Translating OMT* to SDL, Coupling Object-Oriented Analysis and Design with Formal Description Techniques

K. Verschaeve, B. Wydaeghe, V. Jonckers, L. Cuypers
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: kaversch@info.vub.ac.be

Abstract

This paper presents an automated transition from OMT* (a formal variant of OMT) towards SDL. This work is a partial result from a larger research effort proposing an integrated methodology and toolset based on the combination of Object-Orientation and Formal-Description Techniques. In this project OMT is used as the systems requirements analysis technique and OMT* for for System Design, while SDL (Specification Description Language) is targeted for the design phase. The transition from OMT to OMT* is manual process described by a set of guidelines (Holz et al. 1995) We developed a transformational semantic for OMT*, i.e. a set of transformation rules mapping OMT* constructs to SDL constructs. The translation from OMT* to SDL preserves the logical structure of the specification. This way it is possible to preserve the efforts done in the analysis phase and to make a smooth transition towards design.

Keywords

OMT*, SDL, Analysis, Design, Transformational Semantics, Software Engineering

1 INTRODUCTION

1.1 The INSYDE Project

The INSYDE (INtegrated methods for evolving SYstem DEsign, INSYDE 1994) methodology is a set of techniques and tools to enable the evolving co-design of hybrid systems. 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-Orientation and Formal-Description Techniques and covers the development lifecycle from Systems Requirements Analysis over System Design to Detailed Design and Validation in an integrated way.

S. Brinkkemper et al. (eds.), *Method Engineering*
© Springer Science+Business Media Dordrecht 1996

The INSYDE project is an EU ESPRIT III funded project. The consortium consists of Alcatel Bell Telephone (Belgium), Dublin City University (Ireland), Humboldt 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 Technique, 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.

Our Lab for System and Software Engineering (LaSSE) does research in the field of telecommunication systems. Therefor we focus on the use of SDL in the INSYDE methodology, SDL being a widely used specification standard and very well suited for our purpose.

In this paper we limit the scope to the translation of OMT to SDL. This transition happens partially manually and partially automatically as explained below. This translation is important and interesting because there is a strong need for an automatic reuse of analysis information into software specification languages like SDL. An automatic translation encourages the developer to make a more thorough system design model. Moreover, the structure of the resulting SDL model will be like the structure of the OMT model, resulting in system that is easier to maintain.

1.2 OMT*

In our methodology the analysis is done in OMT, using the full richness of OMT as defined by Rumbaugh et al. (1991). Constructs such as classes or associations can have different semantics depending on their context, which is useful during the requirement analysis phase.

While OMT is a good analysis methodology, the informal nature of OMT makes an automatic translation to SDL infeasible. In our methodology the analysis document is manually prepared for translation during system design. During this phase subsystems are identified, communication is formalized and information is ordered. To describe these aspects we developed a new language OMT* (formal definition in Wasowski, 1995), aimed to meet the requirements of system design. In our methodology OMT* is close to both OMT and SDL.

- OMT* is close to OMT because they use the same syntactic structures and because the semantics of OMT* are compatible with Rumbaugh, i.e. the semantics of OMT* do not conflict with OMT. OMT* differs from OMT in that it contains a number of syntactical constraints and in that the possible interpretations of an OMT construct are reduced and clearly described. Detailed guidelines of how to make the transition from OMT to OMT* can be found in the INSYDE application guidelines (Holz et al. 1996). Also a brief overview of the methodology is available in (Sinclair et al., 1995).
- OMT* is close to SDL because there is automatic translation and because OMT* and SDL have corresponding structure and semantics. The generated SDL is readable and contains enough detail to be a good framework as a starting point for detailed design.

1.3 Quick Preview of the Translation from OMT* to SDL

In this paper, we describe the transformation of OMT* to SDL by defining a *transformational semantics* for OMT*. These semantics consists of a set of translation rules for the object model and the dynamic model. We do not use the functional model of OMT, because this model does not give much additional information over the object and dynamic models to generate SDL.

