

Structural and Behavioral Decomposition in Object Oriented Models

J. Fischer, E. Holz
Humboldt-Universität zu Berlin
Institut für Informatik
Rudower Chausse 25, D-12489 Berlin, Germany
{fischer|holz}@informatik.hu-berlin.de

B. Møller-Pedersen
Ericsson AS Norway
Applied Research Center &
Department of Informatics, University of Oslo
birger.moller-pedersen@eto.ericsson.se

Abstract

The decomposition of large systems into parts is a general principle of software design. Even more, in the scope of distributed systems a partition of the whole system into distributable components is necessary. Decisions about what constitutes a component of a system are usually either based on the behavior or on the structure of the system. Nevertheless there is a strong mutual influence between both kinds of decomposition. Despite the importance of structural and behavioral decomposition, many modeling notations and languages define the semantics of these concepts rather vague, and this may lead to incorrect implementation. This paper presents the new structuring mechanisms in the object oriented specification language SDL as of November 1999 (SDL-2000). The paper also gives a critical evaluation of these concepts and a comparison with similar approaches in UML and ROOM.

1. Introduction

Decomposition is an essential principle in order to manage and master the complexity of large software systems. Breaking up a whole system into parts, which may then be defined or further decomposed, helps to overcome the limitations of the human cognition.

In general two main directions for a decomposition can be distinguished:

- *Structural* decomposition focuses on the substance aspects of the system, e.g. the objects or processes within a system and their relations.
- *Behavioral* or algorithmic decomposition on the other side focuses on the actions and reactions of the system.

However, in most cases there is a mutual influence between both kinds of decomposition. A structural decomposition has implications on the behavior associated with objects of the system, and a behavioral decomposition may require a special structural decomposition. This holds especially for the domain of object-oriented specification and programming technique, because an object as the principal foundation of these techniques is dually characterized by its substance (attributes and part objects) and its behavior.

In most object-oriented analysis and design languages both kinds of decomposition are defined separately. The general

concept of structural decomposition are the aggregation and its stronger form composition. Both are expressing whole/part relationships. Hierarchical state machines (composite states) are the main behavioral decomposition mechanism. Most languages do not support a tight relation between structural and behavioral decomposition, as would e.g. be the case if both the structural and the behavioral decomposition are properties of the composite class. Making decomposition a property of the composite class would also imply that it would be subject to other properties of classes in general, as e.g. inheritance, dynamic generation, etc.

SDL is an object-oriented specification and design language. The SDL-96 version ([1]) provides different concepts for the structural and behavioral decomposition. The concept *block* is used to specify the static structure of a system. Blocks are containers for sets of concurrent behavior elements (processes) or are decomposed further into blocks. A process may be just a single state machine or it may be decomposed into services (orthogonal state machines).

This paper presents the new structuring mechanisms as they are in the current version of SDL, SDL-2000 [2]. The most important of these are an extended and harmonized block/process concept (called *agents*), covering all properties of the former blocks and processes, and a composite state concept with state partitions covering the properties of the former service concept. Structural and behavior decomposition are integrated as properties of the composite agents.

2. Structural Decomposition

A system in SDL-2000 is a structure of communicating agents. Each agent may be decomposed into part-agents with either a true concurrent or an alternating execution semantics. There is a strong life-time dependency between a container and its parts: A part can not outlive its container, it may neither migrate into another containers nor exist before its container. Agents may be created dynamically and may cease to exist before their container ceases to exist. The decomposition principles apply for type definitions (corresponds to class definitions in other object oriented languages) as well as for instance definitions (object definitions).

An agent consists of three main parts:

- Attributes in terms of variables (member data, partly

specifying the agents state) and behaviour pattern attributes in terms of procedures.

- Behavior in terms of a state machine.
- An internal structure in terms of contained agents with interfaces and communication infrastructure in terms of connections.

The behavior of an agent is the combined behavior of its own state machine and of the behavior of the contained agents. Each contained agent has its own unique identifier and its own input queue.

SDL-2000 makes a distinction between two kinds of agents, blocks and processes:

- A *block* agent executes *concurrently* with other blocks, both with blocks at the same level and with contained blocks. A block can contain both blocks and processes.
- A *process* agent executes *alternating* with other processes, both with processes at the same decomposition level and with contained processes. Alternating execution implies that every process executes transitions to completion. (cf. *Subsection 2.2*). A process can only contain processes, not blocks.

A *system* agent is the special outermost block agent.

