OMT*, Bridging the Gap between Analysis and Design

*V. Jonckers, K. Verschaeve, B. Wydaeghe, L. Cuyppers, J. Heirbaut
Vrije Universiteit Brussel*

*Laboratory for System and Software Engineering, Vrije Universiteit
Brussel, Pleinlaan 2, 1050 Brussel, Belgium. Telephone: (32)
2-6292974. Fax: (32) 2-6292870. email: vejoncke@info.vub.ac.be*

Abstract

This paper presents OMT*, a formal variant of OMT (Object Modeling Technique) designed to bridge the gap between analysis and design. OMT* is a partial result from a larger research effort proposing an integrated methodology and toolset based on the combination of Object-Oriented and Formal-Description Techniques. In this project OMT is used as the systems requirements analysis technique while SDL (Specification and Description Language) is targeted for the design phase. OMT* is defined by its abstract syntax, static semantics and transformational semantic, i.e. a set of transformation rules mapping OMT* constructs to SDL constructs. OMT* is more strict than full OMT, the possible interpretations of OMT constructs are reduced and the relations between the different OMT models are formalised. The translation from OMT* to SDL preserves the logical structure of the specification.

Keywords

System Design, Integrated Methodology, OMT, SDL, Formal Definition, Abstract Syntax, Static Semantics, Code Generation, Transformational Semantics

1 INTRODUCTION

The INSYDE (INtegrated methods for evolving SYstem DEsign), (Insyde, technical annex 1994) methodology is a set of techniques and tools to enable the evolving co-design of hybrid systems from requirements analysis to implementation and validation. A hybrid system is one which contains significant hardware and software components. As the complexity of such systems is constantly increasing, the development of large systems requires a consistent and integrated methodology for proceeding from analysis to implementation. The INSYDE project will produce a prototype methodology and toolset based on the combination of Object-Oriented and Formal-Description Techniques and covers the full development lifecycle from Systems Requirements Analysis to Implementation and Validation in a unique and integrated way.

The INSYDE project is an EU ESPRIT III funded project. The consortium consists of Alcatel Bell Telephone (Belgium), Dublin City University (Ireland), Humbolt Universität zu Berlin (Germany), Intracom S.A. (Greece), Verilog S.A. (France) and Vrije Universiteit Brussel (Belgium).

The INSYDE methodology integrates the object-oriented analysis methodology OMT (Object Modeling Tool, (Rumbaugh, 1991) with two domain specific formal description techniques, namely SDL '88 (CCITT, 1988) and VHDL (Navabi, 1993). OMT is used as the system requirements analysis technique and also as the technique for the initial design stages. This allows the methodology to provide mechanisms for combining the individual design techniques (OMT, SDL, VHDL), maintaining the consistency of partial models at the detailed design stage and co-simulating the formal description to validate the hybrid system against the system specification. The relative strengths of each design technique (SDL for asynchronous communication systems, VHDL for synchronous reactive systems) can thus be exploited in an optimal way.

Since we work in the field of communication systems we limit our scope to the transition of OMT to SDL. This transition is important and interesting because there is a strong need for an automatic reuse of analysis information into software specification languages like SDL. Since communication systems tend to have large dynamic models and nearly no functional models, we do not use the functional model of *OMT*.

While OMT is a good analysis methodology covering many aspects of system design, the informal nature of OMT makes an automatic translation infeasible. In our methodology the analysis document is prepared for translation during system design. During this phase subsystems are identified, communication is formalised and information is ordered. To describe these aspects a new language is needed.

We present *OMT**, a dialect of *OMT*, aimed to meet the requirements of system design. *OMT** differs from *OMT* in that:

1. it contains only those *OMT* concepts suited for system design (e.g. no ternary associations, overlapping subclasses,...),
2. the possible interpretations of an *OMT* construct are reduced and clearly described,
3. it has a well defined syntax,
4. there are clear relationships between the different models.

In order to make the transition from *OMT* to *OMT** as smooth as possible, *OMT** contains as much as possible of *OMT* and has a semantics that is as close as possible to OMT. Therefore, the semantics of *OMT** can not be defined unambiguously, i.e. the possible interpretations of an *OMT** construct is reduced with regard to *OMT* but not to one single interpretation. Furthermore *OMT** does not contain any new constructs and is completely compatible with Rumbaugh (Rumbaugh, 1991).

