11 Property Modelling

Property Modelling

Introduction

Property modelling is the activity related to making the property model, one of the two halves of our description models. The domain model, the application model, the framework model and the architecture model all have two halves: an object model and a property model. Together these two halves make up a full model. The reader is referred to the example to see how concrete descriptions together form a unit of description.

While the object model eventually becomes the imperative definition of the system, the properties define the requirements and the capabilities of the system relative to its surroundings. Properties are normally more fragmented and partial. Often property descriptions are not as formal and precise as the object descriptions. This is, however, not a necessary characteristics of property models, but more a consequence of the general desire to express more than the formal languages can express.

Property models are expressed in a variety of languages. Prose and structured prose are popular for some descriptions while temporal logic and sequence diagrams (MSC) are popular in others.

In this theme we shall give the most common characteristics of properties in What is Property Modelling? (p.11-3) and then we want to go into some dimensions which also give more insight into what is covered by the concept *properties* in Dimensions of the property concept (p.11-6).

Having achieved a better understanding of what we want to describe, it is reasonable to analyze by which means the description can be expressed. In Some Property Languages (p.11-10) we go through a set of different groups of languages each with its own strongholds and shortcomings. This section is of a slightly more theoretical nature and the readers who are more practically inclined may skip this section until their desire to learn more becomes unbearable.

Then for the more practically inclined we present The SISU Property Modelling Technique (p.11-26). This section can be read and used without thorough knowledge of the other sections. We present our overall strategy for property modelling which can be characterized by the slogans: Service orientation (p.11-26) and Role orientation (p.11-27).

Our dialectic approach to refinement of property models is shown in The dialectics of refinement (p.11-29) where we highlight the idea that to make models more precise is *not* the same as making models more detailed. In order to make the concepts more easily understood we show how these approaches can be used in an example Example: Access Control: Change PIN (p.11-31). The section is wrapped up by Strategies for property modelling (p.11-38) and The art of Formalizing (p.11-39) which in a stepwise manner give the engineer advice on how to go about performing the property modelling.

On the constructive use of MSC (p.11-45) is the section which brings the property modelling together with the object modelling. The ultimate strength of our approach can only be reaped if the two perspectives object orientation and property orientation are jointly persued. In this section we show alternative combinations of object- and property orientation. As a technique for “requirement engineering” we show how SDL skeletons can be mechanically produced from MSCs.

What is Property Modelling?

According to American Heritage Dictionary “property” means “a characteristic trait or quality”. The properties characterize of course the objects which we identify in the object modelling (see theme on object modelling). It is, however, not always the case that the object model has been created before the property model. During the identification of the objects, properties become clear, and during the description of properties, the objects and their interaction is established.

What characterizes a property description compared with an object description? Here are some common properties of property descriptions:

- Property descriptions cover specific aspects (p.11-3) (traits, qualities).
- Property descriptions may overlap and underlap (p.11-3).
- Property descriptions are often declarative rather than imperative (p.11-4).
- Property descriptions supplement object descriptions (p.11-4).

Every language has its “universe of discourse”, that is the set of concepts which makes up its semantic base. SDL expresses signal exchange between objects, with success, but fails to describe the splendor of an Ibsen drama or the humidity of the air in Trondheim. Property notations are often used to supplement the object model in areas it does not cover.

Property descriptions cover specific aspects

Such specific aspect may be:

- *liveness properties*: something good will eventually happen;
- *safety properties*: something bad will never happen;
- *possibility properties*: something which might happen;
- overview of *functionality* (functions and function lists, functional roles);
- focus on *interaction* (use cases, Message Sequence Charts (MSC) diagrams);
- *capacity* and *timing* constraints;
- *physical constraints*: temperature, humidity, power consumption, concrete interfaces,
- *other* not so easily formalized properties: modifiability, security, error handling, user friendliness and price

Property descriptions may overlap and underlap

Since property descriptions concentrate on particular aspects, it is quite natural that the set of property descriptions does not cover the model without mutual overlap/underlap.

The goal of the property descriptions may not be to give a complete model description, but to give descriptions which are sufficient for the verification and validation of certain important characteristics of the system.

As an example we are used to accepting that the MSC document will not comprise a description of all traces possible in the SDL model (object model), but rather defines possibility properties.

It is also the case that overlap between different property descriptions may allow consistency checks provided that there exists a common semantic model. This may not always be the case. The official semantic models of MSC and SDL unfortunately do not have a common basic theory so consistency checks between MSC descriptions and SDL descriptions must take as their starting point formal semantics which differ from the official ones. MSC formal semantics is defined in process algebra (See Process algebra (p.11-12)) while SDL semantics is described in MetaIV – a variant of VDM (See Predicate logic (p.11-16)).

Property descriptions are often declarative rather than imperative

While our ultimate object model in SDL may be seen as a complete imperative description of the system, property models are often declarative, meaning that they express something which either holds or does not hold in the model.

First order predicate logic is an example of a declarative language for property description. The property has the form of a predicate which is either *true* or *false* in a given model (at a given time).

Declarative descriptions are often formulaes which relate the objects of the system in more or less intricate ways. From the literature of structured programming we are familiar with pre- and post- conditions, and with invariants.

Invariants are conditions which hold at certain times and/or at certain program points. Such conditions represent properties which describe stability and structure in executions, but do not prescribe the execution actions.

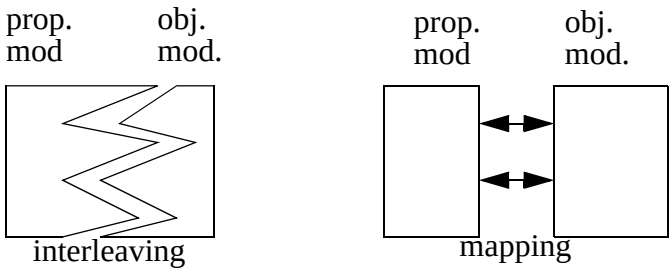
An example is a statement like: “Response shall be given within 15 ms in 90% of the cases.”

Property descriptions supplement object descriptions

We have some property descriptions which may be some MSC diagrams, and we have an object model which is an SDL model. If we have means to “align” the two descriptions such that we know that they are supposed to describe the same reality, we may use logic to verify whether the properties (stated in the MSCs) are satisfied by the object description. In this way the property description supplements the object description in Figure 11-1 (p.11-5).

Figure 11-1: Property model supplementing object model

Open figure



Sometimes the property descriptions may even express properties that are not evident from the object model. This happens when we describe performance capacities and timing constraints which formally cannot be described in (say) SDL.

Dimensions of the property concept

As we have seen from What is Property Modelling? (p.11-3) there are many different aspects of the property concepts. In this section we shall go into more ways to characterize the property concept.

Relating to the existence of the system

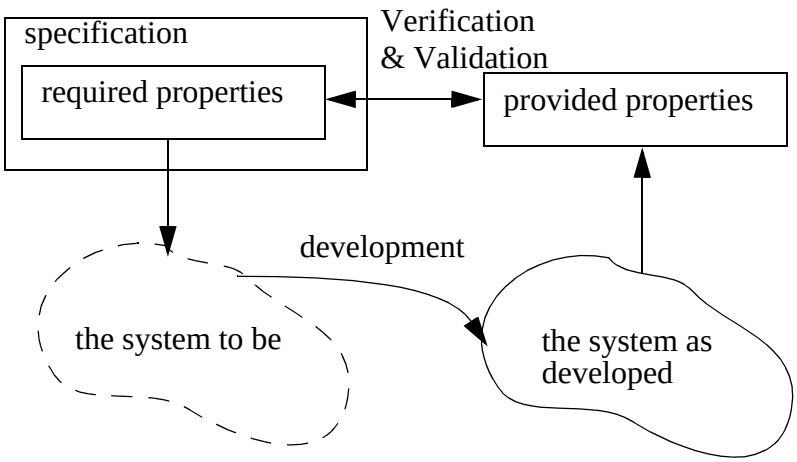
We may distinguish between *required* properties and *provided* properties. The required properties are often made and expressed before the object (to which the properties relate) is actually made, while the provided properties by necessity relate to an existing system.

Still the provided properties may not necessarily be provided by some implemented system. We may also say that an SDL system provides properties.

The required properties also relate to the development process activity “analysing requirements”. The required properties are sometimes included in documents which make up the contract for the development job.

Figure 11-2: Required and provided properties

Open figure



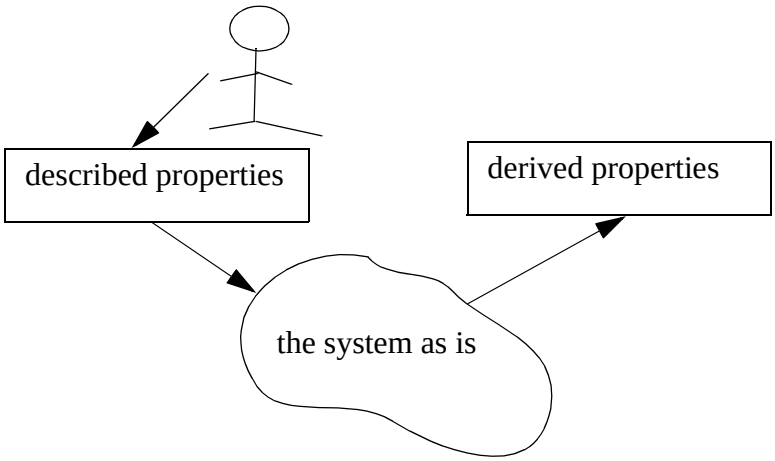
In Figure 11-2 (p.11-6) we try to illustrate the relations between required and provided properties and the system which is being developed.

Relating to the origin of the properties

The *described* properties are made by a system developer or a team of system designers, while the *derived* properties have their origin in the system itself or in a complete specification thereof. Such a complete specification may be the SDL object model of the system.

Figure 11-3: The origin of the properties

Open figure



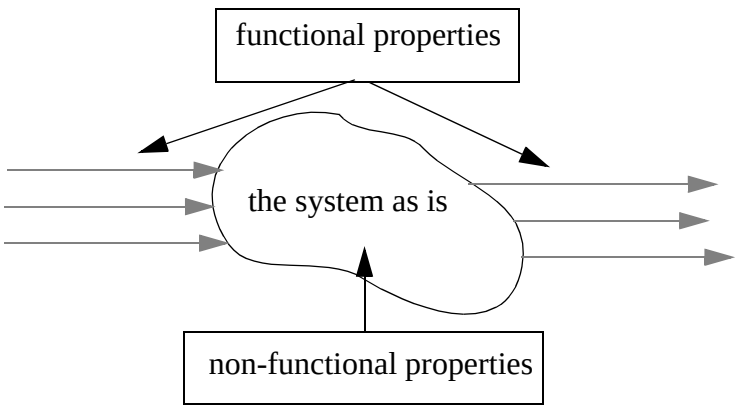
In Figure 11-3 (p.11-7) we illustrate the position of the developer relative to the described and derived properties.