2.1. Concurrent Structural Decomposition with Agents

A sketch of a block agent type definition is given in *Fig. 1*. The block agent type **CompositeBlock** is a composition of agents of two other agent types (**B** and **P**) and its own state machine (referenced by the state symbol). The contained

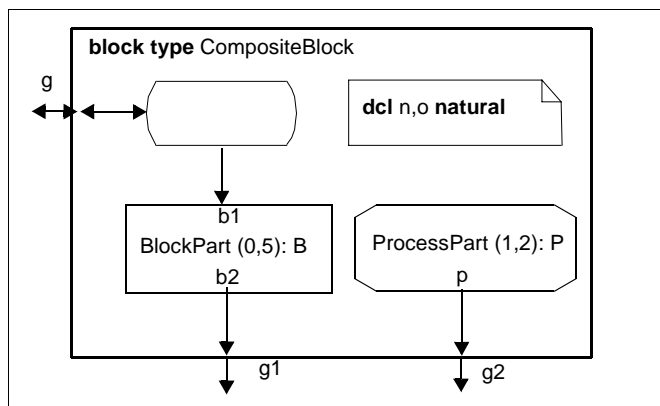


Figure 1. Sketch of block agent type definition

agents are block agents (**BlockPart** of block type **B**) and process agents (**ProcessPart** of process type **P**), distinguished by the different symbols.

Communication between the parts and between the parts and the environment of **CompositeBlock** takes place via special communication routes called channels. They connect gates of the parts (**b1**, **b2**, **p**) with the state machine of the container or with gates of the container (**g**, **g1**, **g2**). The block type **B** defines the gates **b1**, **b2**, while the process type **P** defines the gate **p**. When the block type **CompositeBlock** is used for the

definition of instances, the gates **g**, **g1** and **g2** will in turn be used as connection points. Because the part agents are in fact agent sets with a maximum number of five (**BlockPart**) and two instances (**ProcessPart**), there is a maximum overall number of eight concurrent elements at this level (including **CompositeBlock**'s state machine).

Although the decomposition structure is static (i.e. specified as part of the type definition), the number of contained agent instances within a composite may vary over time. Contained agent instances can be created by the container's state machine as well as by some other contained agent instances. Destruction of an agent instance can only be performed by the instance itself.

2.2. Alternate Decomposition

In contrast to concurrent decomposition by means of block agents, alternate decomposition by means of processes implies that every contained process instance executes its transition to completion. This means that in an agent with alternate decomposition only one instance is executing a transition at a time. Alternate decomposition is expressed similar to concurrent decomposition: *Fig. 2* is a sketch of a process type with contained alternating processes. The only visual differences from block types are the keyword (**PROCESS** instead of **BLOCK**) resp. symbol.

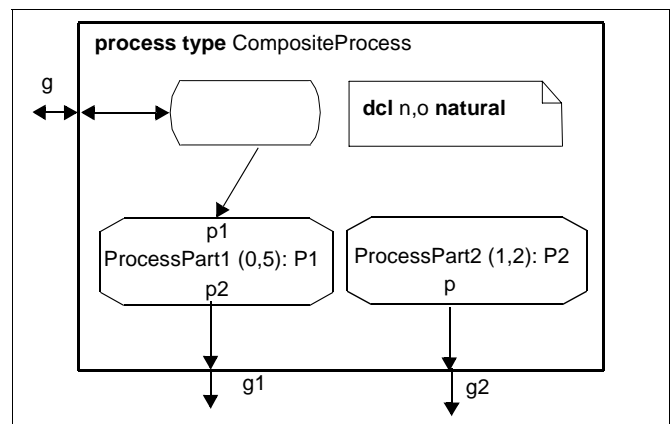


Figure 2. Alternating decomposition of a process.

Due to the execution semantics the contained agents in a process have to be process agents, whereas block agents may contain both blocks and processes.

2.3. Data Sharing.

Variables of a composite agent are shared by the state machine of the container and by all contained agents.

As specified in the sketches in *Fig. 1* and *Fig. 2*, only the state machines of the composite agents have access to the variables. The reason is that the state machines are defined in the same name spaces as the variables, while the block and process types of the contained agents (**B**, **P**, **P1**, **P2**) are defined in some scope units outside the scope units of the composite

agents, e.g. packages. Packages in SDL are name spaces where typically general, related types are defined so that they can be used in different systems.

