Scalable and Object Oriented SDL State(chart)s

Birger Møller-Pedersen[1], and Dagbjørn Nogva[2]
[1)] NorARC - Applied Research Center, Ericsson Norway: [2)] Telox, Norway

Key words: Statecharts, SDL, UML, state type, state type inheritance, virtual state

Abstract:

The notion of composite state in SDL is introduced. The features include entry/exit points to enforce encapsulation, type/subtypes of any composite state, virtual states in order to control the redefinition of states when defining state subtypes, and parameterized state types in order to allow maximum reuse of the same state type in different contexts. This notion of composite states will be part of SDL-2000. The same mechanism can, however, as well be introduced in other languages with a state machine concept.

1. INTRODUCTION

For a family of applications it is important to describe objects by means of state machines. Specification languages support this by the notions of either simple finite state machines, extended (with variables) finite state machines, or variations of Harel's Statecharts [9]).

The most important reason for using state machines is that it ensures that all possible states of the object and its reaction to all possible events are covered. State machines may, however, become large and complex, and their specification correspondingly large and complex. The notion of hierarchical states (sometimes also called states with nested states or composite states) introduced by Harel's Statecharts is a recognized solution to this problem, and it has been adopted by many notations and specification languages.

SDL as of 1996 [6] supports extended finite state machines with one level of states. In addition, procedures may be described by state machines, but the states of a procedure are entered when the calling state machine is in a transition.

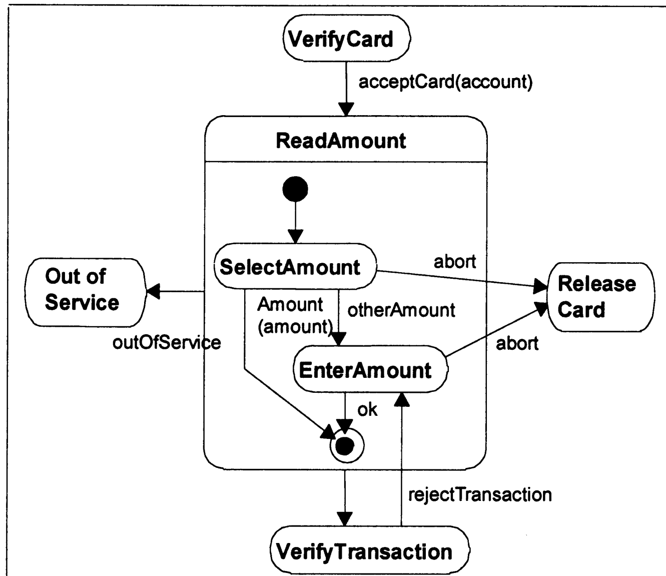
J. Wu et al. (eds.), *Formal Methods for Protocol Engineering and Distributed Systems*
© Springer Science+Business Media Dordrecht 1999

The emerging proposal for SDL-2000 includes support for composite states. This paper presents the most important aspects of these composite states:

- *Types and subtypes of states* for states in general, and not only for the outermost state as in Statecharts, combined with the introduction of *state diagrams* with *state connection points* facilitating encapsulation and scalability of state definitions.
- The distinction between *virtual state types* (of a supertype state) that can be *redefined* in a subtype state, and states that can *not* be redefined (in the spirit of SDL's finalized types/transitions and Java's final methods).
- *Parameterized state types* in line with SDL types with context parameters and UML parameterized object classes, enabling the definition of composite state types that can be used in different state machines within the same system and even in different systems.

2. STATE MACHINES, COMPOSITE STATES

Figure 1 is a partial state machine for an ATM as supported by UML statecharts [8].



The example covers the part of an ATM that reads the desired amount: the user either selects the amount (the choices given by the ATM), or enters the amount. One of the benefits of Statecharts is illustrated by the outOfService event: this applies to all states being part of the composite state ReadAmount.

Figure 1. Statechart (UML)

Figure 1 provides the overview of which states are involved and the transitions between these. As specified here, the transitions are denoted by

the events that trigger the transitions. In general it is possible to specify more than this. Actions being part of transitions are specified textually, in separate diagrams or editor windows, and just associated with the transition arcs.
The same partial state machine is specified in SDL-2000 in *Figure 2* and *Figure 3*.