Relating to whether the properties define functionality

Some properties describe what the system *does*, i.e. which output it gives as a result of which input. When we focus on the results of the behavior of the system, we focus on its functionality. If we focus on the color of the physical components, we focus on non-functional aspects, as the color most probably will have no effect on the operating of the system[1]. The non-functional properties relate to what the system *is*, rather than what it *does*.

Figure 11-4: Functional and non-functional properties

Open figure



1. This may of course be different on certain systems where color play an important role, but normally the color has a visualizing effect only.

In Figure 11-4 (p.11-7) we illustrate that the functional properties in principle relate to the effects of the system as modelled as functions from input to output (See Focus (p.11-19)). The non-functional ones on the other hand relate to the aspects of the system which characterize the system as a whole and independent of the specific input. This does by no means indicate that the non-functional properties are unimportant!

Functional properties

Functional properties can be divided in categories related loosely to the Application reference model which has three parts: interface given, domain given, and system given.

- *interface* properties, related to interface given aspects, such as protocol properties of the channels;
- *general* properties, related to the domain given aspects, such as freedom from deadlock and absence of unreachable transitions;
- *sevice* properties, related to system given aspects, such as how the access control system should react upon the magnetic card in its card reader.

Non-functional properties

The non-functional properties are very varied since they relate to everything but the operation of the system. Even though the properties are non-functional they may be fairly closely related to the functional ones. Some users will say that if an interactive system gives no response within 10 seconds it cannot be said to function. In our terms response times are considered non-functional, but there is no doubt that performance may be very important.

Furthermore the non-functional properties may be just as susceptible to formalization as the functional ones. Response times can be measured in seconds and reliability by MTBF[1]. The non-functional properties may also be very closely attached to the functional ones. Response times may be associated with specific services or operations; capacity may be related to only parts of the system.

We may classify the non-functional properties:

- *Performance* (response times and capacity)
- *Reliability* (how often can one expect an error recovery?)
- *Security* (what are the protection mechanisms?)
- *Physical* (humidity constraints, temperature, color etc.) which again may be subdivided into property requirements to the environment and of the constructed system itself.
- *Modifiability* (how easily can improvements and extensions be made?)

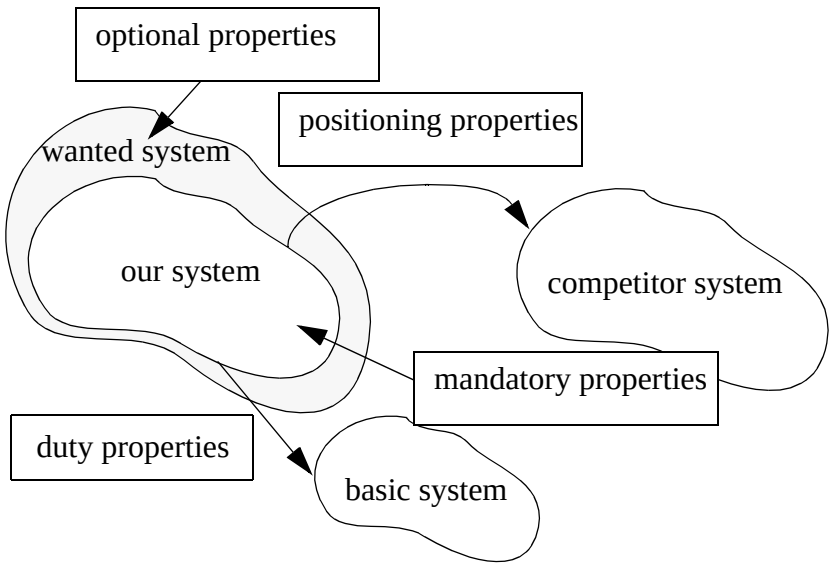
1. MTBF=Mean Time Between Failures

Relating to market of product

A system is of little use if it is not sold or used. To be sold it is often necessary to highlight properties which are different from the competitors (supposedly better).

Figure 11-5: Market oriented properties

Open figure



In Figure 11-5 (p.11-9) we illustrate that some of the properties of a system are related to the position of the product in the marketplace. Some properties may be called *duty* properties meaning that those are the properties which the user will consider essential in order to take your product into consideration at all. Once your product is taken into consideration, it will become important how it compares with other products. The *positioning* properties are those which place your product in the competition.

While the distinction between duty and positioning properties relates to the choice of your product by a customer, the distinction between *mandatory* and *optional* properties relates to the development of a product which in some way has been chosen. The mandatory properties must be fulfilled to adhere to the contract while the optional properties are “nice to have” features which the customer may have to pay extra to get.

For our example *Access Control* system, duty properties must be the ability to control doors subject to access by an electronic card. Positioning properties could be the price of a given system, the compactness of the panels, or the security of the central unit. Mandatory properties of the Access Control system are related to the access control, while optional properties could be related to PIN changing facilities, backup procedures etc.

Some Property Languages

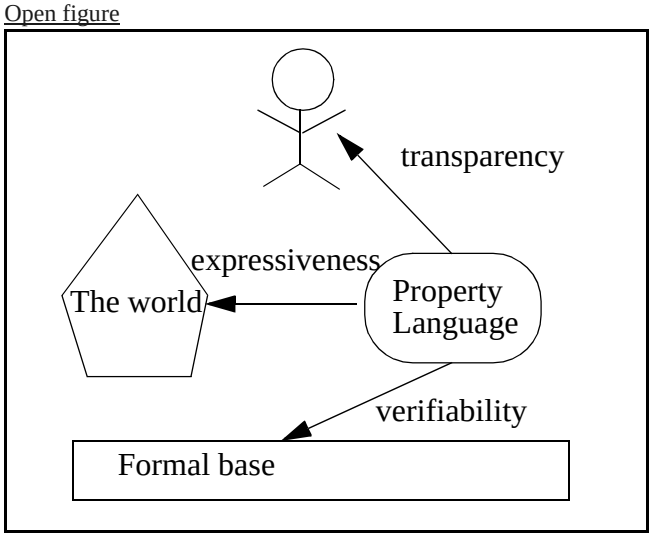
Property languages are like other languages, they are best when they are used to describe what they are designed for. However, it is sometimes the case that the languages can be used for a wider spectrum of purposes than they were originally designed for.

Combes et al. [41] have shown how MSC can be used to express statements of liveness and safety which normally are described by temporal logic.

When evaluating languages we shall focus on three aspects of a language:

Figure 11-6: .Property Languages

Open figure



Firstly we must consider whether statements of the language can be understood by humans (in particular those needing to understand it). We call this transparency. Still simple languages are of little help if it is impossible to express those aspects which are crucial for the success of the system. We need the language to be expressive such that the important aspects of the system can be described accurately. Finally we want the descriptions to be precise such that no ambiguities can occur. This is best achieved by the existence of a formal semantics base such that descriptions can be verified against a model (e.g. defined in some other language). These dimensions are shown in Figure 11-6 (p.11-10).

Completeness is also an issue. Does the property description describe the total set of situations? Property descriptions will normally serve its purpose quite well even though they may not be complete. We are used to MSC documents which normally describe only a small portion of the possible scenarios, and invariants which describe only formulaes on a subset of the variables. Still there is no specific benefit to be reaped from being incomplete when completeness can be achieved. But it may not be trivial to assess whether a given set of formulaes form a consistent and complete model

The question of completeness also relate to the “universe of discourse”. If we have used MSC to express properties, then aspects of humidity or physical security are not within its realm. Not even in the constructive object model of SDL would such properties be described. Still we would claim that the SDL model is complete, meaning that it

describes all possible behaviors of the system projected onto its universe of discourse. On the other hand, an MSC document will normally not be claimed to describe all possible behaviors even if we project onto the universe of discourse. MSC normally describes a set of *some possible* behaviors, while SDL in principle describes the set of *all possible* behaviors.

Prose

Textual prose, possibly accompanied by informal illustrations, is always an important means to describe a system. Prose has its strongholds when the specification should be readable among many different kinds of people: managers, engineers and marketing personel.

To use prose may also be motivated by the lack of proper tools for a more formal description, or simply the lack of a proper language.

Prose is also heavily used as comments inside the other more formal descriptions.

Service lists

Since our methodology may be said to be service oriented (see Service orientation (p.11-26)), it is important to identify the services as early as possible. The services may be classified for different purposes, e.g. for sales (some functions are related in deliverables) or for property tracing.

The identified concepts in the dictionary and in the conceptual model shall be used in the formulation of the functions.

Function specifications contains:

- Function name
- Textual explanation
- Relation to object model and how relations and attributes are involved
- Relation to other property models
- Possible constraints on combination with other services.

Industry standard languages

MSC

MSC (Message Sequence Charts) highlights interaction between instances based on messages. MSC is most effective when the interaction sequences of messages between the acting objects (or roles) is of major importance. See our tutorials of MSC-92 and MSC-96.

MSC has no data concepts and therefore values of variables can only be expressed in comments and informal text.

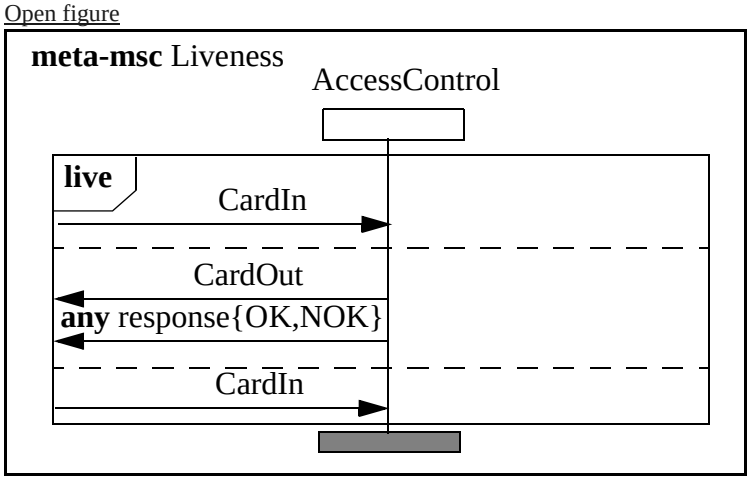
With MSC-92 it is evident that the MSC document cannot be complete. It is reasonable to interpret the MSCs as a well selected set of possible (or impossible) cases. See also our MSC-92 methodology and our MSC-96 methodology.