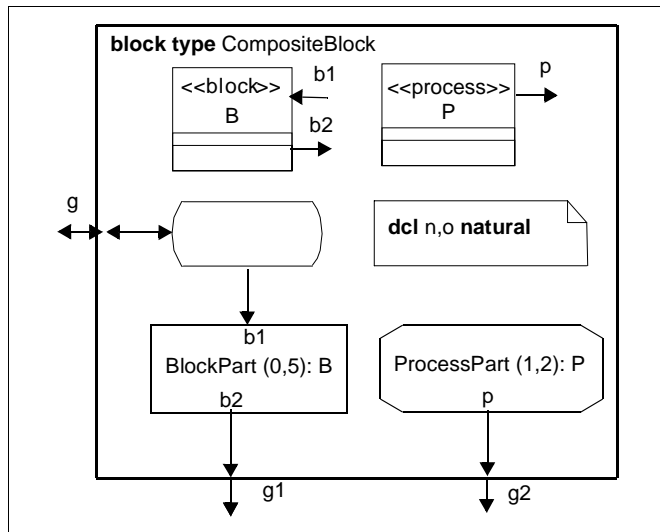


Figure 3. Composite Block type with local type definitions

An agent (and an agent type) is also a name space. It may contain not only instance set definitions and variable declarations as illustrated in Fig. 1 and Fig. 2, but also local block and process type definitions. In Fig. 3 the types of the contained agents are defined locally to the composite agent, and thereby the variables of the container agent become visible also to the contained agents. While the instances (in the sets **BlockPart** and **ProcessPart**) are said to be *parts of* the composite instance, locally defined types are said to be *nested* within the definition of the *enclosing* type definition. Nesting is no prerequisite for composition: the types **B** and **P** do not have to be defined locally to the type **CompositeBlock** in order to specify part objects of these types. Nesting combined with composition provides, however, for part instances to have access to the attributes of the composite instance.

Due to the different type of execution model, there are some important differences between data sharing in a concurrent and in an alternate decomposition.

Concurrent Decomposition. All contained agents have equal rights to read and write a global variable. It is guaranteed, that there are no simultaneous read/write operations, as the reading/writing is performed by the state machine of the owner (by execution of implicit get-/set-procedures). On the other side, due to the concurrent nature of the agents it can not be guaranteed for a single agent, that two subsequent access operations to such a variable will return the same value.

Alternate Decomposition. Variables of a container can be considered as truly shared variables with direct access. The alternate execution of the transitions of the contained processes and the run-to-completion semantics guarantee not only the absence of read/write conflicts between different processes, but also the property that two subsequent read

operations in the same transition to such a variable will return the same value. Only the currently executing process may change the value of such a variable.

An agent with variables and contained agents but no specified state machine has an implicit state machine handling the access to the variables.

No special syntactic constructs are needed in order to access a global variable from within a contained agent. Due to the visibility rules it can be used in the same syntactic way as a locally defined variable.

2.4. Dynamic Creation of Composite Agents

It is possible to create composite agents dynamically. The creation of a composite agent is a three-step procedure:

- The container agent with its state machine is created.
- The appropriate initial number of agent instances for all contained agents are created.
- The start transition of the container and the start transitions of the contained agents are executed.

In case that the contained agents are again containers, the first two steps are repeated on that level too before any start transition is initiated.

Stopping of a container results in a switching its state machine in a special „zombie“-state. The state machine does now only handle internal read/write request for the variables of the container, all other communication requests by signals or remote procedure calls are discarded. If all contained agents have stopped, the container itself will cease to exist.

2.5. Inheritance of Agent Types

Agent type definitions can be arranged in an inheritance hierarchy. For the type symbols this is specified as in Fig. 4. This is a partial specification of the two subtypes: the fact that they are subtypes of a common block type. The complete type definitions are provided in type diagrams, see Fig. 5, 6 and 7.

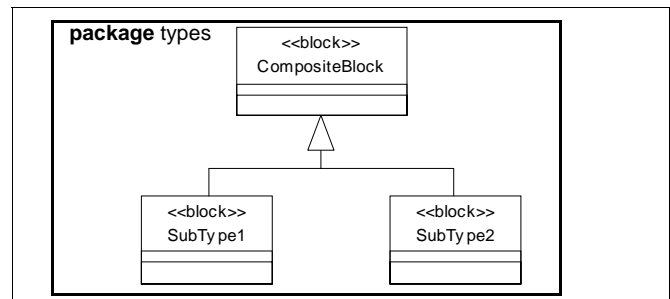


Figure 4. Inheritance specification for agent types

A subtype inherits all structural and behavioral elements, may add new contained agents, and may redefine (override) virtual properties. If the **ProcessPart** of **CompositeBlock** shall be redefineable, then the type **P** of **ProcessPart** is specified to be a *virtual* process agent type, see Fig. 5.

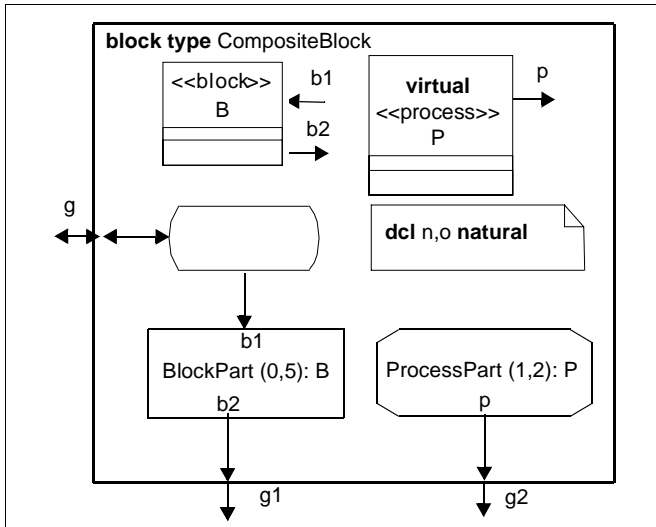


Figure 5. Virtual process type

If we look into the diagram defining the **Subtype1** block type (Fig. 6), one other process set (**ExtraProcessPart**) and two channels connecting this set with the environment and the existing **ProcessPart** have been added. Using the virtual con-