The translation rules for OMT* are based on the availability and the semantics of constructs in both OMT and SDL. Figure 1 shows how some of the OMT constructs are mapped on SDL constructs. For example the basic building blocks in OMT are classes while in SDL they are the system, blocks and processes. So it is a natural choice to map a class on either a system, block and/or process. In the same way the structuring mechanism of OMT is aggregation while in SDL this is done by nesting of blocks. Finally the expression of relationships between classes is done by associations in OMT and with communication paths in SDL.

Semantics	OMT*	SDL
Basic Building Block	Class	System, Block, Process
Structuring (Subsystems)	Aggregation	Nested Blocks
Relationship between classes	Association	Communication

Figure 1 Mapping of object model of OMT* constructs on SDL

We have a similar table for the translation of the dynamic model of OMT*, see figure 2. In OMT* the behaviour of a class is expressed by a state diagram. This state diagram is translated as a SDL process specification. It is straightforward to translate state and state transitions to the equivalent constructs in SDL. Entry and exit actions are translated as actions on the transition to and from that state respectively. Internal transitions are translated as transition with itself as destination.

OMT*	SDL
Activity	Process or Part of Process
State	State
State transition	State Transition
Entry/Exit Actions	Actions on State Transition
Internal Transitions	State Transition to Self

Figure 2 Mapping of dynamic model of OMT* on SDL

As some OMT constructs can take several possible translations, local translation of each construct in the OMT model by a corresponding construct in SDL is not possible. We need global information of the model to make the correct translation. In an extra phase before the translation (sections 3 and 5) we gather this information.

1.4 Structure of the paper

In the next section we will start with a short overview of OMT* to introduce the concepts used in the translation. In section three we will describe how we prepare the translation of the object model followed in section four by the translation rules for the object model. Section five describes the preparation of the translation for the dynamic model and section six gives the translation rules for the dynamic model.

Within this paper we use only tiny OMT and SDL examples to clarify some concepts. The INSYDE methodology and the translation of OMT* to SDL has been successfully tested is an industrial case studies of a Video-on-Demand server system (Peeters et al. 1995).

We assume that the reader is acquainted with both OMT and SDL.

2 OVERVIEW OF OMT*

This section gives a short overview of OMT*. This language is used as a system design language between the analysis in OMT and the detailed design in SDL. OMT* has a syntax which is very similar to OMT but it has well defined semantics defined by its translation towards SDL.

2.1 Object model

An OMT* specification is entered through the object model. 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 model.

The syntax of OMT* contains a number of restrictions as opposed to OMT, because some constructs in OMT have an 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.

These restrictions are only valid during system design. The analysis is done in full OMT, without these constraints. There are detailed guidelines available how to manually translate OMT into OMT* in (Holz et al. 1996), available at the World Wide Web at "http://www.compapp.dcu.ie/ gclynch/papers.html". In general we could say that most changes needed to get OMT* are rather intuitive for somebody that is acquainted with the translation rules to SDL.

Model definition. An OMT* model contains a list of classes and a list of associations. Figure 3 shows the model of a Movie-Box containing three classes. The classes *Control* and *Motor* do have pointers to a state diagrams shown below.

Class definition. An OMT* class is a six-tupple (id,V,O,sn,G,d) where *id* denotes the name of the class, *V* denotes the set of attributes, *O* denotes the possible input events and functions defined, *sn* is the name of the superclass, *G* is the set of components and *d* is the state diagram, describing the dynamic behavior of this class. In our example (figure 3) the six-tupple describing class *motor* is

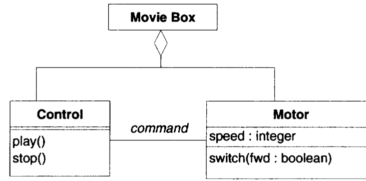

Figure 3 Simplified Model of a Movie-Box in OMT*

$(motor, \{speed\}, \{switch\}, \epsilon, \emptyset, \epsilon)$, where *speed* and *switch* are references to the specific attribute and operation respectively.