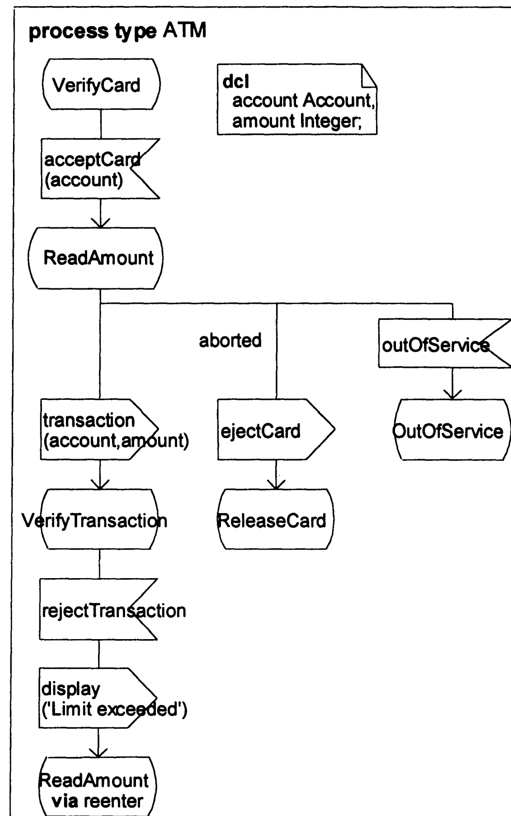


Figure 2. SDL State Machine with Composite State

The most apparent difference between statecharts and SDL composite states is that the content of the composite state ReadAmount is described in a separate diagram in SDL. *Figure 2* is a partial process type diagram with a state machine specification that *references* a state diagram.

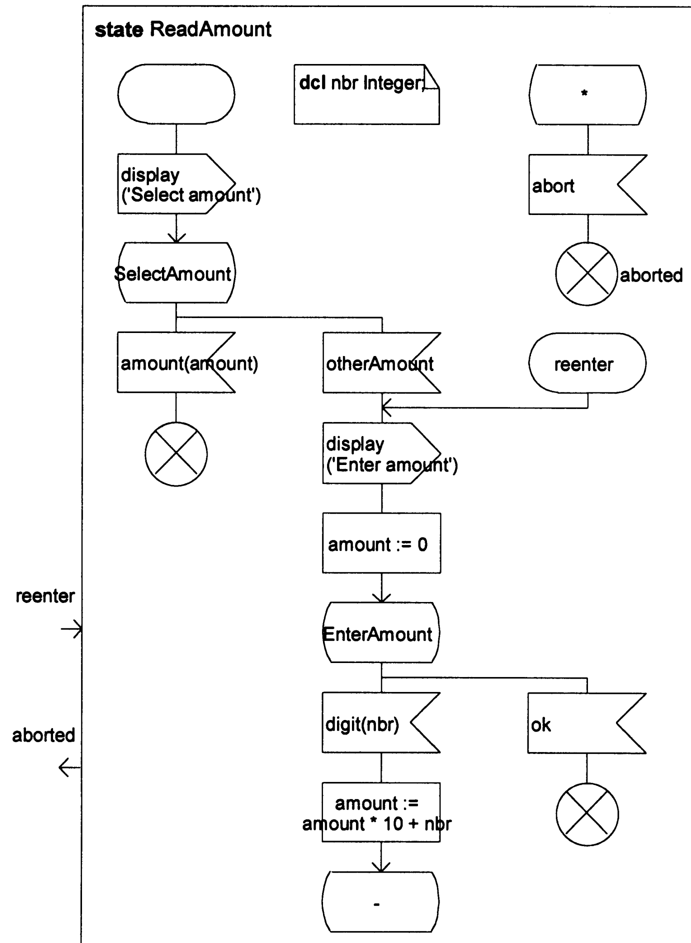


Figure 3. Composite State Diagram

A composite state is entered and exited by means of connection points defined as part of the interface of the composite state. Within the state diagram for the composite state, an entry connection point is represented by a special *start symbol* with the name of the entry points (reenter in *Figure 3*), An exit connection point is represented by an *exit symbol* with the name of the point associated (aborted in *Figure 3*).