Furthermore, the transformation of *OMT** to SDL will be assured by giving the language a *transformational semantics*, this is a semantics defined by specifying the transformations of a construct to SDL. For those constructs having more than one interpretation, multiple transformations will be specified.

1.1 Structure of the paper

OMT* is defined in three sections describing its abstract syntax, static semantics and transformational semantics into SDL'88.

- Chapter two is devoted to the abstract syntax of OMT*. The abstract syntax defines exactly which constructs are allowed and which are not. It also defines the syntactical relationship between the object model and the dynamic model.
- Chapter three describes the static semantics of OMT*. This is done by defining semantic rules and a scope system.
- Chapter four defines the translation rules to SDL'88. In this way, the dynamic semantics of OMT* are defined from the dynamic semantics of SDL'88, and the translation rules.

2 ABSTRACT SYNTAX OF OMT*

This section introduces the abstract syntax of OMT*. We made a set of guidelines about how to make the transition from OMT towards OMT*. This set is not included because of space limits but it can be found in (Holz, 1994). The abstract syntax defines how the syntactical concepts, such as class, association, ... are defined in terms of others, see figure 1.

Figure 1 Structure of Object and Dynamic Model in OMT*

Note that the dynamic model is accessed through the object diagram. The classes in the object diagram contain pointers to the different state diagrams in the dynamic models. A state diagram describes the dynamic aspects of exactly one class.

2.1 Object model

An OMT* specification is entered through the object model. The object model always consists of exactly one object module. (We plan to allow more modules in the future). An OMT* model contains a list of classes and association definitions.

$\begin{aligned} \langle \text{omt model} \rangle &== \langle \text{model name} \rangle \\ &\quad \{ \langle \text{class} \rangle \}^* \\ &\quad \{ \langle \text{association} \rangle \}^* \end{aligned}$
--

The syntax of *OMT** contains a number of restrictions as opposed to *OMT*, because some constructs in *OMT* have ambiguous semantics or are very difficult to translate into SDL. More specifically, the object model is restricted to object diagrams that

- do not contain multiple inheritance
- contain only binary associations,
- do not contain general constraint expressions,
- do not contain discriminator or restrictor rules.

Class definitions. A class definition contains, as in *OMT*, a class name, attributes and operations. In *OMT** however, it also contains the superclass names, component names (class names at the “bottom” side of an aggregation) and a state diagram. This stresses the asymmetric aspect of aggregation and superclasses.

$\begin{aligned} \langle \text{class} \rangle &== \langle \text{class name} \rangle \\ &\quad \{ \langle \text{attribute} \rangle \}^* \\ &\quad \{ \langle \text{operation} \rangle \}^* \\ &\quad [\langle \text{super class name} \rangle] \\ &\quad \langle \text{aggregate} \rangle^* \\ &\quad [\langle \text{state diagram} \rangle] \end{aligned}$
--

With every active class definition, we will associate exactly one state diagram. A state diagram is used to describe the effects of an event on the state of an object and to describe how and when actions are executed (called). Classes that have a state diagram are called active classes. Classes without a state diagram and that have no active aggregate are passive.

Attributes and operations. An attribute has a name, a type and, optionally, a default value. In *OMT** types are only names, it is thus impossible to check whether a value is of a certain type or not. Future versions of *OMT** will contain a data part to allow type handling.

An operation is either an input event or a function. Apart from the fact that an operation can have a result type, input events and functions differ in the following:

- An input event is used to initiate a state transition within the state diagram of its class. It cannot be used to change or retrieve the contents of an attribute. It is our intention that an input event can be described within the dynamic model. The parameters passed to an input event can then be used by passing them to a function activation.
- A function is used to do some calculations on the given parameters and on the attributes of an object. As a result, a value can be returned to the “caller” and the values of some attributes of the function’s object can be changed.

Aggregations and associations. Unlike in *OMT*, in *OMT** aggregations and associations are described differently. This is because the semantic differences are strong enough to separate those two concepts. Aggregations are used to model the “part-whole” relationships within the real world or to model subsystem relations. Because communication is the backbone of SDL, associations are translated as channels. This implies that associations cannot be used for expressing any other relationships than communication. Therefore in making the transition from *OMT* to *OMT** (i.e. from analysis towards system design) it is important to adapt the model towards these new semantics.