With MSC-96 the situation is largely changed. With the introduction of more versatile composition mechanisms and MSC expressions, it is possible to become more complete and to express more precisely properties which are often described by temporal logic.

We could model our liveness property “Whenever you put a card into the Access Control system you will get your card and a response back eventually” by the meta-msc construct shown in Figure 11-7 "Liveness in MSC" (p.11-12).

Figure 11-7: Liveness in MSC

Open figure



Meta-MSC (Meta-MSC (p.11-42)) is an elaboration (by us) of MSC-96 where liveness and safety propeties may be expressed relative to another MSC-96 document. In Figure 11-7 (p.11-12) we have introduced the liveness operator **live** which takes three arguments and the wildcard message **any**. The **live** operator is defined such that whenever the first argument is matched (here *CardIn* input to *AccessControl*), then the second argument will happen (here: the *CardOut* will be output and some response given) before the pattern of the third argument happens (here: another *CardIn* is input). The response is decorated with **any** meaning that either “OK” or “NOK” is to replace “**any** response”.

Still with the advent of MSC-96, MSC does not express time and capacity constraints. The events of an MSC are only ordered ordinally along each instance axis. Any real time requirements can merely be expressed through comments or through the use of pseudo-timers. This may not be satisfactory.

Process algebra

Whether the property description can be said to be complete is not only a matter of having enough pieces of description. Sometimes it is also a matter of levels of detail.

For the Access control system, the description of the system as such in relation with its surroundings (context) may be complete, but still it does not suffice in order to be able to create the system. Then we have to decide what should be inside the Access Control system.

Our SDL/MSC method is a technique where the approach to detail is guided mainly by decomposing the objects (instances). With MSC-96 it is possible to approach details in a more process-oriented way. By process-oriented we mean that processes are divided into sub-processes independent of the instances being decomposed.

Farther in this direction are the languages which describe systems as composed of processes. The most popular one in universities and among the more theoretically inclined, is CCS [139]. In the same tradition we find the more industry-oriented language LOTOS [18] which formerly was considered a competitor to SDL.

There are examples of mixtures of SDL and LOTOS where LOTOS has been used as a property language accompanying an object description in SDL. LOTOS was used to specify the interface behavior on the channels more precisely than SDL structure diagrams do.

A LOTOS example (p.11-13) taken from the standard shows the specification of a simple duplex buffer.

Figure 11-8: LOTOS example

Open figure

```
process new_simple_duplex_buffer [in_a, in_b, out_a, out_b]:=
one_time_buffer[in_a, out_a] ||| one_time_buffer[in_b, out_b]
where
process one_time_buffer[inpt, outp] :=
inpt;outp; stop
endproc
endproc
```

SDL as a property language

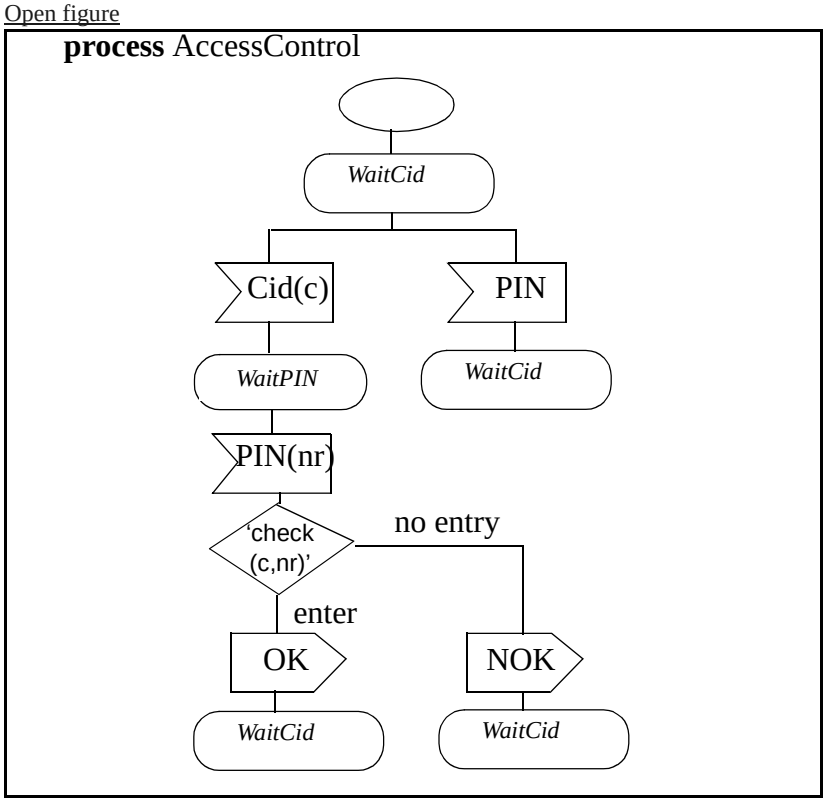
Can SDL be used as a property language? We have presented SDL as the main language for object description. There is no rule against using the same languages for different purposes, but there is no reason to produce two identical descriptions and call one of them the property description.

A state-oriented process view is often practical, also on coarse levels where the object model in SDL shows blocks and their decomposition. Our desire would be also to have a view where states and their transitions were shown. SDL substructures were originally designed to include *both* a state oriented view *and* a block in block view. It turns out, however, that technical problems with the concept often prevent the feature from being actually used.

Even though the internal state of a process (system) may not be observed from the outside the internal state may still be important for the understanding of the process. The internal state is often an implicit cause of the observable behavior

Figure 11-9: .SDL as property language

Open figure



We see in Figure 11-9 (p.11-14) that the whole Access control system can be sketched from a single user's point of view as a simple process. In this diagram we have taken the liberty to simplify the typing of four digits into one single PIN(n) signal. Furthermore we notice that the internal state is important for the understanding of what happens if a PIN is entered. This is the situation when a new person approaches the Access Point: he cannot know whether another person has just pulled his card through and entered no PIN just to mess up the next user. The internal state is not indicated anywhere and the new user would have to know the whole signal history of the system (process) to know what reactions the system will have on his entering his card or a PIN.

In the diagram in Figure 11-9 (p.11-14) we have not been entirely formal as we have not gone into detail about the 'check' operation of the decision. Data declarations of *c* and *nr* are also omitted.

After the design phase, where the final SDL description of the Access Control system has been established, it is interesting to see whether the properties derived from the designed system correspond to the properties described in Figure 11-9 (p.11-14).

The reader may also notice that Figure 11-9 (p.11-14) may be understood as a “role diagram” where the role is the Access Control providing access rights. The diagram shows all possible behaviors of the user applying this service. The reader is referred to Summary of property modelling methodology (p.11-44) for a more thorough introduction to role modelling.

Harel StateCharts

The attractiveness of a state-oriented view makes Harel’s StateCharts[76] a reasonable candidate for system description.

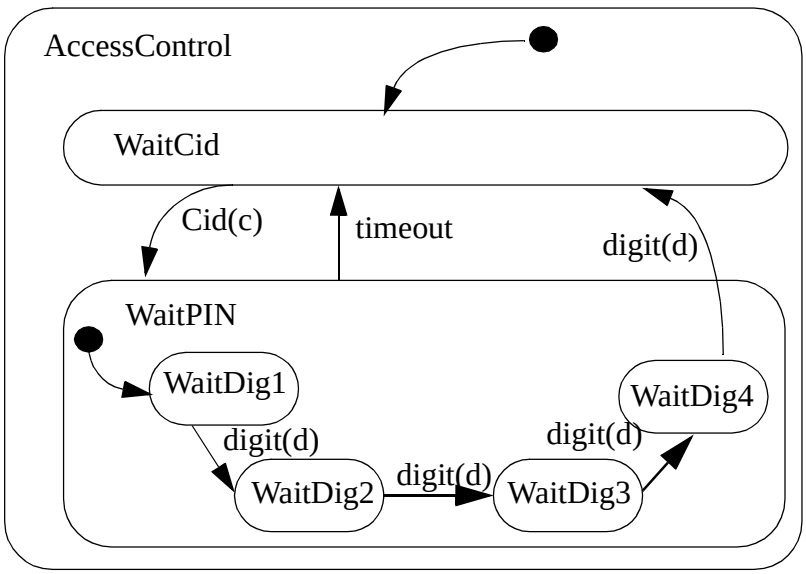
With a sound formal basis and a graphical notation, StateCharts are used both in universities and in industry.

The principle of decomposition distinguishes StateCharts from SDL (see SDL tutorial). In StateCharts the states are decomposed, such that a system at one time is in a stack of states (or a tree of states when a parallel decomposition is involved). In SDL we have something slightly similar when using procedures, as the procedure stack describes a stack of states.

Again we show the properties of the Access Control System

Figure 11-10: .Statecharts

Open figure



In Figure 11-10 (p.11-15) we see that the means for decomposition is the state in states and not as SDL where we decompose structurally block in block. The decomposition of a PIN into four digits is simple. Furthermore we manage very easily to model that the typing of a PIN may be interrupted by a timeout when the user has spent too much time.

Languages based on logic

Predicate logic

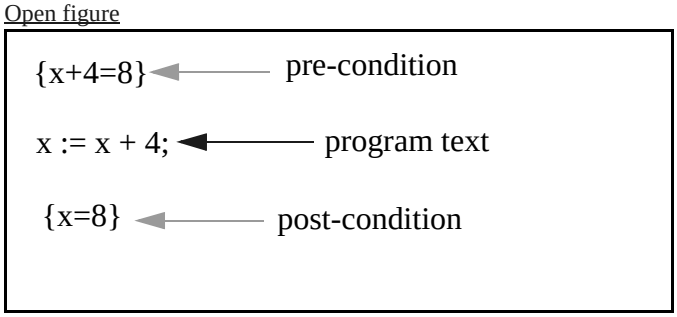
When using predicate logic as the base for descriptions the property model is a set of predicates. The objects of the universe of discourse are variables (identifiers, instances) of the model. The predicates are often aligned with imperative (object-)descriptions by a braid with textual descriptions. This is the case with Hoare style invariants (see Figure 11-11 (p.11-16)) , and with VDM (Vienna Development Method) [113].

In the tradition of Hoare (See Figure 11-11 (p.11-16))[92] , pre- and post-conditions represent property descriptions interleaved with a programming language that represents the object description.