The unlabeled start and exit have the following correspondence to Statecharts: The initial state of Statecharts corresponds to the state following the unlabeled start symbol, and the final state corresponds to the exit symbol without any exit point associated.

Input transitions associated with a composite state apply to all substates. As an example, in process type ATM in *Figure 2*, the input transition with

signal outOfService applies to all substates of the composite state ReadAmount. A transition in SDL is the input of a signal and the following sequence of actions.

As a summary of the SDL composite as introduced above, they differ from Statecharts in the following respects:

- SDL composite states scale up to handle real, large systems by having composite states described in *separate state diagrams*.
- Entering and exiting a composite state is *not* specified by crossing state boundaries as in Statecharts, but by means of *connections* to *state connection points*, thereby providing encapsulation. The internal specification of a composite state can be changed without effecting the enclosing state machine.
- While Harel statecharts appear *state-oriented*, SDL state machines are specified in a *transition-oriented* way. While state orientation provides overview, some developers prefer the transition orientation when it comes to the specification of the transitions. It is usually also regarded as a benefit to see both transitions caused by incoming signals *and* outputs of signals.

3. ENTERING AND EXITING COMPOSITE STATES

Entering a composite state at a specific point is specified by the name of the composite state followed by via <entry point name>. In *Figure 2*, if the ATM rejects the transaction, it reenters the composite state ReadAmount via the reenter connection point, and in the state diagram for ReadAmount it is specified where the state is reentered (by a start symbol with the label reenter.

Entering a composite state in the "initial state" is a special case of this; specified by the name of the composite state, without any via <entry point name>. The composite state is then entered at the unlabelled start symbol. In *Figure 2* this happens when the ATM has got the signal acceptCard (in state VerifyCard). The composite state is entered at the unlabelled start symbol in the left uppermost part of state ReadAmount.

The composite state may exit in two different ways:

- it has got the amount signal in state SelectAmount or the ok signal in state EnterAmount, and simply terminates, in that case the process type ATM continues at the unlabelled connection line after ReadAmount, or
- it aborts; in that case the ATM continues at the connection line labeled abort following ReadAmount.

While a Statecharts transition crossing a state boundary is *one* transition, specified as part of the whole statechart, a corresponding SDL-2000

transition is a combination of a inner sub-transition defined in the composite state and an outer sub-transition defined in the enclosing state diagram. In *Figure 4* it is illustrated what that would mean in UML Statecharts. An SDL-2000 transition being just one sequence of actions is just a special case, where the outer or inner transition is left empty.

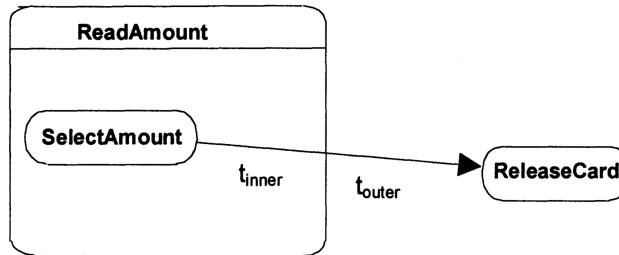


Figure 4. Transitions as combination of outer and inner transitions

Entry/exit actions are specified by means of textual procedures defined locally in the state diagram. The procedures, named *entry* and *exit*, are called implicitly when a state is entered and exited, respectively. The names *entry* and *exit* are predefined names.

The entry/exit actions may not take parameters, but may access variables defined in the state diagram and in enclosing entities according to the usual scope rules.

Note that there is no notational difference between a composite state and a non-composite. One reason for this is that states with entry/exit actions (but not necessarily any contained states) are described by separate state diagrams. As described below, a state may also have just internal transitions. Finally, the decision of whether a state is a composite state or not, may change during development of the complete state machine.

