UML with Action Semantics, Final Adopted Specification, ptc/02-01-09*, January, 2002, available from http://www.kc.com/as_site/home.html .

[5] U. Brockmeyer, H. Hungar, SSL – System Specification Language, Technical Report, Kuratorium OFFIS e.v., Oldenburg, 1999.

[6] J. Bohn, C. Essmann, U. Brockmeyer, H. Hungar, *SMI-System Modelling Interface*, Technical Report, Kuratorium OFFIS e.v., Oldenburg, 1999.

[7] R. Schloer, W. Damm, Specification and Verification of System-Level Hardware Design Using Timing Diagrams, in: *Proc. of the European Conference on Design Automation with the European Event in ASIC Design*, 1993.

[8] http://www.ilogix.com/products/rhapsody/index.cfm

[9] http://www.ilogix.com/products/magnum/index.cfm

[10] http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/

[11] http://www.mathworks.com/products/simulink/

[12] IEEE, IEEE Standard VHDL Language Reference Manual, Std 1076-1993, 1993.

[13] Unified Modeling Language 2.0 Proposal, v. 0.69 (draft), OMG doc # ad/02-04-05), March 15, 2002, http://www.omg.org/docs/ad/02-04-05.pdf .

[14] Y. Kesten, A. Pnueli, *An αSTS-Based Common Semantics for* `Signal` *and* `Statechart`, Weizmann Institute, Israel, 1996.

[15] IST-1999-10069, http://wooddes.intranet.gr .

[16] I. Ober, Action Specification in Omega, part of M2.2.1 Omega milestone, March 2003, http://www-omega.imag.fr/doc/d1000092_3/ActionsRmars.pdf .