In Hoare-logic we have predicate logic combined with the programming language. The idea is the following:

Figure 11-11: Hoare logic

Open figure

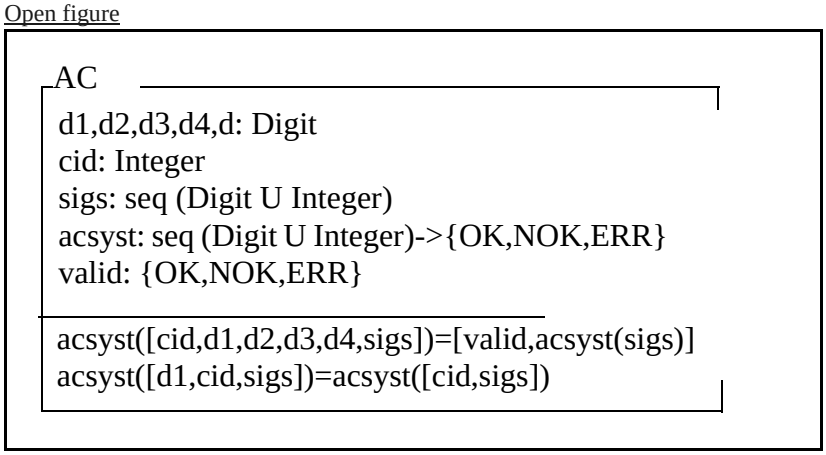


For some odd reason we know that after the program statement “x:=x+4”, the invariant is that {x=8}. Then we can compute backwards by substituting in the postcondition “x” (which is the left hand side of the assignment) by “x+4” (which is the right hand side) to obtain the pre-condition {x+4=8} which is the same as {x=4}. The reader may check that there is no surprise that if {x=4} before the assignment, it will become {x=8} afterwards.

Other languages like Z [90] will use the predicate logic paradigm in a more independent way meaning that the predicates represent the whole description. In Z the “schemas” often represent states or a transition between states.

Figure 11-12: Example of Z

Open figure



The idea of Figure 11-12 (p.11-17) is to indicate how a Z description looks. The top part is a declaration of variables, while the bottom part defines predicates on these variables. We have tried to indicate that the Access Control system can be seen as a function *acsyst* which takes a sequence of signals (either digits or integers) and returns a sequence of validation returns (OK, NOK or ERR). We indicate (not all formal) that the input sequence consisting of one integer (the card identification) and four digits will result in one validation output, and the remainder of the input sequence will be processed similarily. The second line of the predicate indicates that digits preceeding the card will be discarded (ignored) by the Access Control system.

In our methodology we recommend describing invariants of states associated with the state symbol in SDL (See rule).

Temporal logic (CTL)

Two types of properties are normally highlighted as important to describe and to prove.

- *Liveness*: something good will eventually happen;
- *Safety*: something bad will never happen.

We notice that these kinds of properties are not merely dependent upon the values of variables at some point in time. Such properties talk about situations which should come in the future or which should never come in the future. The property descriptions themselves must address time or ordering in time. This is where temporal logic comes in.

In the *Access Control* system a safety property is that the door is never unlocked unless the card is valid. A liveness property is that whenever you have entered a card, you will eventually get a response and the card back.

From these examples we see that while the safety property above is right in the heart of the functionality of the Access Control system, the liveness property is interesting, but hardly sufficient for the satisfaction of customers. This is because safety is the business of the Access control system.

As an example of a temporal logic specification we shall use the language CTL (Computation Tree Logic)[38]. We could equally well have chosen TLA (Temporal Logic of Actions)[130], but CTL was chosen due to the existance of effective algorithms which determine the truth of CTL formulaes.

The language uses a small set of concepts:

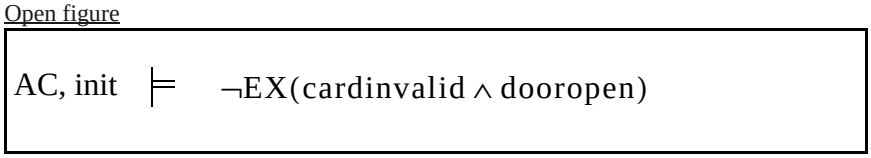
- Any atomic proposition p is a CTL formula,
- If f1 and f2 are CTL formulaes, then so are $\neg f1$, $f1 \wedge f2$, $Xf1$, $EXf1$, $[f1Uf2]$, and $E[f1Uf2]$.
- $\neg f1$ means “not f1”,
- $f1 \wedge f2$ means “f1 and f2”,
- $Xf1$ means “for all states following the current state, f1 holds”,
- $EXf1$ means “for some state following the current one, f1 will hold”,
- $[f1Uf2]$ means “for all paths of states[1] from the current state, f1 will hold until f2 holds”,
- $E[f1Uf2]$ means “for some path of states from the current state, f1 will hold until f2 holds”.

These special constructs can be explained in relation with a transition system, or a computation tree of states. For any state it is possible to assess the truth value of atomic propositions (see Predicate logic (p.11-16)). The truth values of the more composite constructs are derived from this basic notion and they are all interpreted relative to the current state:

If we rephrase our safety requirement above slightly into addressing states instead of events we get: The door should never be open when the last card was invalid. In CTL this could be:

Figure 11-13: Example of CTL

Open figure



The formula is read: “For the *Access Control* system (seen as a transition system named *AC*), relative to the state “*init*” (which we have labelled the initial state of the system), one will not find that some path will end in a state where both *cardinvalid* and *dooropen* are true (two variables which we have assumed present in the state characterizing the Access Control system).

If we had had a realization of the transition system AC, there are algorithms which are reasonable efficient that checks the correctness of the CTL formula relative to the transition system [72].

1. “Path of states” means “sequence of system states in time succession”

Focus

Focus is an approach of the Technical University of Munich where Professor Manfred Broy and his chair has performed research on formal methods for reactive and real time systems for a number of years [29].

Their basic model sees a component as a set of functions that associate output histories with input histories. The histories are infinite *streams* of signals and time ticks. That every component is modelled not as a single function, but a set of functions makes it possible to describe unbounded nondeterminism[1].

The Focus notation is based strictly on mathematical logic and powerful inference rules have been devised. Focus components are compositional such that analysis of a composed system can be derived in steps from analysis of its constituents.

Focus is aimed at describing systems which have a mixture of the following characteristics:

- data transformation;
- data storage and retrieval;
- data transmission;
- synchronization between active objects.

They recognize the fact (as we do) that modern systems very often have all these characteristics combined with strict real time requirements.

- embedded systems (cars, airplanes, manufacturing);
- man-machine interaction handlers;
- components in distributed software.

In order to cope with such diverse requirements, the Focus notation has three modi:

1. Time independent (ti) specifications;
2. Time dependent (td) specifications;
3. Synchronous (sy) specifications.

Their basic theory is based on one form of the *pulse driven* functions which are shown to comprise the three modi above. The three modi are different forms suited for different purposes.

Orthogonal to the choice of modi, there is the choice of *specification style*:

1. Free, direct transitional style;
2. Assumption/Commitment style;
3. State oriented style.

1. Unbounded nondeterminism means that the number of nondeterministic choices cannot be determined in advance. This is the case if you want to specify that a loop will terminate, but you do not know any limit to how many iterations it will run. This can neither be described in SDL nor in MSC-96.

The two latter are styles have fragments of methodology associated. The state oriented style bears close relationships with SDL specifications (see SDL as a property language (p.11-13)) while the assumption/commitment style is a generalization of the pre-condition/post-condition style of the Hoare logic (p.11-16).

We shall describe our *Access Point Controller* process in Focus. To distinguish it from SDL (See AccessPoint in SDL), we will give a specification in the assumption/commitment style.

Figure 11-14: Structure of AccessPoint Controller

Open figure

From Figure 11-14 (p.11-20) we have that the *Access Point Controller* takes a code signal and forwards it to the *Authorizer*, from where it receives a validation result (ok, nok or err) which it forwards to the *Panel*. To make the example simple, we have omitted the controlled door.

Focus[1] **begin**

$$APC \equiv c_2\{code\}, a_2\{(val)\}, \nabla b_2\{code\}, c_1\{(val)\} \overset{td}{o = o}$$

-- The preamble of a time dependent specification defining the signals on the channels just like in the SDL-like illustration Structure of AccessPoint Controller (p.11-20)
-- *td* above the equality sign shows that our specification is time dependent.

assumption

$$\forall(r \in N)(\overline{\#c_2}\big|_r \geq \overline{\#a_2}\big|_r)$$

-- There is more code input than validation input at any given point r in time.
-- The bar over the channel identifiers indicate that only signals and not time ticks are considered.

$$\overline{\#b_2} = \overline{\#a_2}$$

-- The number of code signals out is the same as validation signals in indicating that each code out triggers eventually a validation result in

commitment

1. This notation used here is an adaptation of an early form of the Focus notation. Its final notation varies from this.

$\overline{b_2} = \overline{c_2}$

-- Every code coming in will eventually go out unchanged. We could make it even stronger by demanding that it should never go more than *x* ticks in real time from the code signal was consumed until it was output. This would be written.

$\forall r(\overline{b_2}|_{r+x} \subseteq \overline{c_2}|_r)$ meaning that the output at time (r+x) is always a prefix of the input at time (r)

$\overline{c_1} = \overline{a_2}$

-- Similarily the validation input (from the Authorizer) will be output. This can also be strengthened by real time constraints.

end LSC

We may notice that the requirements on time which we have categorized as non-functional (see Relating to whether the properties define functionality (p.11-7)) are integrated with the functional ones in a way which makes the categorization of response times as non-functional dubious. The choice of basic theory and of notation has an impact on what properties are functional and which are not.

The compositionality of the Focus specification can be seen if we specify each component of the system, and also specify overall system requirements. By the theories behind Focus it will be possible to prove that the system decomposed into components is a refinement of the overall system requirements. Once this has been shown on the topmost level, each of the components may be refined in isolation without upsetting the top level refinement result.

The strong point of Focus is its ability to cope with very diverse components within the same system. Its solid theoretical base makes it suitable as a common semantical framework for modern distributed and diversified systems.

The weak point of Focus is its mathematical notation which makes the specifications less transparent than desirable.

Summary property languages

Table 11-1: Property language comparison

property	Prose	MSC	Focus	SDL
liveness, safety	fair	fair[a]	good	fair[b]
overview	good	good	fair	good
interaction	poor	good[c]	fair	fair[d]
time requirement	fair	poor[e]	good	poor[f]
performance	fair	poor	fair	poor

Table 11-1: Property language comparison

property	Prose	MSC	Focus	SDL
transparency	good	good	poor	good
expressiveness	fair	fair[g]	good	good
formalization	poor[h]	good	good	good

a. MSC must be supplemented by more information to cover temporal logic.
b. SDL may provide a complete specification, but temporal properties are not transparently highlighted.
c. MSC is especially good for describing interaction where message sending is important.
d. SDL focuses on describing signalling locally, therefore the sequence of signals may be more difficult to grasp.
e. MSCs may be annotated to accommodate for some timing requirements. Real time is not a part of MSC.
f. SDL is, like MSC, not good at describing absolute time.
g. MSC is not so expressive when it comes to other aspects than message sequences
h. The lack of formality is the main problem with prose. A related problem is that prose is often ambiguous and therefore easily misunderstood.