4. SPECIAL FEATURES

The SDL composite state example also illustrates a special SDL feature: the *asterisk state* with input of the signal abort implies that this transition applies to *all* states. In general, a transition can be specified to be applicable to

- all states,
- all states, except a specified list of states,
- a list of states.

This feature provides an alternative way of grouping states than provided by composite states. An input transition specified for a composite state

applies to all substates, while an input transition specified for a list of states applies to a set of states that are not necessarily substates. SDL-2000 with composite states will provide both alternatives and their combination.

5. INTERNAL TRANSITIONS

An internal transition is a transition that remains in a state. An internal transition is not equivalent to a self-transition from a state back to the same state. A self-transition causes the exit and entry actions on the state to be executed, whereas an internal transition does not.

Internal transitions in SDL-2000 are part of composite states, even if they do not have substates. In a state with no substates, the asterisk state is merely a syntactical way of referring to the composite state itself, see *Figure 5*.

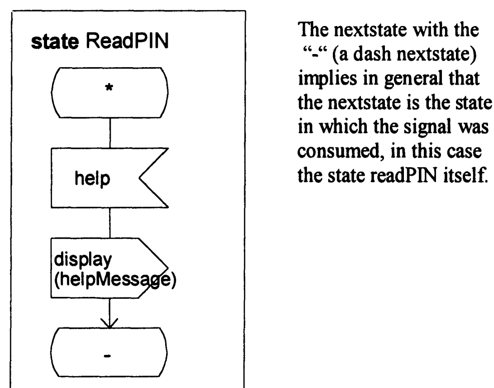
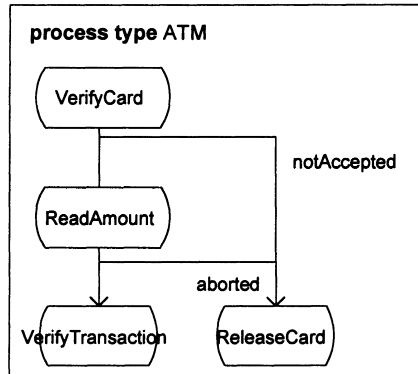


Figure 5. Internal transition in SDL

6. STATE OVERVIEW DIAGRAMS

A composite state can as a special case have transitions only (and not just internal transitions). This can be used to obtain state overview diagrams very similar to Statecharts. The state diagram of ATM in *Figure 6* is an example of a state overview diagram, where only the connections between the states are specified. The transitions corresponding to these connections are specified in separate state diagrams. These are sketched in *Figure 7*.



This state overview diagram relies on the fact that the composite states VerifyCard and ReadAmonut define the transitions. In this diagram it is only specified that there may be two transitions from VerifyCard, and two transitions from ReadAmount. The details about these transitions, e.g. which signals that trigger them and what actions are performed as part of the transitions, are found in the separate state diagrams.

Figure 6. State overview diagram

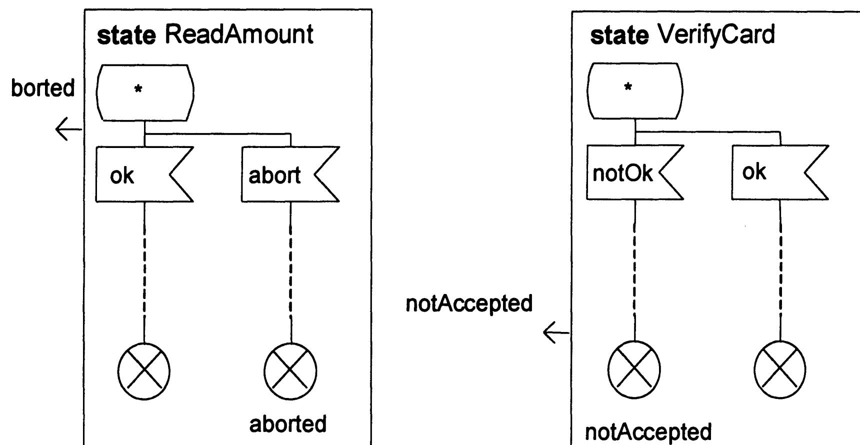


Figure 7. Composite State Diagrams covering all Transitions from the State Overview Diagram

7. STATE TYPES AND SPECIALISATION

If a composite state shall be used in different situations, a *composite state type* is defined, for short called a *state type*. A state with the properties of a state type is called a *type-based state*. The syntax is simple. The type name is specified after the name of the state, separated by a colon: "<state name>:<state type name>". This is by the way the standard syntax for type-based instances in SDL.