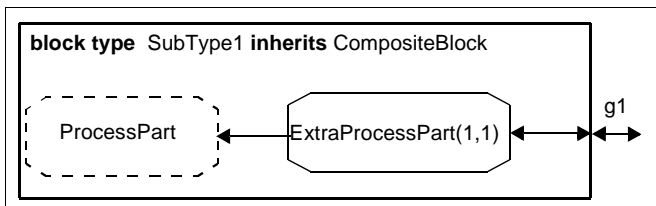


Figure 6. Inheritance of types

cept it is even possible to change the inherited elements in a subtype definition. **Subtype2** (Fig. 7) has the same structure as **CompositeBlock**. However, **ProcessPart** is here based on the

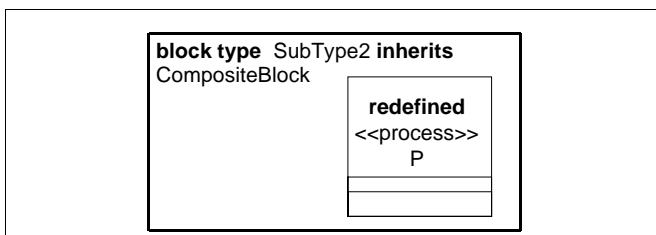


Figure 7. Redefinition of the virtual type P

redefined type definition of **P**. It may therefore have a different behavior as well as a different internal structure than the original process type **P**.

2.6. Communication

In order for two agents to communicate with each other, both have to be connected by a channel or a chain of channels. A channel is a reliable uni- or bidirectional medium with optional delaying properties. Channels may be specified

explicitly or may be derived from the connection points (gates) and the implemented interfaces of the agents. Communication may take place as asynchronous signal exchange, synchronous remote procedure calls or as access to public (exported) or shared variables.

No special rules are prescribed for the communication with a composite agent. The specifier can completely hide the internal structure by connecting all external channels to the agent's state machine. The state machine would in this case dispatch or delegate signals and remote procedure calls to the contained agents and send the results back to the caller. The other extreme is to connect the contained agents with bidirectional channels to the containers interaction points. In this case a communication between an outside agent and a contained agent is possible without any involvement of the containers state machine.

3. Behavioral Decomposition

The behavior of agents is specified by means of extended finite state machines. A state machine is a connected graph of states and transitions. Transitions are triggered by external stimuli (signals, conditions, remote procedure calls). A state machine is always waiting for a stimulus in a state or performing a transition. During a transition a sequence of actions may be performed, including the sending of signals and the creation of agents. This implies that the underlying semantic model of a state machine is a Mealy automaton.

Three main ways for a decomposition of behavior are provided by SDL-2000:

- Decomposition of the full state machine into state partitions (state machines) handling disjunct sets of stimuli.
- Decomposition of a state into sub-states.
- Decomposition of a transition into procedures.

In contrast to structural decomposition by agents, which always implies also a behavioral decomposition, the behavioral decomposition does not imply a structural decomposition.

3.1. State Partitioning

State machines can be directly specified by a state-transition graph or by a decomposition into *state partitions*. Such a decomposition is obtained by dividing the set of all input stimuli into disjunct subsets.

Each of these subsets is then assigned to a single state partition. The behavior of such a partitioned state (machine) is the combination of the behaviors of its state partitions. The execution semantics is alternation of transitions (run-to-completion), i.e. exactly one state partition in a state decomposition is performing a transition at any point in time

An example of a state partitioning is given in Fig. 8 The state machine is partitioned into two state partitions, one (**Controller**) handling the signals **newgame**, **endgame** and **result**

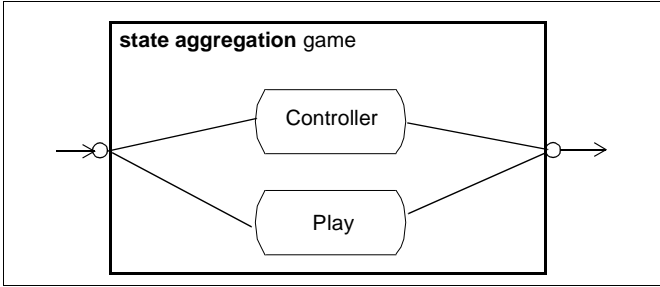


Figure 8. State partitioning

and another one (**Play**) handling the signal **probe**. These states are directly defined by state-transition graphs as part of state diagrams (Fig. 9).