Alignment

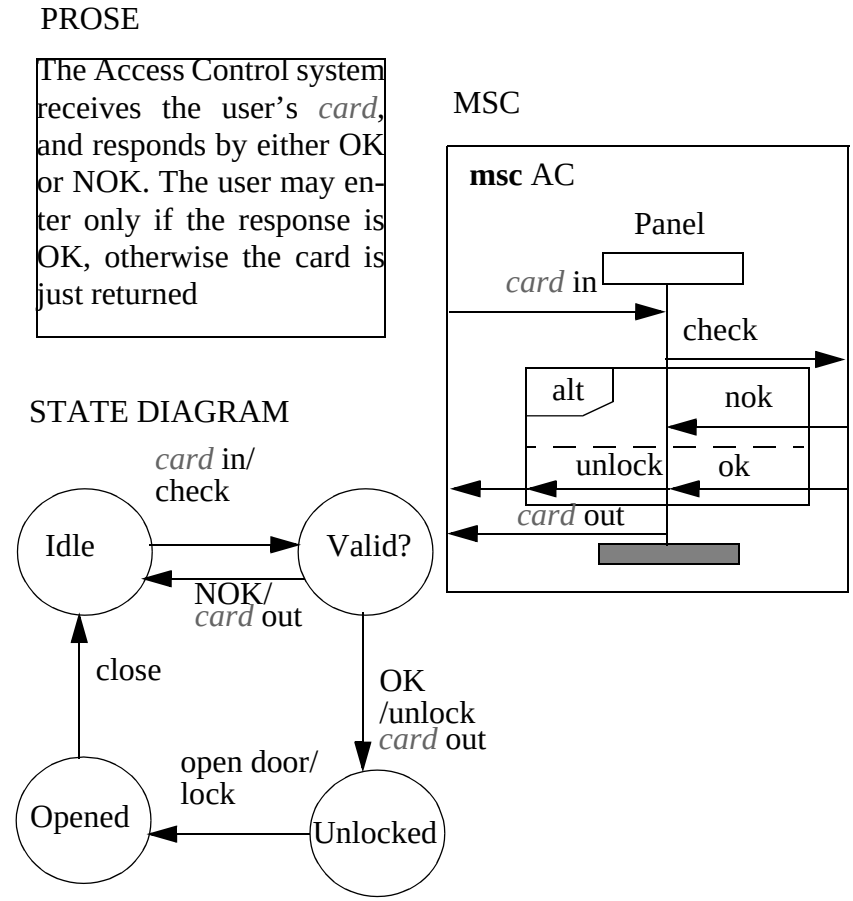
The property descriptions describe specific aspects of the model, on different aggregation levels and on different abstraction levels. An important issue is to be able to assess that the different descriptions of the same model actually talk about the same model.

In some cases this is a simple task, while in other cases it is not so simple. The most difficult situation is when the alignment appears to be simple, but in fact is more intricate.

Here is a simple example, which still turns out to pose some problems:

Figure 11-15: Aligning prose, MSC and state diagram

Open figure



In Figure 11-15 (p.11-23) we have little problems with recognizing that the three different descriptions all describe the same situation in the Access Control system. Still the three perspectives do not cover each other completely.

The prose says very little about the exact sequencing of the messages, and little about the requirements which the user must fulfill (such as opening the door himself and closing it afterwards) before the system is again ready for another card.

The MSC says little about the fact that there is a User, and it has no indication of the states which the system passes through. Furthermore, the (visual) response to the user is omitted.

The state-oriented diagram is the most complete one, but still it is still not formal enough to be used for automatic code generation.

The informal alignment is basically given by identical identifiers. We have highlighted *card* as one such identifier.

When we want to align the three descriptions more formally we run into questions such as:

- Where are the states (of the state diagram) in the other descriptions?
- Where is the Panel (mentioned in the MSC) in the others?

- Where is the AccessControl system (mentioned in the prose) in the others?

Such differences may not be important, as it is obvious that each perspective will have its own “aids for the thought”. The state-diagram uses states as a major means for expression, while MSCs need instances which produce and consume messages. A process algebra description (and other notations based on logic) often uses auxiliary functions.

In our methodology, where the object model is a complete SDL specification and important properties are expressed by MSC, there is a fairly straight forward alignment mapping (see MSC-92 Methodology).

Types of alignment

The idea of alignment is that the object description and the property description together form a combined description. The combination of descriptions can be done in more than one way:

- Textual braid (see Figure 11-11 "Hoare logic" (p.11-16))
- Separate descriptions of the same defined situation (see Figure 11-15 "Aligning prose, MSC and state diagram" (p.11-23))
- Different descriptions within the same overall system (see combined descriptions)

Formal basis

As pointed out in Some Property Languages (p.11-10) an important dimension of a description language is whether it has a precise semantics. From the summary Table 11-1 "Property language comparison" (p.11-21) we see that verifiability corrolates negatively with transparency for those languages which are specifically strong on either dimension. Languages which are strong on formalisms such as Focus (p.11-19), Temporal logic (CTL) (p.11-17) or Predicate logic (p.11-16) score badly on transparency. On the other hand languages which score facvorably on transparency such as Prose (p.11-11) (and other informal notations not mentioned in our chapter here) have little to offer on verifiability.

We realize that our complex systems are best described by a number of languages, due to their different strongholds or due to historical reasons. Then it becomes increasingly important to be able to handle them within the same precise conceptual framework which preferably also should support chances to perform formal reasoning.

Alignment between the descriptions can be seen as adjusting the binoculars such that both eyes focus on the same thing. Still we cannot be sure that both descriptions describe the same situation. The specified structures or behaviors may not be consistent.

What we need is a common basis on which to build our semantics of the different descriptions. The semantics of CTL (Temporal logic (CTL) (p.11-17)) is given relative to a labelled transition system. This means that given a transition system, it is possible to determine the consistency of a CTL description relative to this labelled transition system.

In the same manner, MSC can be given an interpretation relative to an SDL system and thereby the consistency between an MSC and an SDL can more easily be determined.

In Geode[1] the extended MSC and the SDL both have interpretations as labelled transition systems and thus their consistency can be established.

By defining MSC and SDL relative to a formal basis of labelled transition systems, the tool vendors have formally a proof obligation that their interpretation is equivalent to the formal definitions. For SDL the official formal definition is given in MetaIV (a variant of VDM), and for MSC it is given in a process algebra framework.

Here we would like to give an informal procedure for how a common formal basis can be utilized for consistency checking:

1. Translate each description into the common formal notation.
2. Apply the alignment mapping to make the set of common identifiers as comprehensive as possible.
3. Filter out all aspects which are not covered by the alignment mapping.
4. Compare the resulting descriptions (formulaes).

Often it is not effective to translate the whole descriptions before comparing. Comparison can be done in parallel with the translation, the application of the alignment mapping and the filtering.

This is the case with commercial SDL/MSC validators. After the alignment of states to position in MSC has been done, the parallel execution of the SDL and the MSC takes place. Since there are several possible execution paths the execution involves backtracks. Still since what the validator is normally looking for is the existance of a specified trace (described by MSC), a result may occur earlier than if the full descriptions had been translated to some labelled transition system.

For more on formal reasoning and consistency examination, the reader is referred to the theme on Verification and Validation.

1. Verilog (1994). GEODE Simulator. Basic Concepts. Reference Manual - GEODE Simulator . Toulouse, France, Verilog. 1-36.

The SISU Property Modelling Technique

As shown in Some Property Languages (p.11-10) there are many languages and each has a different purpose. This methodology also needs to be able to express the different property aspects, but we want to keep to a fairly small set of notations.

Our basic formal notation for property modelling is MSC-96. For MSC to be able to describe liveness and safety properties, there is a need to extend MSC in the direction of quantifiers. We propose an MSC extension which we call Meta-MSC (p.11-42) which is inspired by Temporal logic (CTL) (p.11-17).

Our technique is based on a scheme which can be applied to any level of detail, but different levels of detail have different characteristics and different criteria for completeness.

On the domain level, there are two points which characterize our property modelling technique:

- Service orientation (p.11-26)
- Role orientation (p.11-27)

Service orientation

For the kind of systems which the SISU methodology is suited to design, it seems practical to consider that the behavior of the systems can be seen as a menu of a set of services which the system provides to its users.

A service is a unit of behavior which the user recognizes to be provided by the system. The first approach to a system analysis is often to list the services by using simple prose as described in Prose (p.11-11). The next step will be to formalize these services by using formal languages, e.g. MSC.

To make more precise what we mean by a service we shall list some of the properties held by services. Services are charcterized by the following:

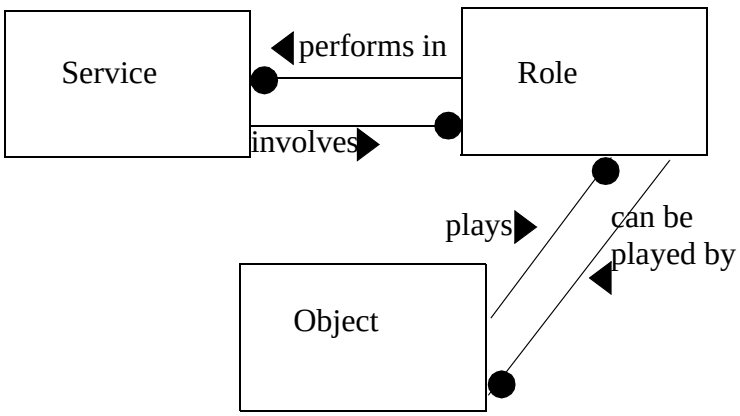
- A service is a pattern of behavior involving one (or more) service provider(s) and at least one user.
- Services may be interleaved in time.
- Important aspects of a specification of a system (or service provider) is achieved when its services are satisfactory described.
- A system can be characterized by a set of services. Each service may again use subservices.

Role orientation

In the end our system will consist of active objects which provide the services. When an object provides a service, it plays a *role*. An active object may play a number of different roles, and more than one object may have the capacity to play the role. It is also the case that a role may perform in a number of services.

Figure 11-16: Role, service and object

Open figure



In the domain analysis we describe the services with roles as the acting entities. This gives better flexibility when the concrete design of a system is to take place.

A role is a behavioral pattern which describes how one acting object performs a set of related services. Normally when describing role diagrams (e.g. in MSC) only messages involving the focused actor will be shown. On the other hand it is normally assumed that all messages involving the focused actor concerning this specific service will be shown. The different alternative courses of behavior should be described.