State types are useful in cases where the properties of composite state are applicable in many systems in the same domain, or even in different parts of

the same system. Several larger state machines may use composite state types describing the states and transitions needed for e.g. starting/stopping, providing status or handling general management signals.

A state type may be defined as a *specialisation* of another (super) state type, *inheriting states and transitions* of the supertype and *redefining virtual states* and *transitions*. This is the same mechanism as already present in SDL. States of the supertype cannot be deleted, but virtual states can be redefined. States and transitions can be added in a subtype.

Entry/exit procedures may also be defined as virtual procedures that can be redefined.

Figure 8 shows an example of a state type that inherits a general type and adds abort functionality. Adding this functionality includes adding a state exit point (aborted). The asterisk state in the subtype specifies that the abort input applies to all states, including the inherited states (in this case SelectAmount).

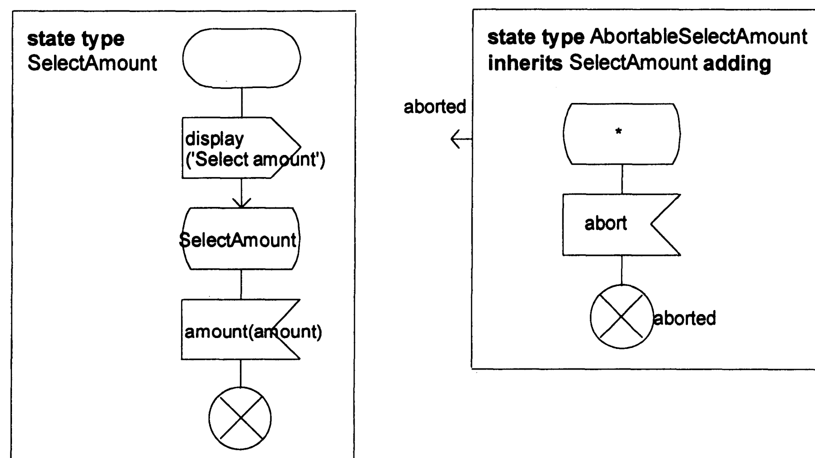


Figure 8. State Type Inheritance

In SDL-2000, any composite state type at any level can be defined as a subtype of another state type. For Statecharts associated with object modeling notations, the outermost state is associated with a class, and inheritance for states follows the class inheritance. The implication of this is that inheritance only applies to the outermost state.

8. VIRTUAL STATES

A state type may be defined to be a virtual type in order to allow redefinition, when the enclosing type is subtyped. *Figure 9* and *Figure 10* provide an example of the redefinition of a virtual state type when the enclosing process type is subtyped. The example also shows how an exit transition in the enclosing process type is defined as virtual and redefined in the subtype.

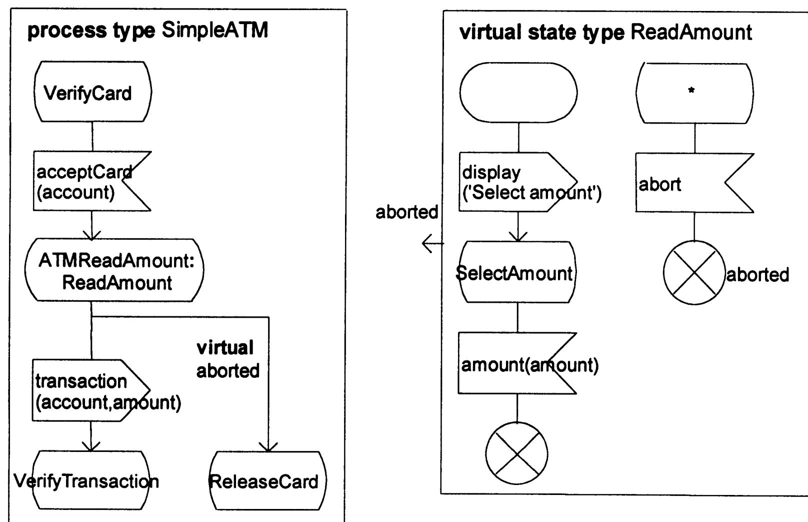


Figure 9. Virtual states/redefinition of states (1)