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.

An operation either is 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, aggregations and associations in OMT* 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. Associations denote communication between objects.

In OMT* an association is a unbounded construct, described as a seven-tupple (id,lc,rc,lm,rm,lr,rr) where id denotes the name of the association, lc and rc are the names of the classes that are connected by this association, lm and rm denote the multiplicities and lr and rr denote the roles. In our example (figure 3) the seven-tupple of the association *command* could be described as $(command, Control, Motor, 1, 1, \epsilon, \epsilon)$.

In OMT* an aggregation is part of the specification of a class. An aggregate tupple contains a component id, the aggregate multiplicity and the component multiplicity. We limit the aggregate multiplicity to the values 1 and {0,1}, mainly because in SDL is strictly hierarchical, i.e., a process can never be in two disjunct blocks at the same time. The set of aggregates (G) of the MovieBox class is $\{(Control, 1, 1), (Motor, 1, 1)\}$.

2.2 Dynamic Model

States Diagram. A state diagram in OMT* describes the control aspects of *one specific* class. It contains the possible states of an object, describes how input events are used to initiate state transitions and describes when and how functions are activated.

A state diagram consists of a number of state definitions (in figure 4, the states *idle* and *play*)

and exactly one initial lambda transition (in figure 4, the transition going from • to *idle*). This initial lambda transition is fired on creation of the object. 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. contain exactly one initial lambda transition.

Each of these restriction can be worked around: restriction 1 by using two classes, restriction 2 by using extra synchronization events and restriction 3 by introducing an extra state. All other features of OMT state diagrams, like nested state-diagrams, activities and entry and exit actions, are fully supported in OMT*.

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

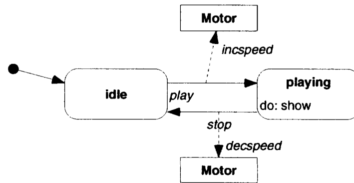

Figure 4 Dynamic Model of the *Control* class of figure 3.

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, like in OMT, are executed on entering and leaving the state respectively.

While in OMT activities are poorly defined, in OMT* three kinds of activities are defined: substate-diagrams, continuous activities and time consuming activities. An activity of a state always starts when the state is entered and ends when the activity is finished or when a state is left.

- A substate-diagram is a complete state diagram, with states and transitions, embedded in a state. There is no limitation in the nesting of substate-diagrams. During the translation to SDL, substate-diagrams are flattened.
- A continuous activity is a simple activity that is automatically terminated when leaving the state. A continuous activity consists of a name from which a placeholder for an entry and exit action is generated. The entry action should start up the activity as a side effect, while the exit action should stop the activity. In figure 4, *show* is an continuous activity because *show* is not defined as an operation in the *Control* class. The translation of *show* is shown in figure 9.
- A time consuming ending activity calls a function. Executing this function may take time and cannot be interrupted. After the activity has finished, the state is left by firing a lambda transition.

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.

An external transition consists of the name of the destination state, a list of input events, a condition and an action list.

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.

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 PREPARING THE OBJECT MODEL TRANSLATION

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.

Before translating an OMT* object model we will first remove inheritance using some flattening functions. This is necessary because SDL'88 does not contain the notion of inheritance. Afterwards we build a kind of an annotated aggregation tree. This step facilitates the translation process considerably, because the aggregation structure is not available as such in the OMT* abstract syntax, where all classes are defined on the same level. As a last step before the translation all associations have to be rerouted because of the translation of aggregation into subblocks.

3.1 Removing inheritance

To flatten the inheritance structure we first have to introduce an auxiliary function *Subtree*. This function returns the set of classes that are in the aggregation tree for a given class.

Definition 31 ($Subtree(c)$) *Let* $c = (id, V, O, S, G, d) \in$ *<class>*.
Then $Subtree(c) = \{c\} \cup \bigcup_{g \in Aggregates(c)} Subtree(g)$

Using this function we can flatten every OMT* class with the following flattening functions.