<aggregate>	==	<part class name> <whole multiplicity> <part multiplicity>
<association>	==	<association name> [<association class name>] <left class name> <right class name> <left multiplicity> <right multiplicity> [<left role>] [<right role>]
<multiplicity>	==	{<range>}*
<range>	==	<lower bound> <upper bound>

We limit the whole multiplicity of an aggregation to the values 1 and {0,1}, because one instance of a part can never be in two wholes at the same time.

2.2 Dynamic Model

A state diagram in *OMT** describes the control aspects of *one specific* class, see also (Rumbaugh, 1991). It contains the possible states of a object, describes how input events are used to initiate state transitions and describes when and how functions are activated.

For the same reasons as why we restrict the object model in *OMT**, the state diagrams of *OMT** are currently restricted to state diagrams that

1. do not contain concurrent sub-state diagrams,
2. do not contain any splitting/synchronization of control,
3. do not return a result at the terminal state, and
4. contain exactly one initial lambda transition.

States. A state diagram consists of a number of state definitions and exactly one initial lambda transition. This initial lambda transition is fired on creation of the object.

A state then is defined by its state name, which must be unique within the state diagram. It will optionally contain a number of entry actions, exit actions and an activity.

Actions in the states and on the transitions are used to compute values, assign values

to attributes and generate output events. Entry and exit actions correspond to entry and exit actions in OMT.

Three kind of activities are defined in *OMT**: substate diagrams, continuous activities and time consuming activities. An activity of a state starts when the state is entered and ends when the activity is finished or when the state is left.

- Making the activity a state diagram is the only way to express the notion of hierarchical state diagrams. As a state can only contain one activity, it is not possible to have concurrent substate diagrams.
- A continuous activity simply consists of an entry action and an exit action. The entry action should start up the activity as a side effect, while the exit action should stop the activity. Also, the actions of the continuous activity should be the most inner ones, i.e. the last entry action and the first exit action.
- A time consuming ending activity calls a function, executing the function may take time. When the activity is finished a random lambda transition is fired. Actually, no normal external transitions are allowed in a state with a time consuming activity, because the activity cannot be interrupted.

State Transitions. State transitions allow performing actions or activities reacting on incoming events. Three kinds of transitions exists: external transitions, external lambda transitions and internal transitions.

<div style="border: 1px solid black; padding: 10px; display: inline-block;"> <div style="display: flex; align-items: center;"> <div style="flex: 1;"> <p><external transition></p> </div> <div style="flex: 1; text-align: center;"> <p>==</p> </div> <div style="flex: 2;"> <p>[<<u>destination state name</u>>]</p> <p><input event></p> <p>[<condition>]</p> <p><action list></p> </div> </div> </div>

An external lambda transition is exactly like a normal external transition except that the former does not have an input event nor a condition associated with it. If combined with an ending activity, the lambda transition is fired when the activity is finished.

An internal transition differs from an external transition in that it has no destination state name, since both the source and destination state are the state in which the transition is defined. Firing an internal transition does not cause the state to be left, as a consequence, the entry and exit actions are not executed.

Within the abstract syntax a terminal state is denoted by defining an empty destination state name in an external (lambda) transition. As a consequence the terminal state (graphically a dot within a circle) cannot have a name.

Actions list. An action list consists of several actions. An action list may contain any combination of assignments, output events and function calls. The order in which the actions are executed should not be important. If the order is important, extra states must be used.

A function is called or an event is sent by specifying the function name or the input event name and giving expressions for every formal parameter in the definition of the function or event. An output event optionally takes a receiver, with specifies to which class the event is sent.

3 INFORMAL SEMANTICS

Here we define in an informal way:

- the conditions for an *OMT** construct to be valid,
- the semantic link between the object model and the dynamic model.

For a more formal definition see (Holz, 1994).

3.1 Scope and Visibility

The validness of many *OMT** construct is expressed using the notion of a scope. Therefore we define a function *scope* that returns, for every part that has to be scoped, an unique identification in the model. In practice we append a kind of path to every scoped construct. For example an attribute *a* within a class *c* in a model *m* gets the following identification: (*modelT, m, classT, c, attribT, a*).

In practice this has the following effect:

- in a model: all classes and asociations must have different names
- in a class: all attributes, inpuvent declarations, function declaration and the statediagram must have different names.
- in an inpuvent declaration or a function declaration: all parameters must have a different name.
- in a statediagram: all states and the initial lambda transition must have a different name.
- in a state: all entry actions, exit actions, activities and transitions must have different names.
- in a transition: all parameters and conditions must have different names.