In the method OORAM [162] the active objects are built from roles. This technique is called *synthesis* and it amounts to defining the active objects as a union of a set of roles.

The relations between different roles can be shown by an object model. The role behavior is described by some property modelling language (cf. Some Property Languages (p.11-10)) such as typically MSC (Figure 11-17 "Roles expressed by MSC-96" (p.11-28)) or SDL (Figure 11-9 ".SDL as property language" (p.11-14)).

In domain analysis the relations between services and roles are of primary concern, while during the design the relations between the roles and their performing objects become the focus. We will refer to these associations between roles and their performers as casting.

During design we recognize a special kind of role which can be constructively used during the design and in connection with validation. When the objects have been identified and the communication lines established, it is possible to isolate roles describing the projections of the object behavior onto each communication line. Such roles are called interface roles or *interface projections*.

These kinds of roles can be used constructively, since whenever an interface role is known on one end of a binary communication line, the interface role of the other end is the inverse and can therefore be constructed. Depending on what is initially known, the interface roles can be used to produce the behavior description of an object or to ensure its consistency.

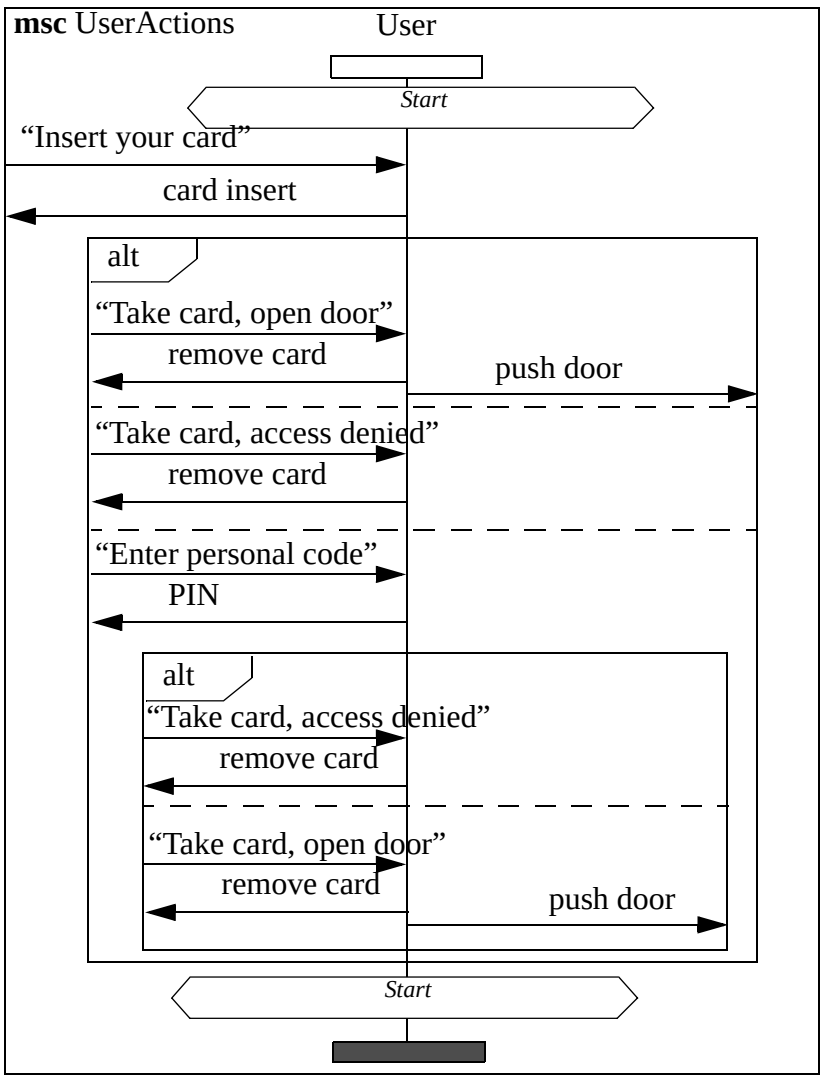
During validation the projections can be used to check for certain consistency rules which assures that general design criteria are met. (See rule for Risk index).

When services are binary (one service provider and one service requester) and only one communication line is used, interface roles and service roles overlap.

In Figure 11-17 (p.11-28) we show how the *User* will behave relative to an *Access Control* system. In SDL the corresponding *Access Control* system may act like in Figure 11-9 ".SDL as property language" (p.11-14) (where the cases of PIN not needed is not covered).

Figure 11-17: Roles expressed by MSC-96

Open figure



The company strategy

It is very important for the success of the descriptions that there is a clear understanding of why the descriptions are made, and for whom they are made.

Awareness of these questions makes the production of property descriptions more focused and thereby better. More about how the company strategy can be specified can be found in our MSC-92 methodology.

The dialectics of refinement

We will present our basic principle of property refinement in this section. The starting point is that we have a description of some entitity. This entity can from a property point of view be seen as a piece of behavior. We do not here take a stand on exactly how the entity is described, but we may assume that both informal prose and more structured MSCs are used.

Our task here is then to describe the road ahead from such a description, which also represents a state of understanding in the development team.

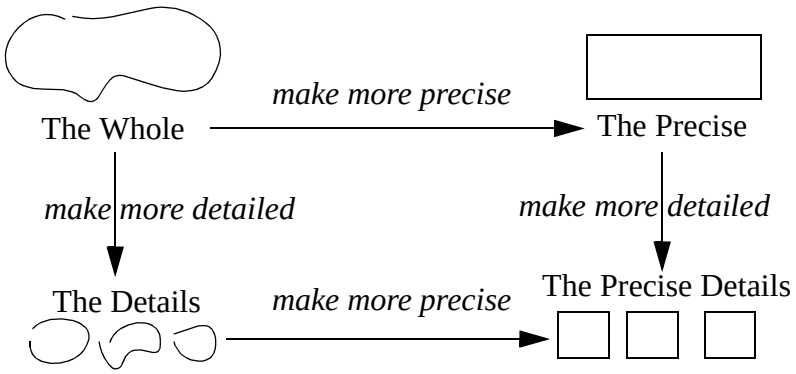
We face two different needs which interact:

- The need for more precise description (p.11-30);
- The need for more detailed descriptions (p.11-30).

It is important here to realize that these needs indeed are different, and that their fulfillment requires different means, and that they are interlinked. Following a brief introduction there is an example Example: Access Control: Change PIN (p.11-31) where the concepts are applied in developing a service in our *Access Control* system. From each concept a specific reference is given to the appropriate section in the example

Figure 11-18: .The Whole, The Precise and The Details

Open figure



The need for more precise description

In property modelling making the property descriptions more precise may involve a number of different approaches:

1. To *formalize*, meaning that we move from descriptions in prose to descriptions in a formal language (MSC (p.11-11), Focus (p.11-19), Temporal logic (CTL) (p.11-17) etc.). See example in Formalize (p.11-31).
2. To *narrow*, meaning that we add more properties such that our description is less underspecified. See example in Narrow (p.11-32).
3. To *supplement*, meaning that we add properties of other aspects than what we did before. This could mean adding time-dependent properties explicitly in addition to the plain MSC description. See example in Supplement (p.11-33)

The need for more detailed descriptions

Even though it may be the ideal to describe a situation without going into much detail, it is repeatedly shown that sometimes the understanding of the details are necessary for the understanding of the whole.

If we assume that our whole entitity is represented by an MSC, we can easily distinguish a set of means to make the description more detailed:

1. To *decompose* the instances, meaning to see how the intances of the whole looks inside. Which instances do they consist of, and how do these instances interact to make up the already known whole? See example in Decompose (p.11-35).
2. To *break down* the protocols, meaning that what on the upper level looks like one message or one behavioral pattern (e.g. an MSC), on the next level is a protocol of interaction. See example in Break down (p.11-36).
3. To *reveal*, meaning that more of the total scene is considered, more messages, more instances. See example in Reveal (p.11-37).

We see that several of these approaches to precision and detail can be used simultaneously, but we should be aware of which mechanisms we actually use.

During the process of making more precise and more detailed descriptions, the understanding improves and the improved understanding should feed back to the original starting point making the description of the Whole more accurate.

Description distillery

The process of reaping the experience on the upper level from work on lower levels and from making the description more precise, we choose to call “tdisilling”. It amounts to making the description cleaner, more pure, and more valuable. In this process, an important task is to define the accurate bounds of the Whole.

Compositionality preparation

A central idea of our methodology is to be able to apply the same overall techniques also on the next level of detail. Then we have to make sure that the distinction between the levels is reasonably clear. Why this is important is that we want to assess – informally or formally – that the levels of detail define a refinement relation.

Example: Access Control: Change PIN

The distillery approach should be more easily grasped when we go through parts of our *Access Control* example in some detail to explain what we mean by each of the individual approaches. We shall try to apply only one individual approach at the time while in practice a development step may use more than one approach.

The example starting point

We shall start in the domain model of *Access Control* systems. We have a Domain Statement from which we extract:

- Users shall be able to change their secret code

This is the service that we shall develop in this section. It is obvious that the simple line above of the domain statement may give rise to a number of interpretations and a number of possible implementations.

Make more precise

It is reasonable to start by trying to make the statement more precise, and we have three approaches: formalize, narrow and supplement. It is not obvious which of these approaches will give the best progress.

Formalize

Trying to formalize will often make the designers discover more about the prose statement than they first thought was in it. On this level of understanding, it may not be the best strategy to take MSC and make message sequences because then we get into a very constructive way of thinking before we have pondered about what we really mean by our prose statement. Rather a more declarative approach in the style of Hoare logic Figure 11-11 "Hoare logic" (p.11-16) could prove fruitful. First we consider how the situation before and after the service could be described formally.

In this case we have a notion of a Personal Identification Number (*PIN*) associated with a Card identification (*Cid*). The database (*DB*) consists of a set of such pairs such that *Cid*s are unique. The service changes one such pair such that the *PIN* has been changed.

Figure 11-19: PIN change in Hoare style

Open figure

$$\text{Let } DB \subseteq \{(c,p) \in Cid \times PIN \mid ((c1,p1) \in DB \wedge (c2,p2) \in DB \Rightarrow \neg(c1 = c2))\}$$

$$\{(c,p) \in DB\}$$
$$\text{ChangePIN}$$
$$\{(c,q) \in DB \wedge \neg(q = p)\}$$

The formula in Figure 11-19 (p.11-32) should be understood like this: The first line defines the database DB as a subset of the set of pairs (c,p) of a Cid and a PIN such that no two pairs in DB share the same Cid. The second formula gives a precondition (assumption) which only says that the pair (c,p) is in DB, and the postcondition (commitment) says that after the ChangePIN operation has been applied, there is another pair (c,q) in DB such that the new PIN q is not equal to the old PIN p.