Definition 32 (*Flattening-functions*) *Let* $m = (id, C, A) \in$ *<model>, where* $C \subset$ *<class>. Let* $c = (id, V, O, s, G, d) \in C$, *and let* $E \in$ *<inputevent dcl> and* $F \in$ *<function dcl> such that* $O = E \cup F$
Define then

- $Attributes(c) = V \cup \{v \in Attributes(s) \mid v \notin V\}$
- $Events(c) = E \cup \{e \in Events(s) \mid e \notin E\}$
- $Functions(c) = F \cup \{f \in Functions(s) \mid f \notin F\}$
- $Operations(c) = Events(c) \cup Functions(c)$
- $Components(c) = G \cup \{g \in Components(s) \mid g \notin G\}$
- $Associations(c) = \{a \in A \mid lc \in Subtree(c) \text{ or } rc \in Subtree(c)\}$

3.2 Building the aggregation tree

The aggregation tree is built by adding a path to every class. This path consists of an ordered list with the names of all classes which connect the given class with a top node. Therefore we will first introduce the function TopClasses. This function returns the set of all classes in a given model m that are on top of the aggregation trees (they are no part of the set of aggregations of any class in the model).

Definition 33 ($TopClasses(m)$) *Let* $m = (id, C, A) \in$ *<model> be an OMT* model such that* $C = \{c_1, \ldots, c_k\}$, $A = \{a_1, \ldots, a_l\}$, *then*
$TopClasses(m) = C \setminus \bigcup_{x \in C} Aggregates(x)$

If the model contains only one topclass, this class is translated into SDL as the *system*. Otherwise a new *system class* is added to the model, see figure 5. The *system class* for a model m is constructed by taking all *TopClasses* as components, but no attributes, operations or state diagram. Because of this definition there will always be only one topclass.

Definition 34 (*Paths*) *A path is a n-tuple* $(p_1, \ldots, p_n)$ *such that* $p_1 \in TopClasses$ *and* $\forall i, 1 < i \leq n : p_{i+1} \in Aggregates(p_i)$. *The set of all paths is called "Paths".*

Because of the construction of a system class, the paths of all classes in a model will start with the system class, since it is the only TopClass in the model. For example, in figure 5 Class C has as path (System, A, C).

Definition 35 ($ExpandedClasses(m)$) *Let* $m = (id, C, A) \in$ *<model> be an OMT* model then* $ExpandedClasses(m) =$ *{ (id, Attributes(c), Operations(c), superclass, Aggregations(c),Path)* $\mid Path = (p_1, \ldots, p_n) \in Paths$ *such that* $p_n = c \in C\}$

In other words expanded classes is the set of all classes after flattening inheritance and extended with path information. This path is then used to reroute associations, see below.

3.3 Annotate the associations

Since we translate aggregation into subblocks 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 partial-associations and complete-associations, see figure 5. A partial-association denotes a connection between a block and its environment and a complete-association is a connection at the lowest level were we can connect the two parts of an association.

The following rules define the partial associations and the complete associations.