Using this scope concept we can also easily define the visibility of *OMT** constructs. Classes (not aggregations!), inpuvents, functions, statediagrams, states, and transitions then act like scope boundaries where you can look from the insyde to the outside and where the deepest construct with the right name is taken.

3.2 Object Model

In this section we will give some examples of semantic rules for the object model. For a complete list see (Holz, 1994).

The first rule defines the semantics of the model.

Semantic Rule 31 (omt model)

Let $m = (id, C, A) \in \langle omt\ model \rangle$, where

- $id \in \langle name \rangle$,
- $C \subset \langle class \rangle$, and
- $A \subset \langle association \rangle$,

Then *m* is a valid *OMT** model, if

- each $c \in C$ is a valid OMT^* class,
- each $a \in A$ is a valid OMT^* association.
- the scoping rules hold for this model.

Thus, a model is valid when every class and association in the model is valid and when the scoping rules hold.

A class is valid when the following rule holds.

Semantic Rule 32 (class)

Let $c = (id, A, O, sn, G, d) \in \langle class \rangle$,
where

$id \in \langle name \rangle$, $A \subset \langle attributes \rangle$, $O \subset \langle operations \rangle$,
 $sn \subset \langle name \rangle$, $G \subset \langle aggregate \rangle$, $d \subset \langle state-diagram \rangle$
then c is called a valid OMT^* class, if

- superclass name $sn \in names(visible-classes(c))$,
- each $a \in A$ is a valid OMT^* attribute,
- each $o \in O$ is a valid OMT^* operation,
- each $g \in G$ is a valid OMT^* aggregate, and
- d is a valid OMT^* state diagram or d is empty.
- the scoping rules hold for this class.

3.3 Dynamic model

One of the important aspects of the static semantic rules is the definition of the link between the dynamic model and the object model. In OMT^* every active class has its own state diagram. The collection of these state diagrams defines the dynamic model in OMT^* . This way we obtain a strong link between the dynamic model and the object model. We also demand that every function or attribute used in a state diagram of a given class is defined in, or is visible from this class. Input events have to be declared in the given class and output events have to be declared in the destination class. All this is done because it is very unclear what the link between the dynamic and the object model in OMT is.

The link is defined in the semantic rules of the dynamic model. See for example the following rule:

Semantic Rule 33 (external transition)

For each transition $t = (dest, ev, cond, instr) \in \langle external\ transition \rangle$, with $ev = (id, A)$ where

- $dest, id \in \langle name \rangle$,
- $A = (a_1, \dots, a_n) \in \langle parameter\ name \rangle^n$,
- $cond \in \langle condition \rangle$, and
- $instr \in \langle action\ list \rangle$.

Then t is called a valid OMT^1 external transition, if the following holds:

- There exists an input event declaration $e = (id', A')$ that is visible from this transition where $A' = (a'_1, \dots, a'_m) \in \langle parameter\ dcl \rangle^m$, such that

- $id = id'$ and
- $m = n$.
- $dest \in names(visible-states(t))$ or $dest = \epsilon$.
- $cond$ is a valid OMT^1 condition.
- $instr$ is a valid OMT^1 action list.
- the scoping rules hold for this transition.

This means that an external transition is only valid when the input events are already declared in the object model and the destination state of the transition is visible from this transition.

4 TRANSFORMATIONAL SEMANTIC

The dynamic semantics of OMT^* are indirectly defined by translation rules towards SDL '88. For example OMT^* blocks get the semantics of processes in SDL and components of an aggregation express subsystems, while inheritance is flattened. Figure 4 shows an example of an OMT^* model with its corresponding translation into SDL. In the dynamic model a close mapping between OMT and SDL of states and transitions allows a translation with little compromises. Only hierarchical substate diagrams and inheritance of state diagrams are difficult to translate.

Figure 2 Translation from OMT^* towards SDL

The translation rules given here work only for rather simple problems because they are not flexible enough to adapt to the different semantics some concept can have in OMT. The translation process and therefore also the dynamic semantics of OMT^* is ongoing research. Our goal is to define several possible translations for classes, aggregations and associations. The choice of which translation rule to apply could then be made manually by the designer or supported by the translator program through a set of heuristics.