The state type ReadAmount is defined as a virtual state type and used for defining a state, by the expression

"ATMReadAmount: ReadAmount"

The definition of the contents of the virtual state type will in all redefinitions be inherited, that is only virtual properties of the virtual state type can be redefined.

In *Figure 10* the redefined state type inherits the corresponding state type definition in the super process type, and extends it with the state EnterAmount and with three transitions. The state SelectAmount in the redefinition is the inherited state with the same name defined in the virtual state type.

The virtual exit transition (aborted) is a transition defined as part of SimpleATM and not as part of ReadAmount. It is therefore redefined as part of the enclosing process subtype ATM.

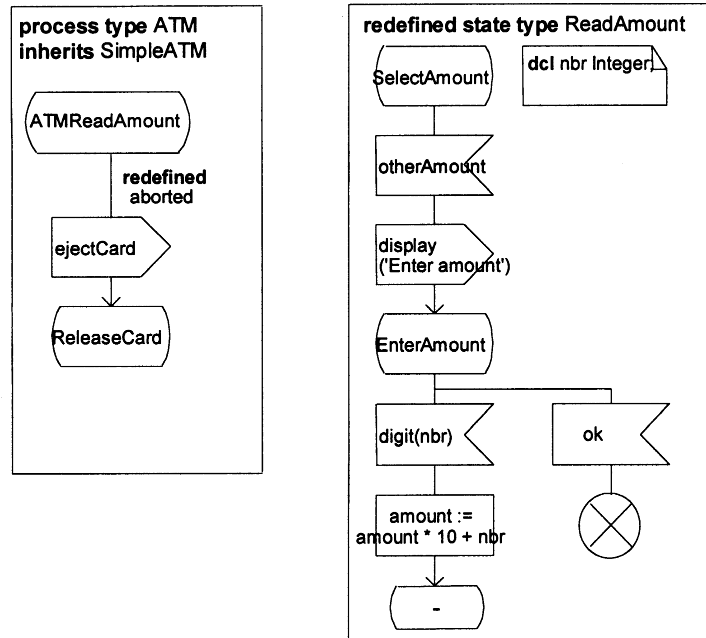


Figure 10. Virtual States/redefinition of States (2)

A virtual state can not just be redefined to any state with the same name. A virtual type has a constraint, and redefinitions have to be extensions (by inheritance) of the constraint. A constraint is either another, more general state type, or it is the virtual type itself. In the last case, the redefinition is simply an extension of the virtual state.

9. PARAMETERIZED STATE TYPES

In order for state types to be defined in *packages* and used in different systems with different local definitions, state types can have type parameters, in the same way as provided for classes (template parameters) in many other languages.

Virtual types in SDL (the correspondence would be virtual nested classes in UML and virtual inner classes in Java) provides one way of expressing that a type has type parameters, in that a virtual type can be redefined to any (visible) type that satisfies the constraint on the virtual type. For a further description of this in SDL, see [5]. For its application to Java, [4] is recommended. Virtual state types have been described above.

SDL has, in addition to virtual types, the notion of context parameters in general for type definitions, and this mechanism is in SDL-2000 also applied

to state types. Context parameters are a little more general than just type parameters: a context parameter can be of any kind of entity that may be defined in the context of the type. Possible context parameters for state types are data types, signals, procedures, variables and other state types. *Figure 11* gives an example, where the parameter is a variable.