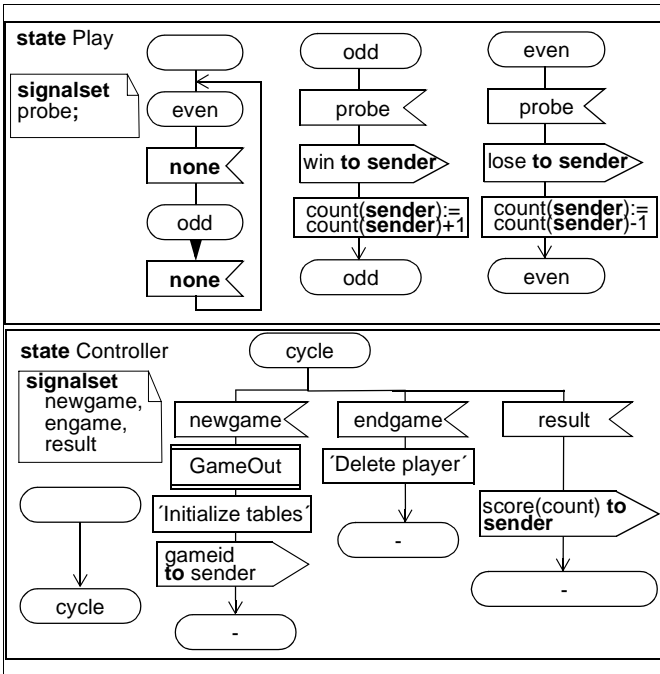


Figure 9. State partitions

State partitions are created together with their containing state machine. It is not possible to create state partitions dynamically.

3.2. State Decomposition

Whereas state partitioning uses the set of input signals, which is the first element of the Mealy output function, as basis for the decomposition, *state decomposition* is based on the second element, the states. State decomposition in SDL-2000 resembles the concept of hierarchical states known from Harel's state charts. A sketch of a state decomposition is given in Fig. 10, Fig. 11 and Fig. 12. It is a partial specification of an automatic teller machine (ATM). It also illustrates the combination of state machine and internal contained agents. In Fig. 10 it is specified that each ATM consists of a state machine and two block agents (controlling the panel and

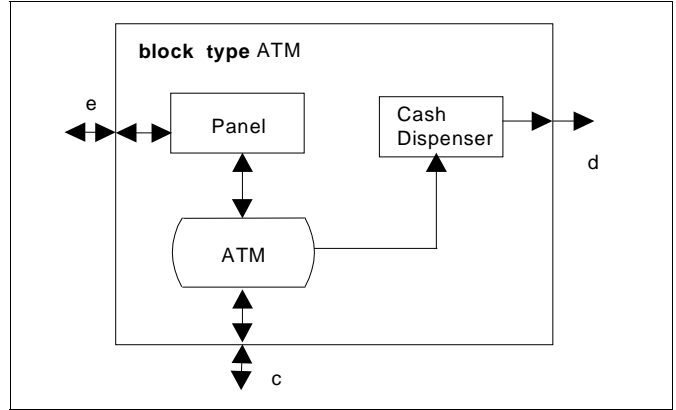


Figure 10. Composite States

the cash dispenser). The ATM state machine is sketched in Fig. 11. The composite state **ReadAmount** is referenced as part of the ATM, and it is defined in Fig. 12. Note that an interface in terms of connection points is a notion that applies to both agents and state machines. While the gate connection points

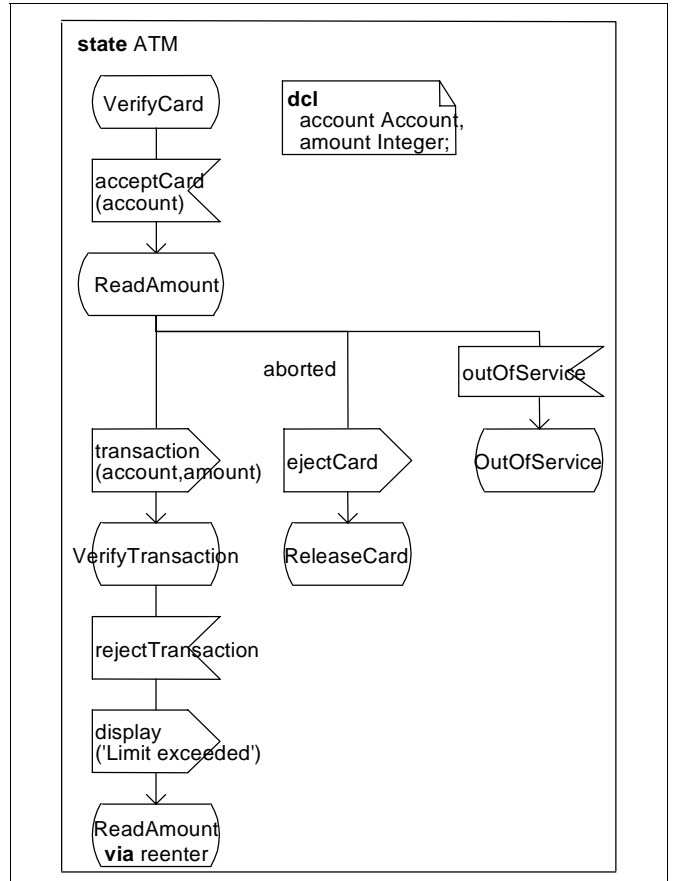


Figure 11. State ATM with composite state (ReadAmount)

for agents (e, c, d in Fig. 10) specify which signals may be communicated through the points, the *state connection points* for composite states (**reenter**, **aborted** in Fig. 11 and Fig. 12) specify possible transitions through the points.