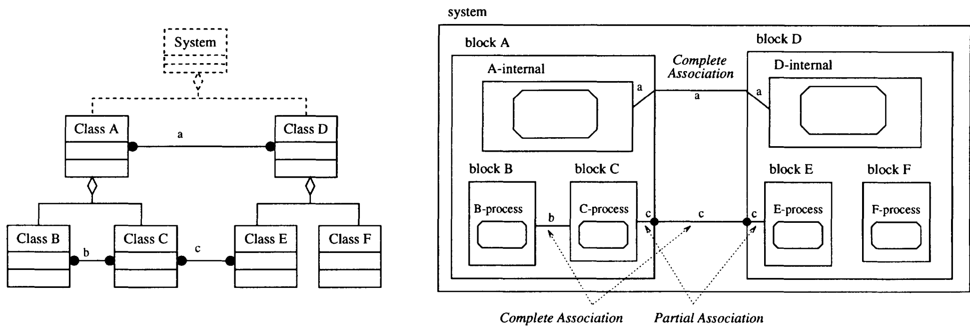

Figure 5 Translation of the OMT* structure and the associations towards SDL

Definition 36 (*Partial-association(c)*)
Let $a = (id, lc, rc, lm, rm, lr, rr) \in Associations(c)$,
let $plc = (plc_1, \ldots, plc_m) = path(lc), prc = (prc_1, \ldots, prc_n) = path(rc)$, *and*
let $commonp = (c_1, \ldots, c_p)$,*such that* $\forall i \leq p : c_i = plc_i = prc_i$.
Then
$Partial\text{-}associations(c) =$
$\{a \in Associations(c) \mid c \in (plc_{p+1}, \ldots, plc_m)$ *or*
$c \in (prc_{p+1}, \ldots, prc_n)\}$

A complete associations is added when a class is the "deepest" class in the common part of the paths of the left and right class of an association. In order to store which components must be connected, the functions return a set of 3-tuples. For example, in figure 5 the *System* gets two complete association because the paths for *A* and *D* and for *C* and *E* come together in System. The classes to be connected are A and D in both cases.

Definition 37 (*Complete-association(c)*)
Let $a = (id, lc, rc, lm, rm, lr, rr) \in Associations(c)$,
let $plc = (plc_1, \ldots, plc_m) = path(lc), prc = (prc_1, \ldots, prc_n) = path(rc)$, *and*
let $commonp = (c_1, \ldots, c_p)$, *such that* $\forall i (i \leq p) c_i = plc_i = prc_i$
Then
$Complete\text{-}associations(c) =$
$\{(a, tlc, trc) \mid a \in Associations(c)$ *and* $c = plc_p = prc_p$ *and*
$tlc = plc_{p+1}$ *and* $trc = prc_{p+1}\}$

In addition we use a function *Local-Signals(c)* to gather the necessary signal declarations in a given class. The gathering of declarations are defined by three rules. A signal is declared in a given class if

- The class itself uses the signal.
- Or two components of the class use the same signal.
- And signal is *not* already declared in one of its aggregates (recursive definition).

4 TRANSLATION RULES FOR THE OBJECT MODEL

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

Translation rule 41 (*sdl-module*) *Let* $m = (id, C, A) \in$ *<model> be an OMT* model. Let system = (id,V,O,sc,G,sd,path) be the expanded-system-class of* m *as defined in section 3.2, and* $\{top_ec_1, \ldots, top_ec_k\}$ *the expanded classes of Components(system), and* $\{ec_1, \ldots, ec_k\} = ExpandedClasses(m) \setminus Components(system), and \{sa_1, \ldots, sa_n\}$ =Complete–Associations(m), $and \{ev_1, \ldots, ev_m\}$ = *Local–Signals(system)*

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

system *<id> ;*
  signal *sdl-event-declaration*$(ev_1), \ldots,$ *sdl-event-declaration*(ev_m);
  <sdl-class (top_ec$_1$*)>; /* system blocks*/*
  ⋮
  <sdl-class (top_ec$_k$*)>;*
  <sdl-CompleteAssociation (sa$_1$*)>; /* channels */*
  ⋮
  <sdl-CompleteAssociation (sa$_n$*)>;*
endsystem *<id>;*
<sdl-class (ec$_1$*)>; /* referenced blocks*/*
⋮
<sdl-class (ec$_k$*)>;*

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. If c does not have any components, the surrounding subblock is skipped.
- A subblock for every component class p in the aggregation tree of c, generated by calling *sdl-class*(p) recursively.