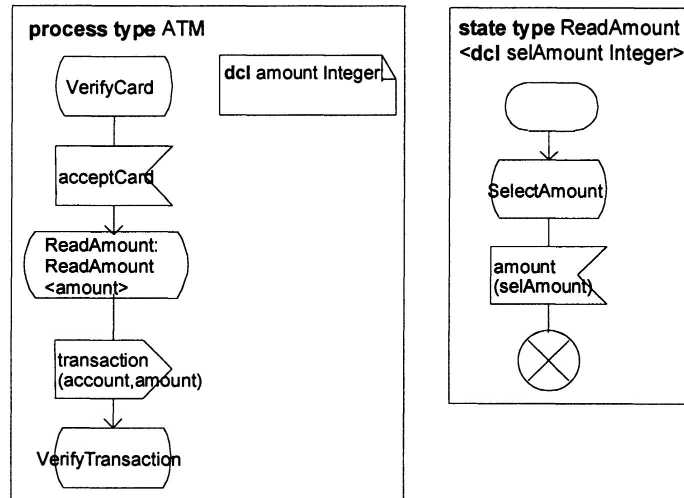


Figure 11. Parameterised State Type

The state type ReadAmount is defined to have an Integer typed variable as parameter. Binding this variable to a local variable in the process type ATM implies that all accesses to selAmount in the state type will become accesses to the process local variable amount.

Parameterised types are necessary in order to facilitate reuse in real system development, and together with virtual types it allows the definition of frameworks by means of abstract (super) types that are specialized to actual applications. In [5] it is demonstrated how virtual types can be used to make frameworks. It covers only the virtuality of types like system-, block-, and process types, and virtual procedures. With virtual state types it is possible to extend the notion of frameworks to cover also the state machine description of process types.

10. FORMAL DEFINITION

In the current stage of standardization, composite states are defined by transformation to basic SDL, which in turn has a formal definition. This

transformation is part of the Z.100 recommendation. This scheme may change with the new approach to the formal definition of SDL, using Abstract State Machines (ASMs), [11].

11. RELATED WORK

UML state machines have a notion of *submachinestate*, which resembles the SDL-2000 notion of either a composite state that *references* a separate state diagram, or a type-based composite state. According to the UML Semantics, this mechanism should "facilitate reuse and modularity", but submachinestate is not a Classifier and not a GeneralizableElement, so inheritance is not supported. It is also said in the UML Semantics that it is a shorthand "that implies a macro-like expansion". Graphical macros have been removed from SDL-2000, and state diagrams in SDL-2000 are *not* macros, but real scope units. This implies that a state diagram forms its local name space, i.e. names of entities defined in enclosing scope units are visible in the same way as for other kinds of scope units in SDL. Entities of the composite state scope unit have to be specified as part of its interface in order to be visible from outside.

A *stubstate* in UML state machines corresponds to a *nextstate via an entry point* in SDL-2000. The difference is that a stubstate can reach any substate to any depth, while a nextstate via an entry point can only reach a first level starting point within the composite state. In addition, this starting point has to be defined as an entry point as part of the *interface* of the composite state. The SDL-2000 notion of nextstate as part of a composite state is therefore less flexible than stubstate, but it supports encapsulation and thereby independent development. The enclosing state machine only has to know the entry/exit points of a composite state, and the content of a composite state can be changed without changing the use of it in the enclosing state machine.

As mentioned above, the notion of type/subtype of composite states is in UML state machines only supported for the topmost state associated with a class. But even for this topmost state, the UML Semantics does not say anything definite about inheritance of states and transitions. A note describes some alternatives that may be considered.

SDL-2000 has first of all the notion of type/subtype for composite state at any level, not only the topmost. Secondly, it is well defined how inheritance works. *Virtual states and transitions* as parts of a state type (or directly of a process type or procedure) can be *redefined* in state subtypes (or process subtypes or subprocedures). If they are finalized (like Java's final), they can

not be redefined in further subtypes. States and transitions can also be added to the states and transition inherited from the supertype.

Even though UML does not define inheritance for state machines, various methods for using UML recommend approaches that are similar to the one described here. In [10] it is recommended to follow a.o. these rules when inheriting a state machine type (in [10] called a "state model"):

- New states and transitions may be added
- States and transitions defined by the parent cannot be deleted
- Action and activity lists may be changed
- Actions and activities may be specialised.