4.1 Object Model

A primary requirement for the transformation of the *OMT** object model to SDL, is that the resulting SDL specifications should match the logical structure of the *OMT** specification as much as possible. This is mainly because the generated specifications will be further refined by human developers. Therefore, they must be able to recognize the logical structure defined within the original *OMT** specification. Concretely, this implies that in case of a trade off between completeness and readability, readability should be favored as much as possible.

Object Model

The translation of the object model is done in several steps.

Figure 3 Translation the Object Model towards SDL

Before translating an *OMT** object model we will first flatten the inheritance. Doing this we lose some of the nice properties of inheritance. This is inevitable because SDL'88 does not contain the notion of inheritance. In the future however we are planning a translation towards SDL'92 which has object oriented properties. We hope that this will solve most of these problems.

Since we want to translate components of an aggregation into subblocks we have problems when we translate associations. Therefore we first add to every class a kind of aggregation-path, and the set of associations which have to be defined partially or completely in the corresponding SDL block.

Flattening inheritance

Definition 41 (*Flattening – functions*) Let $c = (id, A, O, s, G, d) \in \langle class \rangle$, and let $E \in \langle inputevent dcl \rangle$ and $F \in \langle function dcl \rangle$ such that $O = E \cup F$. Define then

- $Attributes(c) = A \cup \{a \in Attributes(s) \mid a \notin A\}$
- $Events(c) = E \cup \{e \in Events(s) \mid e \notin E\}$
- $Functions(c) = F \cup \{f \in Functions(s) \mid f \notin F\}$
- $Operations(c) = Events(c) \cup Functions(c)$
- $Aggregates(c) = G \cup \{g \in Aggregates(s) \mid g \notin G\}$

Defining the aggregation-path

Definition 42 (*TopClasses(m)*) Let $m = (id, C, A) \in \langle model \rangle$ be an OMT* model such that $C = \{c_1, \dots, c_k\}$, $A = \{a_1, \dots, a_l\}$, then
 $TopClasses(m) = C \setminus \bigcup_{x \in C} Aggregates(x)$

Definition 43 (*Path*) A path is a n -tuple $(p_1, \dots, p_n)$ such that $p_1 \in TopClasses$ and $\forall i, 1 < i \leq n : p_{i+1} \in Aggregates(p_i)$

Definition 44 (*ExpandedClasses(m)*) Let $m = (id, C, A) \in \langle model \rangle$ be an OMT* model then $ExpandedClasses(m) = \{ (id, Attributes(c), Operations(c), superclass, Aggregations(c), Associations(c), Path) \mid \exists Path = (p_1, \dots, p_n) \in Paths \text{ such that } p_n = c \in C \}$

Annotate the associations Since we translate components of an aggregation into sub-blocks we have to reroute an association to the environment of the enclosing block before we are able to connect it. Therefore we define functions to split the associations into system-associations, internal-associations, partial-associations, complete-associations and complete system-associations, see figure 4. A system-association connects two topclasses. An internal-association connects an internal process block with its environmental block. A partial-association denotes a connection between a block and its environment, a complete-association is a connection at the lowest level where we can connect the two parts of a non-system-association and a complete-system-association is a connection at the system level of two parts of a non-system-associations.

Because of space constraints we give only the definition of a partial association. For a complete description see (Holz, 1994).

Definition 45 (*Partial – association(c)*) Let $a = (id, lc, rc, lm, rm, lr, rr) \in Associations(c)$, let $plc = (plc_1, \dots, plc_m) = path(lc)$, $prc = (prc_1, \dots, prc_n) = path(rc)$, and let $commonp = (c_1, \dots, c_p)$, such that $\forall i \leq p : c_i = plc_i = prc_i$. Then $Partial - associations(c) = \{a \in Associations(c) \mid c \in (plc_{p+1}, \dots, plc_m) \text{ or } c \in (prc_{p+1}, \dots, prc_n)\}$

Note that multiplicities are not taken into account in this translation. The multiplicities of the associations define in fact the number of instances to be taken of a given class. When a class has more than one association this number becomes ambiguous. There is not enough information in the analysis stage to take this kind of decisions. Therefore we have chosen to postpone this decision to the detailed design phase in SDL'88.

System An OMT* model is translated into a SDL system containing the SDL translations for the classes and associations defined within the model.