There are of course other basic models, such as describing the Cid,PIN association as a function from Cid to PIN.

In this case our aim was to show how little the domain statement actually said. Many interpretations are possible. For instance the statement and the formalization says nothing about whether the User is to choose his new PIN himself or if the system selects it for him! This leads us to our next approach.

Narrow

When the prose (or formalization) gives rise to too many interpretations, we must add more properties so that the number of valid interpretations decreases.

In our example a valid interpretation is that the User gives a command to change PIN and a new PIN is delivered from the system. There is no requirement that the User shall have the possibility to choose his new PIN.

Furthermore we have not specified any requirement that the ownership of the card is to be validated before a new PIN is accepted.

By adding these new requirements we narrow the set of valid interpretations without formalizing.

Figure 11-20: PIN change narrowed

Open figure

Service PIN Change

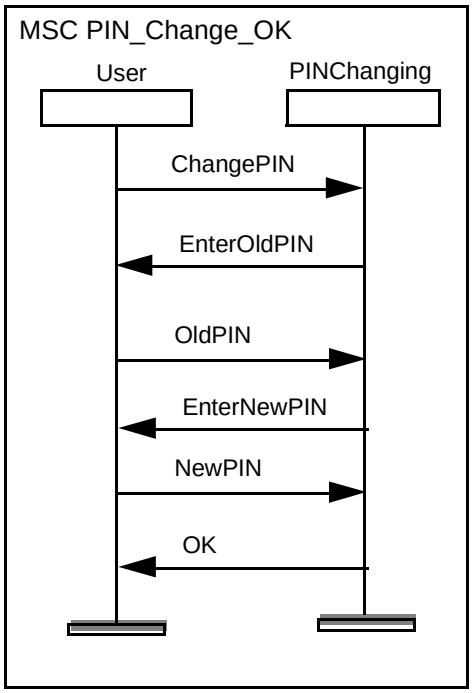
- Users shall be able to change their personal identification
- The User shall be able to choose his new PIN
- The Card shall be validated by the old PIN before a new PIN can be given. The new PIN shall subsequently also be validated.

We notice that our initial formal model is not rich enough to formalize the narrowed informal specification in Figure 11-20 (p.11-32). In order to formalize this, it is necessary to introduce a User concept in the formal model. One way could be to include more values as given in the precondition designating the User giving old and new PIN.

The last requirement, however, is not merely a requirement on the situation before or after the service, but a requirement on how the service should be performed. The pre- and postcondition style is not well suited to express such requirements since the service itself is considered a black box and its results are all that matter. MSC, on the other hand, is especially tailored to express such execution sequences.

Figure 11-21: MSC User changing PINwith success

Open figure



We may now validate whether the formalization of Figure 11-21 (p.11-33) satisfies Figure 11-20 (p.11-32). We easily see that this is not the case!

Firstly the formalization does not validate the new PIN again. Secondly the formalization covers only successful runs.

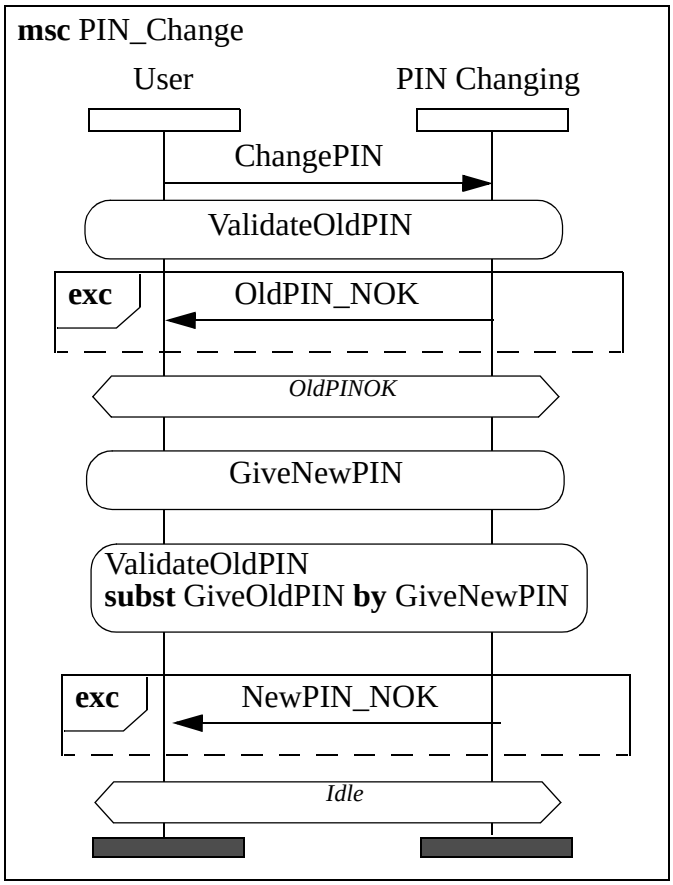
Supplement While narrowing restricted the set of valid interpretations, supplementing increases the scope of the specification. New and important aspects are considered which were not an issue earlier.

Typically, when the formalization is done in MSC, the MSC does not quite cover all the situations covered by the informal specification. There is a need to supplement with exceptional and erroneous cases and more normal cases.

In our example we need to specify what happens if the Old PIN is not properly typed in, and moreover that the New PIN should also be validated and what happens if it is not handled properly.

Figure 11-22: Change PIN (MSC-96)

Open figure



In Figure 11-23 (p.11-36) we have used MSC-96 with MSC references and exceptional operators. We have used substitution to show that to validate the new PIN, one can use the same sequence structure as when validating the old PIN.

Other aspects which could possibly supplement our MSC are time and capacity aspects. Are there any requirments on how fast the system should respond to the User commands and data? Such requirements could be attached here by comments in the MSC or through more textual prose.

Requirements to the response times of the User may be specified through the use of timers, but that seems more adequate as a matter of specifying the actual protocols in more detail.

For this service it would be valuable to specify some idea of how frequent the service is intended to be used. We believe that changing the PIN is a service which will occur very infrequently compared with the main service which is the access of the zones. In Break down (p.11-36) the reader can see how this supplement affects the functional specification of the service.

Supplementing and narrowing can sometimes be difficult to distinguish, but in practice it is not important that the designer knows whether he has applied one or the other. The goal of both approaches is to make the descriptioin more precise such that misinterpretations cannot occur.

Make more detailed

We have in Make more precise (p.11-31) reached a formal description Figure 11-23 "Change PIN message" (p.11-36) which covers the informal and narrowed description Figure 11-20 "PIN change narrowed" (p.11-32). A reasonably smart tool would be able to simulate (execute) that description e.g. by implementing all MSC references as the empty MSC, but tracing the MSC reference occurrence. The informal descriptions do not give much clues to how the pending MSCs shall be described or implemented. This means that, regarding the top level requirments almost any implementation of these MSCs will actually be valid interpretations of the service.

Furthermore it is quite obvious that at some point we must express something more about what the message “ChangePIN” actually boils down to.

One of our instances in the MSC is the “PIN Changing”. This is not an object of the object model, but a functional role (See Role orientation (p.11-27)). Behind this name there is hidden any structure of interacting objects. At some point in the development it is necessary to associate the functional role with an object of the object model. This is called casting. In principle the functional roles can be seen as projected objects such that they may be decomposed into components. This will, however, increase the complexity of the casting.

Decompose

Our first approach in making the description more detailed is to decompose the instances. We have up to now two instances in our description: the User and the PIN Changing. The User may of course be decomposed into Hand, Eye, Foot (to kick the door), but this is hardly very fruitful.

The PIN Changing role may possibly be decomposed. As we find in many situations it may be practical to separate out one component which takes care of the interface and one part which does the functionality. Regarding PIN Changing, we may have one component for the User Interface (called PINChangeIF) and one component which performs the validation of the PINs (called the Validator). It is reasonable to believe that the Validator role is similar to something needed in the User Access service which should be supported by reuse.

We notice here that we are not dependent upon having a formal description of the specification to perform decomposition. Decomposition can be done on informal as well as formal descriptions.

Using MSC-96 we have two choices in the methodology for handling decomposition:

1. Remake the MSC Figure 11-23 "Change PIN message" (p.11-36) such that the components take the place of the former PIN Changing. The original MSC will become historical.

2. Apply MSC **decomposed** on PIN Changing within ChangePIN. This must be done with some care since decomposition is orthogonal to MSC references, but consistency should be maintained.

The MSC-96 tutorial gives a more thorough procedure for how decomposition should be handled.

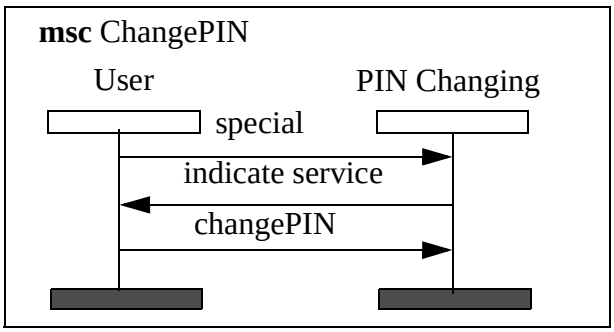
Break down

There are concepts in Figure 11-23 "Change PIN message" (p.11-36) which are basic, but which still may need some elaboration. The signal *ChangePIN* introduces the service, but what is this signal? Is it the pushing of a special key on the panel? That would mean an increased production cost of the panels due to a service which is probably rather infrequent.

Another alternative is to use a special key for entering a specific mode for all services which is not the main service (namely the accessing of the zones).This is a possibility and it would require that *ChangePIN* boils down to a human interface protocol.

Figure 11-23: Change PIN message

Open figure



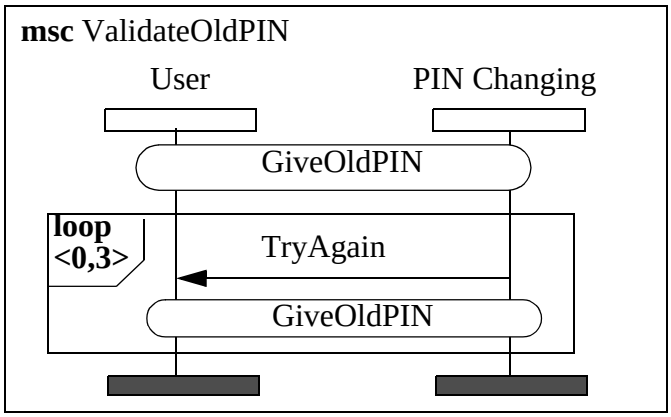
It may in addition be the case that there are only a set of special terminals which can be used for such administrative services. That would be input for the object synthesis activity where services are associated with objects.