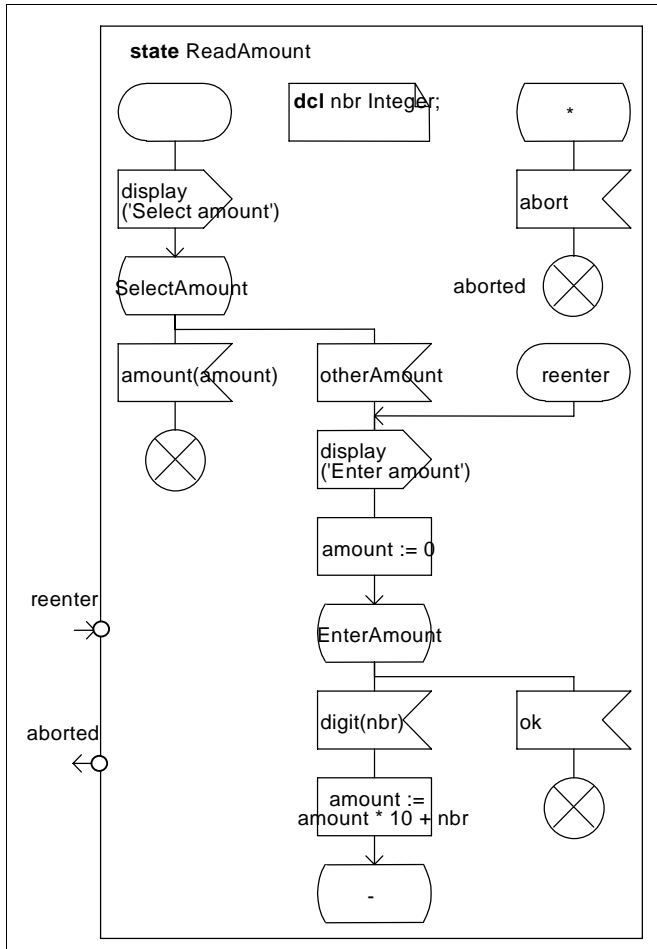


Figure 12. State diagram for composite state ReadAmount

4. Relation to SDL-96

As mentioned above agents in SDL-2000 consists of three main parts:

- data in terms of variables,
- behavior in terms of a state machine, and
- an internal structure of contained agents and their connections.

Because all three parts of an agent are optional, the SDL-96 concepts of block and process are two special cases of an agent:

- An SDL-96 block is an agent, which has no state machine nor local variables, parameters, etc.; it only contains processes.
- An SDL-96 process is an agent, which only has a state machine, optionally local variables, parameters, etc., but no contained agent definitions.

The alternating execution of the processes in a process resembles the concept of services from SDL-96, with the following differences:

- Each process instance within a process has its own input queue and its own unique process identifier, while ser-

vices shared these features with their containing process.

- Process instances within a process may be dynamically created and destroyed, while services live and die with their containing process.

The valid input signal sets of the services of a process had to be disjoint; that is not the case for processes in processes. State partitions replace the concept of service from SDL-96.

5. Related Work

Many object-oriented analysis and design techniques base the description of behavior similar to SDL on extended finite communicating state machines. This section gives an overview over decomposition mechanisms in the two modeling languages UML and ROOM (Real-Time Object-Oriented Modeling, [6]).

5.1. Decomposition techniques in UML

UML provides different means for structural and behavioral decomposition. The logical structure of a system in UML is specified by static structure diagrams using classes and relations between classes, especially aggregation and composition relations. Component and Deployment diagrams on the other side specify the physical structure of the system. The relationship between the logical elements and the physical representation is expressed by residence meta links. While SDL provides class definitions (type definitions) that specify the internal structure of objects of the container class, UML emphasizes the relations between the class of the container and the classes of the part objects. Structures of objects are mainly used as temporary snapshots in object diagrams and in collaboration diagrams.

An aggregation is a whole/part relationship between two classes with composite aggregation being a strong form which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. The composite object has the life-cycle control over its parts. On destruction of the composite object it has to destroy its parts or to force their migration to another composite object. The physical distribution of objects is modeled by assigning classes to components (as residents). In a further step components may be distributed over physical devices (nodes) i.e. processors. Unfortunately UML does not specify any well-formedness rules relating the distribution of objects onto components to the aggregation relationships in the static structure diagrams.