Translation rule 41 (*sdl-module*) Let $m = (id, C, A) \in \langle model \rangle$ be an OMT* model such that $ExpandedClasses(m) = \{ec_1, \dots, ec_k\}$, and $SystemAssociations(m) = \{sa_1, \dots, sa_n\}$, and $CompleteSystemAssociations(m) = \{csa_1, \dots, csa_m\}$

Then *sdl-module(m)* is constructed by

```

system <id> ;
  <sdl-class (ec1, ..., eck)>; /* blocks with process */
  <sdl-CompleteSystemAssociation(m) (csa1, ..., csam)>; /* channels */
  <sdl-SystemAssociation(m) (sa1, ..., san)>; /* channels */
endsystem <id>;

```

Classes An OMT class definition *c* is translated to SDL as a block containing:

- A subblock containing the behaviour and data of *c*. This includes the attributes, operations and state diagram of *c*. This is a leaf block.
- A subblock for every component class *p* in the aggregation tree of *c*, generated by calling *sdl-class(p)* recursively.

In order to describe the translation of a class we have to define some functions to calculate which signals we need. The function *intern* returns the events that should be defined within this block. Currently, this are all the events defined within the components of the current class. The function *between* returns the events that are passed from its first to its second argument. Currently, this is the intersection between the events sent by its first argument and the events declared within its second argument.

Translation rule 42 (*sdl-class*) Let $c = (id, V, O, S, G, d)$ be an OMT* class such that $attributes(c) = \{v_1, \dots, v_k\}$, $operations(c) = \{o_1, \dots, o_l\}$, $PartialAssociations(c) = \{pa_1, \dots, pa_m\}$, $CompleteAssociations(c) = \{ca_1, \dots, ca_n\}$ and $InternalAssociations(c) = \{ia_1, \dots, ia_p\}$ and $Aggregates(c) = \{g_1, \dots, g_q\}$

Then *sdl-class(c)* is constructed by

```

block <id> ;
  substructure
    signal <intern(c)>; /* Signals local to this block */

    /* Components */
    block <g1> referenced;
    :
    block <gm> referenced ;

  block <id>-internal ; /* skipped if q=0 */
    process <id>-process ;
      dcl <sdl-operation(o1, ..., ok)>

      <sdl-action(a1, ..., al)>
      /* no signal routes */
      <sdl-state-diagram(d, attributes(c))> ; /* optional */

```

```

    endprocess <id>-process ;
  endblock <id>-internal /* skipped if q=0 */
  <sdl-InternalAssociations (ia1, ..., iap)>;
  <sdl-PartialAssociations (pa1, ..., pam)>;
  <sdl-CompleteAssociations (ca1, ..., can)>;
  substructure
endblock <id>;

```

Associations. Associations are translated to channels connecting the blocks associated with its left and right classes. The signal list is defined to be the events that can be passed between both classes.

Translation rule 43 (*sdl-SystemAssociation*) If $a = (id, lc, rc, lm, rm, lr, rr) \in \text{SystemAssociations}$ then *sdl-SystemAssociation*(a) is constructed by ,

```

channel <id>
  from <lc> to <rc>
    with <between(lc, rc)> ;
  from <rc> to <lc>
    with <between(rc, lc)> ;
endchannel <id>
connect <id> with <id> ;

```

The other associations (partial, intern, etc.) are translated in a similar way.

Dynamic Model

Figure 4 Translating the Dynamic Model towards SDL

An *OMT** state diagram will be translated into an SDL state diagram. For each state within the *OMT** state diagram one state in the SDL state diagram is introduced. As SDL does not distinguish between internal and standard transitions, caution is needed in the translation of entry and exit actions. This is solved by executing an entry action only on external transitions, before the SDL state is entered.

The most difficult part is the translation of substate diagrams. The substates need a

copy of the transitions of their superstates, but these transitions need to be expanded with additional exit actions for the substate. Also, the destination of a transition should be changed to the initial state of the substate diagram of the destination state. Therefore we expand the states with path information which allows us to build all transitions.

Building the Substate Tree. Here, all states are expanded with path information, so that each state exactly knows in which substates it is defined. First we explain some access functions used in the definitions for an easier notation.

- *Subdiagram(state)* returns the activity of state, if that activity is a state diagram.
- *Substates(state)* returns the states in Subdiagram(s).
- *TreeSubStates(state)* returns all Substates, defined recursively.