Translation rule 42 (*sdl-class(c)*)
Let $c = (id, V, O, sc, G, sd)$ *be an OMT* class, such that* $attributes(c) = \{v_1, \ldots, v_k\}$, $operations(c) = \{o_1, \ldots, o_l\}$, $PartialAssociations(c) = \{pa_1, \ldots, pa_m\}$, $CompleteAssociations(c) = \{ca_1, \ldots, ca_n\}$, *and* $Components(c) = \{g_1, \ldots, g_q\}$. *Local–Signals(c)* $= \{ev_1, \ldots, ev_p\}$

Then sdl–class(c) *is constructed by*

block *<id> ;*
  substructure
    / Signals Definitions*/*
    signal *sdl-event-declaration*$(ev_1, \ldots, ev_p)$;
    / Components */*

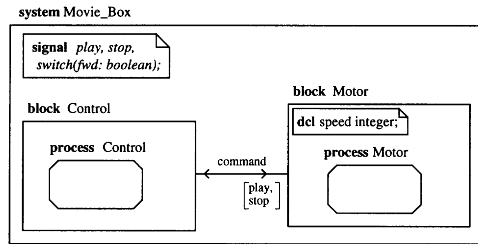

Figure 6 Translation of the Object Model of the Movie-Box

block $<g_1>$ **referenced**;
⋮
block $<g_q>$ **referenced** ;

block <*id*>*-intern ; /* skipped if q=0 (no components) */*
  process <*id*>*-process ;*
    dcl $<sdl\text{-}attribute(v_1, \ldots, v_k)>$
    $<sdl\text{-}operation(o_1, \ldots, o_l)>$
    / no signal routes */*
    $<sdl\text{-}state\text{-}diagram(sd, attributes(c))>$ *; /* optional */*
  endprocess <*id*>*-process ;*
endblock <*id*>*-intern /* skipped if q=0 */*

$<sdl\text{-}PartialAssociations\ (c, pa_1, \ldots, pa_m)>$*; /* channels */*
$<sdl\text{-}CompleteAssociations\ (ca_1, \ldots, ca_n)>$*; /* channels */*
endsubstructure
endblock <*id*>*;*

Associations. Associations are translated to channels connecting the blocks associated with its left and right classes. To calculate the events sent between two classes we use the function *between*. This is the intersection between the events sent by its first argument and the events declared within its second argument.

Translation rule 43 (*sdl–CompleteAssociation*)
If $ca = (a, tlc, trc)$ *where* $a = (id, lc, rc, lm, rm, lr, rr) \in <Associations>$ *and* $tlc, trc \in <class>$. *Then* $sdl\text{-}CompleteAssociation(ca)$ *is constructed by*

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

PartialAssociations are translated in a similar way, it differs only in that a partial association goes to the environment (**ENV**) instead of to a class.

Attributes and operations. An attribute is translated into an SDL declaration of the correct type and initial value, e.g. *Speed* in figure 6. A function is translated into a skeleton of an SDL procedure. Th return type of the function is translated as an in/out parameter of the procedure.

5 PREPARING THE TRANSLATION FOR THE DYNAMIC MODEL

An OMT* state diagram will be translated into an SDL state diagram. For each state within the OMT* state diagram one state within the SDL state diagram is introduced. As SDL, however, 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.

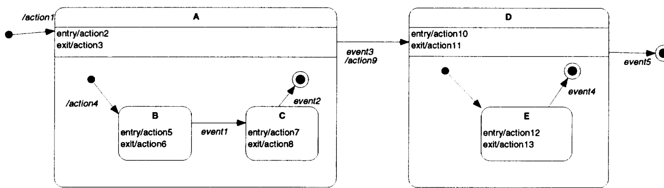

Figure 7 Example of a Nested Statediagram.

The most difficult part is however 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.

5.1 Building the Substate Tree.

As with expanded classes, we expand all states with path information, so that each state exactly knows in which substates it is defined. In following definitions we use two functions: *Substates(state)* returns the substates of a given state and *TreeSubStates(state)* which is like *Substates* but include the substate of the substates and so on.