Other similar notations have an opposite view on this. In [3] it is argued that strict inheritance (as the above approach is called in [3]) does not guarantee anything, so in ROOM it is possible to delete states and transition. It is true that strict inheritance is a kind of syntactic restriction on inheritance (that cannot guarantee anything in terms of the actual application), but it is still better than complete freedom in deletion of properties.

No other approaches to inheritance of state machines have the notion of constraints on redefinition of inherited, virtual states. The "strict inheritance" in SDL-2000 is extended with the notion of a virtual state type *constraint* (in terms of a more general state type or the virtual state type itself). A virtual state type with constraint can only be redefined to a state type that is a subtype of the constraint This assures that all redefinitions are extensions of this constraint. If no constraint is specified, the virtual state is itself the constraint; the implication is that redefinitions are extensions of the virtual state. The last case is covered by the rules in [10]. For SDL-2000 it is also covered by the language, and thereby checked by tool.

The notion of constraints on virtual state types is not a mechanism special for SDL. It first appeared in [2], and recent proposals for adding type parameters to Java ([4] being one of these) also provide constraints on type parameters. This is needed in order for Java to be a strongly typed language. The parameterised class can use the type parameters according to the constraint. The same is the case for virtual states in SDL-2000: the enclosing state machine can use the constraint of a virtual state, e.g. entry/exit connection points.

12. CONCLUSION

It has been demonstrated that it is possible to obtain the best of two "worlds": the transition-orientation of SDL state machines *and* the state-orientation of Statecharts. The transition-orientation has focus on the where the functionality is specified and therefore provides designers with the right

mechanism during detailed design. The state-orientation is superior for providing an overview of large and complex state machines. The notion of composite state of SDL-2000 provides this combination, but it does more than this. It scales up and handles large state machines by state references and by separate state diagrams. It supports encapsulation by defining interfaces of composite states, instead of allowing any substate of a composite state to be entered directly. Finally it has been demonstrated that it is possible to have real object oriented composite states. Any composite state can be subject to the type/subtype mechanism; state types can inherit states and transitions, and redefine virtual states; state types can be generalised by type parameters in order to be used in different contexts.

ACKNOWLEDGEMENTS

Although the authors are responsible for the idea behind composite states in SDL, for the first versions of their definition, and of course for the contents of this paper, the notion of composite states is the result of the joint effort of the working group ITU-T Study Group 10 Question 6.

REFERENCES

[1] D. Harel & E Gery: *Executable Object Models with Statecharts*. IEEE Computer, 1997.

[2] O. Lehrmann Madsen & B. Møller-Pedersen: *Virtual Classes - a Powerful Mechanism in Object Oriented Programming*. OOPSLA'89, Sigplan Notices, Vol 24 Number 10, 1989.

[3] B. Selic: *An efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems*. http://www.objectime.com/otl/technical/efficient.html.

[4] K. Thorup: *Genericity in Java with Virtual Types*. European Conference on Object-Oriented Programming, LNCS 1241 Springer-Verlag, 1997.

[5] Bræk & B. Møller-Pedersen: *Frameworks by means of virtual types*. FORTE XI/PSTV XVIII'98, Paris November 1998.

[6] ITU Z.100 Specification and Description Language SDL, 1996.

[7] A. Olsen et al: *Systems Engineering Using SDL-92*. North-Holland 1994.

[8] OMG *Unified Modeling Language Specification* (draft, Version 1.3, 1999).

[9] D. Harel: *Statecharts: a visual formalism or complex systems*. Science Computer Program Vol. 8, 1987.

[10] B. P. Douglass: Real-Time UML – Developing Efficient Objects for Embedded Systems. Addison-Wesley 1998.

[11] R. Gotzheim, B. Geppert, F. Rössler, and P. Schaible: Towards a new formal SDL semantics. In Proc. Of the 1st Workshop of the SDL Forum Society and SDL and MSC, Berlin June 98.