Definition 46 (StatePath) A statepath is an n -tuple $(s_1, \dots, s_n)$, such that for each $i \in \{1, \dots, n-1\}$ holds $s_{i+1} \in \text{Substates}(s_i)$.

Definition 47 (ExpandedState(sd)) Let $sd = (i, S) \in \langle \text{state-diagram} \rangle$. Then *ExpandedStates(sd)* is the set of all tuples $(id, \text{entry}, \text{exit}, \text{activity}, T, \text{Path})$ where exists a $\text{Path} = (s_1, \dots, s_n) \in \text{statepath}$, such that

- $s_1 \in S$
- $s_n = (id, \text{entry}, \text{exit}, \text{activity}, T, \text{Path}) \in \text{TreeSubStates}(S)$

We assume that expanded states are compatible with the *OMT** states, so the functions defined before can also be used on expanded states.

Deep Action-Access Functions

Definition 48 (path access functions) Let $p = (s_1, \dots, s_k)$, where $s_i = (id_i, \text{entry}_i, \text{exit}_i, \text{activity}_i, T_i, \text{path}_i)$. Then

$$\text{path-exits}(p) = (\text{exit}_1, \dots, \text{exit}_k)$$

$$\text{path-states}(p) = \bigcup_{i=1 \dots k} s_i$$

Path-exits is used when a transition is copied to one of its sub-states. The new transition then executes the exit actions of all superstates up to the state which contains the original transition.

Two functions are defined to calculate which actions need to be performed when entering a state, taking into account that the state may contain a substate diagram.

- *initial-action(state diagram)* returns the actions on the initial lambda transition of the state diagram and the entry-actions of the initial state.
- *entry-actions(state)* returns the entry actions of the state and the initial-actions of the substate diagram if there is one.

Initial-actions and entry-actions call each other recursively to make a list of all the initial transition actions and entry actions all the way down to the deepest initial substate.

Finally, the function *Subdest(state)* calculates the real destination when entering the state, this is the initial state of the substate diagram if there is one. This initial state can again have a substate diagram and so on.

Copying and Rerouting transitions. The function `ExpandedTransitions` calculates all the transitions for a specific (sub)state, given its path. The algorithm is based on the fact that the base-state is the last element in the path and the state from which the transitions are copied is the first element. In each recursive step all transitions from the top state are copied and extended with the exit actions of all states on the path and the initial actions for entering the destination state. The set of transitions is then extended by a recursive call with a shorter path, i.e. the first element is removed. In this way the target state gets the transitions of all its superstates.

We use two auxiliary functions `External-transitions(s)` and `Internal-transitions(s)`. `External-transitions` returns both the normal external transitions and external lambda transitions, where the lambda transitions become external transitions with an empty event. `Internal-transitions` returns the internal transitions of the state.

Definition 49 (*ExpandedTransitions(path)*) Let $path = (s_1, \dots, s_k)$ such that

- $s_1 = (id, e, x, a, T, p)$
- $s_k = (id', e', x', a', T', path')$, where $path' = (s'_1, \dots, s'_l)$

Then

- $ExpandedTransitions(path) =$
 $\{ (dest', event', cond', action') \mid$
 $\exists t = (dest, event, cond, action) \in external-transitions(s_1) :$
 $dest' = sub-dest(\epsilon, dest) \text{ for } dest \neq \epsilon$
 $dest' = s'_{l-1} \text{ for } dest = \epsilon \text{ and } l > 1 \text{ /* trace back */}$
 $dest' = \epsilon \text{ for } dest = \epsilon \text{ and } l = 1 \text{ /* terminal state */}$
 $event' = event$
 $cond' = cond$
 $action' = append(path-exits(path), (action), entry-actions(d))$
 $\} \cup ExpandedTransitions(s_2, \dots, s_k)$

- $ExpandedTransitions(\epsilon) = \{ \}$

Translation Rules Given the expanded states and expanded transitions, the translation is straightforward. No environment information is needed, because each *OMT** construct can be translated in the same order as it appears in the syntax tree. Notice that the state-diagram is flattened, so all states are on the same level.

Translation rule 44 (*sdl-state-diagram(sd)*) Let $sd = (i, S) \in \langle state\ diagram \rangle$, where $i = (dest, \epsilon, actions) \in \langle \underline{initial}\ lambda\ transition \rangle$, let $d \in S : name(dest) = d$, d is then the initial state of sd ($a_1, \dots, a_k$) = $initial-actions(sd) \{s_1, \dots, s_k\} = expanded-states(TreeSubStates(S))$.