Definition 51 ($StatePaths$) *A statepath is an n-tuple* $(s_1, \ldots, s_n)$*, such that for each* $i \in \{1, \ldots, n-1\}$ *holds* $s_{i+1} \in Substates(s_i)$.

Definition 52 ($ExpandedState(sd)$) *Let* $sd = (i, S) \in$ *<state-diagram>, where* i *ist the initial state and* S *is the set of states of* sd*. Then* $ExpandedStates(sd)$ *is the set of all tuples* $(id, entry, exit, activity, transitions, Path)$ *where exists a* $Path = (s_1, \ldots, s_n) \in StatePaths$*, such that*

- $s_1 \in S$
- $s_n = (id, entry, exit, activity, transitions) \in TreeSubStates(S)\}$

In other words, *ExpandedStates* gathers all the states in a statediagram, including substates, and appends path information to each state. In the example in figure 7, state A has path (A), state B has path (A,B), state C has path (A,C), etc. So B and C inherit all transitions from A.

5.2 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.

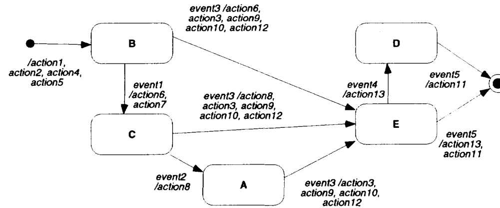

Figure 8 Flattening in OMT* of the Nested Statediagram Example.

The function *ExpandedTransitions* calculates all transisitions of a state, given its path. But because of the complexity and size of the function definition, we only show here the result after applying the flattening functions to the example in figure 7. The flattened statediagram is shown in figure 8.

Notice that the state transition with event3, previously going to state D, is now going to state E immediately because state E is the initial substate of D. For the same reason, state B is now the initial state of the statediagram.

Notice also that event3 is now present in states A, B and C, but that the transition starting in state A calls less exit actions than the transition starting from B and C. Therefor it is not possible, in general, to assign the same transition to a state and its substates.

6 TRANSLATION RULES FOR THE DYNAMIC MODEL

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. Figure 9 shows the translation of the state diagram of the *control* class (figure 4).

Translation rule 61 (*sdl-state-diagram(sd)*) *Let* $sd = (i, S) \in$ *<state diagram>, where* $i = (dest, \epsilon, actions) \in$ *<initial lambda transition>, let* $d \in S : name(dest) = d$, *d is then the initial state of sd* $(a_1, \ldots, a_k) = initial-actions(sd)$ $\{s_1, \ldots, s_k\} = expanded-states(TreeSubStates(S))$.

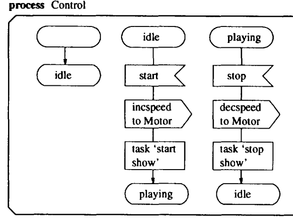

Figure 9 Translation of the Dynamic Model of the Control

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)>

6.1 States

An OMT* state is simply translated as an SDL state containing all Expanded-Transition on the path of the state and all internal transitions of the state, see figure 8 and figure 9.

Translation rule 62 (*sdl-state(s)*)
Let $s = (id, e, x, activity, T, path)$ ∈ <*expanded-state*>, *where* id ∈ <*name*>, e ∈ <*entry action list*>, x ∈ <*exit action list*>, *activity* ∈ <*activity*>, $(t_1, \ldots, t_m) = ExpandedTransitions(path)$, $(i_1, \ldots, i_n) = InternalTransitions(path - states(path))$, *and path is a statepath.*

Then sdl-state(s) is constructed by

state <*id*> ;
 <*sdl-external-transition*(t_1)> ... <*sdl-external-transition*(t_m)>
 <*sdl-internal-transition*(i_1)> ... <*sdl-internal-transition*(i_n)>
endstate <*id*> ;

6.2 Transitions

When performing an external transitions, all the entry actions and exit actions that were calculated in the expanded transitions should be executed. The following rule also applies to lambda transitions.