In contrast to SDL the semantics of the derived behavior of a composite object or of a component is left open. It is up to the user to specify what kind of execution mechanism (concurrent, sequential, alternate) applies. Also the rules for the creation of and the communication with composite objects have to be specified explicitly in a behavior diagram.

As in most object-oriented modeling techniques behavior in UML can be specified by state charts. A single state of a state machine can be decomposed into (orthogonal) concurrent sub-states. The execution of transitions follows always a run-to-completion semantics. Due to this fact, and to the orthogonality of concurrent sub-states, an interleaving of concurrent transitions is always possible. Although UML has special annotations to specify the rules for a concurrent external access to an object (call of operations), the semantics guide does currently not contain rules for the access to shared attributes of a container. The three ways to handle concurrent operation calls are:

- sequential: external synchronization is required
- guarded: calls are sequenced by the called object
- concurrent: all calls are executed simultaneously.

A detailed execution model is not (yet) given in UML.

5.2. Decomposition techniques in ROOM

The ROOM notation has the concept of concurrent entities called actors. An actor may have both a state machine and a decomposition into sets of connected actors. Actors are defined by actor types. ROOM does not support nesting of definitions. This means that actor types can not be locally defined to an actor type. Actor types are defined in packages. An actor can be instantiated as part of another actor, but during run-time the actors are flattened out as completely encapsulated objects. All forms of communication between actors are via ports and bindings (connecting two ports). There are no global shared variables. Variables in the container actor are thus not visible in the contained actors. The implication of this is that a container actor is more or less just a container. The structure of contained actors can not reflect the effect of their execution in the common container actor, so the reason that they are contained in the same actor is only for generation of actor structures.

The behavior of actors is specified by a variant of Harel's state charts. To control the complexity of the behavior description of an actor, ROOM supports the same state hierarchy concepts as SDL:

- A state at one level of abstraction may be refined into a complete state machine at a lower level. The current state of an actor is described by a nested chain of states (state context).
- Each nested state represents a separate context (scope unit) with its own optional set of data variables.
- A so-called group transition is the concept in ROOM to model the common behavior (set of transitions) that is independent of any sub-state at the next conceptual level.
- ROOM supports the history concept.
- State entry actions are optional detailed actions that are executed whenever the associated state is entered, independent of which transition this state was addressed. Similar state exit actions are supported.

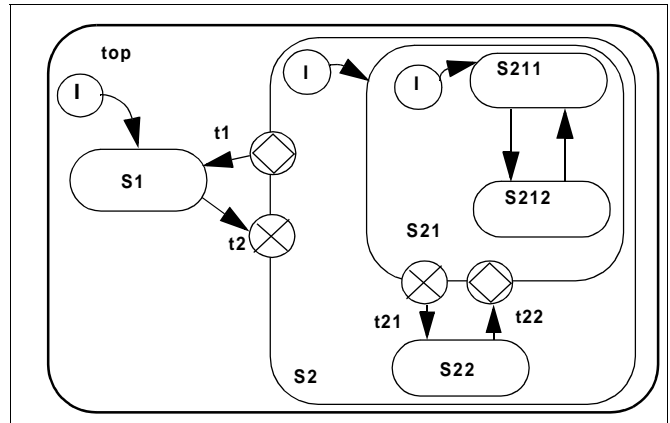


Figure 13. Composite state in ROOM

The example in Fig. 13 shows a state hierarchy with the superstate **top** which is refined by the states **S1** and **S2**. State **S2** is composed by the sub-states **S21** and **S22**. The leaf sub-states of **S21** are **S211** and **S212**. Transitions which return to the history state of a composite state is a transition segment that terminates on the outer border of a composite state (here **t21** and **t2**). Assume that the current state is **S212** when the stimulus for **t1** is fulfilled by an event. Then transition **t1** will be applied because it is a group transition (whose execution is regardless in which sub-state the actor is). In that case transition **t2** will force the behavior back to the context **S12-S21-S2-top**.

Besides various syntactic differences the main difference between SDL and ROOM is that SDL allows type and subtypes of composite states with the following mechanisms:

- inheritance of states and transitions
- redefinition of virtual transitions
- redefinition of virtual states/sub-states
- context parameters in order to be usable in different systems.

An important aspect in favor for ROOM is its distinction between high-level and low-level behavior description. This allows to separate out implementation dependent control and communication activities.

Compared to UML or ROOM the semantics of structural decomposition in SDL is more rigid. Rules for creation, destruction and communication are directly integrated in the language and free the specifier from overspecification. Nevertheless, due to the provision of different semantic variants the rigid semantics does not imply severe restrictions on the applicability. An overview over the structural and behavioral composition/decomposition concepts in the three different languages is given in Table 1. Table 1..

In the area of structural decomposition SDL-2000 provides the same expressive power as UML, but adds both a clearer semantics and the feature of nested types and the implied access control by the container.