Then *sdl-state-diagram(sd)* is constructed by

```

start ;
  sdl-action-list( $a_1$ ) ... sdl-action-list( $a_k$ )
  nextstate <name(sub-dest( $d$ ))> /* stop if  $d = \epsilon$  */

```

```

<sdl-state( $s_1$ )>
:
<sdl-state( $s_k$ )>

```

An *OMT** state is simply translated as a SDL state containing all Expanded-Transition on the path of the state and all internal transitions of the state. If state contains a time consuming ending activity, then the activity is called first and a random lambda transition is fired.

Translation rule 45 (*sdl-external-transition*(t))

Let $t = (dest, event, cond, exit, action, entry) \in \langle expanded-transition \rangle$, where $dest \in \epsilon(\langle destination\ state\ name \rangle)$ $event \in \epsilon(\langle input\ event \rangle)$, $cond \in \epsilon(\langle boolean\ expression \rangle)$, and $action = (a_1, \dots, a_k), a_{1..k} \in \langle action-list \rangle$.

Then *sdl-external-transition*(t) is constructed by

```

input <sdl-input-event(event)>; /* Skipped if event =  $\epsilon$  */
provided <sdl-expression(cond)>; /* Skipped if cond =  $\epsilon$  and event  $\neq \epsilon$  */
/* provided true if event = cond =  $\epsilon$  */
<sdl-action-list( $a_1$ )> ;
:
<sdl-action-list( $a_k$ )> ;
nextstate dest;

```

Internal transitions are translated like external transitions, except that an internal transition only contains one action list and that nextstate is set to “-” to return to the same state at the end of the transition. Note also that for internal transitions there must always be an event, i.e. an empty event is not allowed, so the “input” line is never skipped.

5 CONCLUSION

We presented an automated transition from OMT to SDL. OMT has been chosen for its wide spread use in system engineering and for its integration of static and dynamic concepts. SDL on the other hand, is very well suited for the design of highly interactive systems in a formal way. In the development of large complex systems which involves many people, it is important to have a smooth transition from analysis to design while preserving as much information as possible. In order to allow such a transition we introduced *OMT** to formalise OMT.

*OMT** was formally defined in three steps. The abstract syntax and static semantics define the requirements for an *OMT** model to be valid. The transformational semantics then map this valid object model and dynamic model onto SDL.

Apart from SDL, other languages can be generated starting from OMT. In the IN-SYDE project (Insyde, Technical Annex, 1994) SDL and VHDL are generated for hybrid systems co-design. A translation for the object model into Z is given in (CCITT, 1988)

and the programming techniques group at CERN (Aimar, 1993) describes a configurable code generator for OO methodologies. However, most of these proposals support only the translation of the object model of OMT, while we also developed a thorough translation for the dynamic model. This aspect is very important in the telecommunication domain which is of interest to us.

In the future we will work on a translation toward SDL'92 which will make it possible to keep the whole methodology object oriented. In addition to this, we are working to make the translation more flexible, i.e. different translation rules for the same construct depending on additional information. Last but not least, we are validating *OMT** in cooperation with industrial partners. During the process the methodology is tested by the industrial partners in the consortium and the first results look very promising.

REFERENCES

- G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. *Procs of the 1. ACM SIGSOFT, Symposion on the Foundations of Software Engineering*, December 1993.
- A. Aimar, A. Khodabandeh, P. Palazzi, and B. Rousseau. A configurable code generator for OO methodologies. Technical report, Programming Techniques Group, CERN, 1993. CCITT, Geneva. *ITU Specification and Description Language SDL, Recommendation Z.100 Blue Book.*, november 1988.
- E. Holz, M. Wasowski, D. Witaszek, S. Lau, J. Fischer, L. Cuypers, J. Heirbaut, K. Verschaeve, and V. Jonckers. The insyde methodology - report. Report IN-SYDE/WP1/HUB/003/v3, ESPRIT Ref: P8641, November 1994.
- INSYDE. *Technical Annex: "Integrated Methods for Evolving System Design"*, ESPRIT-III Project P8641, restricted report edition, December 1994.
- Z. Navabi. *VHDL Analysis and Modeling of Digital Systems*. McGraw-Hill, Inc., 1993.
- J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.