Translation rule 63 (*sdl-external-transition(t)*) *Let* $t = (dest, event, cond, exit, action, entry)$ ∈ <*expanded-transition*>, *where dest* ∈ ϵ(<*destination state name*>) *event* ∈ ϵ(<*input event*>), *cond* ∈ ϵ(<*boolean expression*>), *and action* = $(a_1, \ldots, a_k), a_{1 \ldots k}$ ∈ <*action-list*>.

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

input <*sdl-input-event(event)*>; /* *Skipped if event* = ϵ */
provided <*sdl-expression(cond)*>; /* *Skipped if cond* = ϵ *and event* $\neq \epsilon$ */
/* **provided** *true* *if event* = *cond* = ϵ *
<*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.

6.3 Actions

An action-list is of course translated as a list of actions. There are three kind of actions: function-call, output event and assignment. For each kind there is a different translation rule, described below.

- function-call, let $f = (func, arg) \in$ <function call>
  call <func> (<*sdl-expression*(arg_1)>, ..., <*sdl-expression*(arg_k)>);
- output-event, let $e = (event, arg) \in$ <output event>
  output <event> (<*sdl-expression*(arg_1)>, ..., <*sdl-expression*(arg_k)>);
- assignment, let $a = (attr, expr) \in$ <assignment>
  attr := *sdl-expression*(*expr*);

7 CONCLUSION

We present 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 information. 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 developed OMT*.

OMT* is used as a system design language. The transition from an OMT requirement analysis to an OMT* system design model requires manual design decisions. Detailed guidelines about this transition are available in (Holz et al. 1996). OMT* is a subset of OMT, but OMT* contains as many constructs of OMT as possible. The semantics of OMT* are well defined with a transformational semantics to SDL'88.

OMT*, and the translation to SDL, should not be seen as a way to design and implement an arbitrary system modeled in OMT. Instead, OMT and OMT* should be seen as a front-end to the design of a system that is being designed in SDL anyway. We state that OMT provides the right abstraction level for requirements analysis and that OMT* successfully couples the object-oriented modeling technique (OMT) with the software description language (SDL). We do know of some companies (Alcatel Mobile and IskratTEL) that already use both OMT for analysis and SDL for

design in the development of very large telecommunication systems. Of course, those companies could get immediate benefit from our methodology.

Future plans on OMT* include a translation towards SDL'92. This version of SDL has object capabilities and allows probably a better translation in that it preserves more information expressed by the system design model such as aggregation and inheritance.

Apart from SDL, other languages can be generated starting from OMT. In the INSYDE project (INSYDE, 1994) SDL and VHDL are generated for hybrid systems co-design. A translation for the object model into Z is given in (Abowd, 1993). A formal semantics in terms of algebras has been defined for the object model in (Bourdeau, 1995). The programming techniques group at CERN (Aimar et al., 1993) describe 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 integrated a thorough translation for the dynamic model. This aspect is very important in the domain of telecommunication.

REFERENCES

G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. *Procs of the 1. ACM SIGSOFT, Symposium 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, 1993.

R. Bourdeau and B. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, October 1995.

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, P. Roques, K. Verschaeve, E. Mariatos, and J.-P. Delpiroux. The insyde methodology. Deliverable INSYDE/WP1/HUB/400/v2, ESPRIT Ref: P8641, January 1996.

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. Peeters, M. Jadoul, E. Holz, M. Wasowski, D. Witaszek, and J.P. Delpiroux. Hw/sw co-design and the simulation of a multimedia application. In *7th European Simulation Symposium*, October 1995.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

D. Sinclair, G. Clynch, and B. Stone. An object-oriented methodology from requirements to validation. In *2nd International Conference on Object Oriented Systems*, December 1995.

M. Wasowski, D. Witaszek, K. Verschaeve, B. Wydaeghe, E. Holz, and V. Jonckers. The complete omt*. Deliverable INSYDE/WP1/HUB/300/v3, ESPRIT Ref: P8641, December 1995.