Table 1. Comparison of decomposition concepts

	SDL	UML	ROOM
Structural Decomposition	y	y	y
concept	Contained (and possibly nested) Agents	Aggregation/ Composition	Contained (but not nested) Actors
derived behavior semantics	concurrent or interleaved	no semantics predefined	concurrent
data sharing	y	y*	n
access to contained elements	controlled by gate/interface	no restrictions	controlled by port/protocol
migration of contained elements	n	y	n
separate class/type and structure definition	y	n	y
nested class/type definitions	y	n [†]	n
Behavioral Decomposition	y	y	y
nested states	y	y	y
concurrent sub-states	y	y	y
state types/classes	y	n	n

*. No semantics defined.

†. Allowed by Semantics Guide, but no notation available.

In SDL the definition of types (classes) and the use of instances or instance sets of types in a structural decomposition is strictly separated whereas UML expresses decomposition as an aggregation/composition relationship at the class level. SDL composition does not allow part objects to migrate to other composite objects. Concerning behavioral decomposition, all three techniques are based on Harel's state charts. The main additional SDL feature here is the ability to define separate state diagrams, state connection points and state types. The notion of state types is not described in this paper, but can be found in [15].

Although the semantics of the decomposition concepts in SDL-2000 can be entirely explained using a transformation to basic SDL, a direct semantic foundation is the preferred solution. The current work towards a new formal semantics for SDL [13] based on the ASM concept (Abstract State Machine), provides such a direct semantic foundation.

6. Conclusion

In most object oriented languages structural and behavioural decomposition are defined as separate concepts with no or few combinations or relations to each other. Concepts for structural decomposition are e.g. aggregation and composition. Concepts for behavioural decomposition are e.g. composite states for state machine specified behaviour and

procedures as part of transitions. While aggregation and composition are specified by special associations between classes, state machines are specified for individual classes. Implications of a structural decomposition on the behavior and of behavioral decomposition on the structure are often defined rather vague.

SDL has had its primary force in structural decomposition. Object-oriented concepts as classes (types in SDL) and inheritance were introduced into the language in 1992.

This paper has presented the combined structural and behavioural decomposition mechanisms of SDL-2000. It has been demonstrated that it is possible for a language to provide (with formal semantics) combined mechanisms like objects with both state machines and internal structure of connected objects, composite states with well-defined interfaces, mechanisms for inheritance of such combined state machine and internal object structure. Moreover, the language also provides mechanisms for the dynamic creation of such composite structures, and mechanisms for the communication and data sharing between the contained elements. Combining the new concept of agent with the concept of *interfaces* makes SDL well suited for the design and development of applications in the context of distributed systems and middleware platforms (e.g. CORBA)

REFERENCES

- [1] ITU-T: Rec. Z.100 - *SDL - ITU-T Specification and Description Language*, ITU-T, Geneva, 1996.
- [2] ITU-T: Rec. Z.100 - *SDL-2000 - ITU-T Specification and Description Language*, ITU-T, Rec. Z.100, Geneva, November 1999.
- [3] ITU-T Rec. X.901 | ISO/IEC 10746-1: *Open Distributed Processing - Reference Model Part 1*, ITU-T '95
- [4] ITU-T: Rec. Z.109 - *SDL Combined with UML*, ITU-T, Draft Standard Proposal, Geneva, 1999
- [5] OMG: *The OMG Unified Modeling Language Specification* (UML V1.3), OMG, 1999.
- [6] B. Selic, G. Gullekson, P. T. Ward: *Real-Time Object-Oriented Modeling* (ROOM), Wiley&Sons, 1994.
- [7] O. Lehrmann Madsen, B. Møller-Pedersen, K. Nygaard: *Object Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.
- [8] B. Møller-Pedersen: *SDL/UML Alignment: Composite States*, Ericsson AS, 1998
- [9] B. Møller-Pedersen, E. Holz: *Dynamic Block Generation & Sharing of Data among Processes*, ITU-T Temp. Document TD-25E, Geneva 1999.
- [10] E. Holz: *Active Entities*, ITU-T Temporary Document, TDT-620, Toulouse, 1999.
- [11] E. Holz: *Application of UML in the SDL Design Process*, Proc. of SAM98, Berlin, 1998.
- [12] E. Holz: *Application of UML within the Scope of new Telecommunication Architectures*, GROOM Workshop on UML, Mannheim, Physica-Verlag, 1997
- [13] U. Glässer, A. Prinz: *An ASM semantics for SDL*, Proc. of the Ninth SDL Forum (SDL'99), Montreal, 1999.
- [14] D. Harel: *Executable Object Modeling with Statecharts*, in IEEE Computer, Vol.30, No. 7, pp 31-42, 1997.
- [15] B. Møller-Pedersen, D. Nogva: *Scalable and object Oriented SDL State(chart)s*. FORTE/PSTV'99, Beijing.