Formally MSC-96 cannot refer an MSC from a message name. When we have broken down the ChangePIN message by the MSC ChangePIN as in Figure 11-23 "Change PIN message" (p.11-36), we must return to all MSCs using the message ChangePIN and perform modifications. See the MSC-96 tutorial for examples.

When it comes to breaking down the behavioral patterns, MSC-96 offers good support. In Figure 11-23 "Change PIN message" (p.11-36) there are several MSC references which need definitions. As noted earlier almost any definition would do. Here we show one possible implementation of ValidateOldPIN.

Figure 11-24: Validate Old PIN

Open figure



Still we have not broken down the MSC references to *GiveOldPIN*. The reader should also note that in MSC-96 it is not possible to specify *why* a specific alternative occurs. In our example where Figure 11-24 "Validate Old PIN" (p.11-37) implements ValidateOldPIN in Figure 11-23 "Change PIN message" (p.11-36) it is avalid bahavior that the message OldPIN_NOK is sent even when the loop in ValidateOldPIN has been executed less than three times.

Reveal

When we are considering the situations in more detail there may be aspects which we have omitted or forgotten which now become more interesting. In our case we seem to have forgotten that the access is dependent upon two independent pieces, the card and the personal PIN. In our MSCs we have highlighted the PIN, but forgotten about the card.

Possibly we have thought that all services start by entering the card and that the card is returned after each service has been completed. This would mean that the card is handled on a higher level than each individual service. In general we do not advise this strategy since our service-oriented approach considers the services to be a flat set on the top level. But, assume for now that this approach has been chosen. Then one interesting aspect with the card and the cardreader is whether the cardreader is capable of keeping the card if the validation fails. This is interesting because then there cannot be an automatic return of the card after the service!

Alternatively the entering of the card must be part of the breaking down of the message ChangePIN, and the return of the card must be part of the breaking down of the final OK or NOK messages.

The distinction between revealing and breaking down is similar to the distinction between supplementing and narrowing. Revealing means to discover new elements, enter new aspects, which was what supplementing also did. The difference is that revelation introduces new instances and objects, not merely new information. On the other hand breaking down actually limits the scope as an infinite number of other configurations are excluded. This is the same as narrowing, but with respect to the details of some entitiy. Just as the distinction between supplementing and narrowing could be difficult, the distinction between revelation and breaking down can also be difficult or a question of definition.

Distillery

In distilleries the pure substance is separated from the waste. In software engineering the problem is that the purified results are mixed with old and wasteful designs. We say that we apply top-down design or stepwise refinement or something similar, but when it comes down to providing the steps and the layers, our descriptions do not always live up to the expectations.

During our progress with our example service “Change PIN” we have also gained understanding. We discovered that our original specification was too vague and could lead to implementations which were clearly undesirable.

We were able to reach a fairly complete, and fairly precise description of the whole which did not go into great detail. (Figure 11-19 "PIN change in Hoare style" (p.11-32), Figure 11-23 "Change PIN message" (p.11-36)). This precise, but not so detailed description serves well as a communication medium with non-professionals and as an introduction to newcomers to the project group. Furthermore it is precise enough (formal enough) to serve as base for simulation and verification. This description is the “*distilled whole*”, a description which has been purified through a process of making the initial description more precise. That process may also have benefitted from the process of detailing as it may include details which on first thought appeared to be irrelevant on this level anyway.

The software engineers should not consider the precise detailed description as the only valuable result, even though it marks the next step toward realization. It is important that also the precise detailed description is verified to be an implementation of the distilled whole.

See MSC-96 Tutorial for examples.

Strategies for property modelling

Strategy for Domain Property Modelling

1. Identify separate services which should be offered in the domain.
2. For each service, provide a prose description.
3. For each service, define which roles provide the service.
4. For each service, make the description more precise by:
   - *Formalizing (1)*: Transform those aspects which may into a formal language. The behavior should preferably be described in MSC or SDL. See language specific methodology for details (MSC-92, MSC-96, SDL).
   - *Formalizing (2)*: Those aspects which do not lend themselves easily to descriptions in MSC or SDL should be described in semi-formal prose (See Specifying performance (p.11-40) and Formalizing Liveness and Safety (p.11-41)) and structured comments.
   - *Narrowing*: Find out what questions were not addressed in the prose version and make decisions on these matters. (See Narrow (p.11-32))

- *Supplement*: Make sure that the precise description covers all those cases which the prose covers. (See Supplement (p.11-33))

5. Associate every role with objects of the object model (alignment through casting).

Strategy for Design Property Modelling

1. Take every service of the corresponding domain model and make sure that all roles are played by objects in the design structure. Remake all domain property descriptions so that they refer to the design software structure which is preferably in SDL.
2. Make the descriptions more detailed by:
   - *Decomposition*: Transform the descriptions such that they apply to the substructures of the objects and not only to the objects themselves. (See Decompose (p.11-35))
   - *Breaking down*: Break down the messages and higher level protocols such that their internal structure becomes known. (See Break down (p.11-36))
   - *Revealing*: Reveal new instances and messages which prove to be interesting when a more detailed view is to be described. (See Reveal (p.11-37))
3. Having reached a precise and detailed description, make sure that it is covered by the precise, but more abstract corresponding domain description.
4. Make sure to retain the structured comments and associated semi-formal prose of the domain descriptions in the corresponding design descriptions.
5. Use the design MSC property model as base for producing SDL process skeletons (Construction of SDL Skeletons from MSC (p.11-47)). The automatic production of skeletons can be used for discovering inconsistencies in the MSC property model. The produced skeletons should then be compared with the design object model and a complete design SDL model should be produced.

The art of Formalizing

The process of formalizing an informal description is of utter importance for the success of the engineering project. In this section we shall give some guidance to how formalizing can be achieved based on the classification of properties presented earlier in this chapter.

Strategies for formalizing

1. Try to separate the procedural aspects from the results of the services.
2. Specify for each service what assumptions are made about the situation or the behavior of the environment before the service is invoked.
3. Produce a Hoare-style specification of the service, which describes pre- and post-conditions. This requires that there is a fairly formal model of the results available. (See examples in Figure 11-19 "PIN change in Hoare style" (p.11-32) and Figure 11-12 "Example of Z" (p.11-17))

4. Specify performance assumptions and requirements related to the service. Performance consists of aspects of capacity, timing and duration.
5. The procedural aspects should be modelled in MSC (See example Figure 11-23 "Change PIN message" (p.11-36)) (or SDL).

Specifying performance

The assumptions and requirements of performance are often used in determining which detailed functional properties the service should have. Therefore it is important to agree upon these properties early in the engineering process. Here we provide the engineer with a template for making the specification of performance somewhat more precise without using a formal language as such.

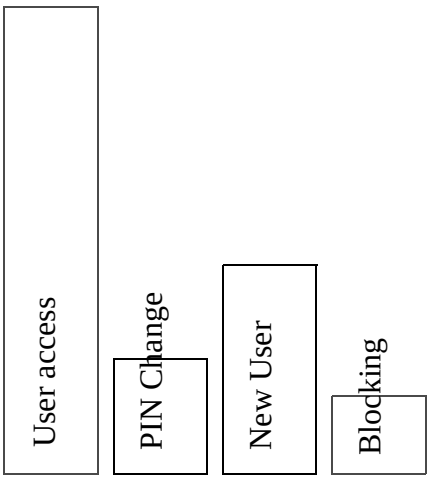
Invokation frequency

- Service X normally counts for Y % of the User service invokations.
- Service X is invoked indirectly from services A,B,C,... which together make up approximately Z % of the User service invokations.
- Service X is critical (not critical) for the overall functioning of the system.

The above results may be arranged in a simple bar chart where each service has one bar and where the color of the bar depicts whether it is critical or not to the overall functioning.

Figure 11-25: Bar Chart for Direct Invokation Fequency

Open figure



In Figure 11-25 (p.11-40) we show a possible example for our Access Control system showing the small invokation frequencey of PIN Change relative to User Access.

Response

- Service X should never take more than Y seconds.
- On the average service X should take Z seconds.

If the service includes interaction with other participants (either User or technical partners), their response times should be set to zero. Specification of their response times are assumptions which can be described like the following:

- During service X at point P, the environment will respond within Q seconds. The point P is defined within an MSC by a structured comment.

Other non-functional

While invokation frequency and response times are mostly service based, reliability, security and physical properties are normally properties of a whole system

- The MTBF[1] of the whole system S should be more than T. A Failure of the system S is defined as D.

Typically a failure can be defined as “The system S must be booted” or “the system S is unable to respond to service X within R seconds”.

- The security of the system should be such that services A, B, C should never be subject to unauthorized completion and these services are secured by E. Services U,V,W have moderate protection and are secured by F.

The security is here described service-wise. Protection means are typically:

- User password
- User physical key (such as a magnetic card)
- Physical protection (such as a locked room for specific terminals)

Physical constraints include such aspects as temperature, humidity etc.

- The system will function in the following temperature range: U,V.
- The system should not be subject to humidity higher than H.
- The system may (may not) function when subject to direct sunlight.
- The system may (may not) be subject to excessive shaking or bumping. (This property may be further elaborated if the system is intended for use in harsh environments such as on sea, under ground or with great mobility)

Formalizing Liveness and Safety

Temporal properties, traditionally called liveness and safety properties, are functional properties which cannot easily be described by MSC as such. We will advise the engineer to select from two slightly different approaches:

1. Structured prose inspired by CTL (Temporal logic (CTL) (p.11-17)).
2. MSC with added interpretation (Figure 11-7 "Liveness in MSC" (p.11-12)).

Prose CTL

Below we present a set of template statements inspired by CTL which may be combined with logical connectives like *and* and *or*.

- Whenever the system S comes in state T, P will always be true afterwards.

1. Mean Time Between Failures

For our Access Control system we could formulate: “Whenever the system *Access Control* has come in the initial state, afterwards it will always hold that for every *Local Station*, if a display shows “*No access*”, then the *door* is closed.”

- Whenever the system S comes in state T, Q will never be true afterwards.

The latter statement is identical to the first statement provided that Q=not P.

- Whenever the system S comes in state T, R might hold sometime afterwards.
- Whenever the system S comes in state T, P will hold until V holds.

We shall interpret the above statement such that V eventually will hold, and before it holds P will hold in all states following the state T.

In our Access Control system we may apply the above template: “Whenever the *Local Station* comes in a state where the *door* is open, the *cardreaders* will be empty until the *door* is again closed”. We have here that S=Local Station, T=door is open, P=cardreaders will be empty, V=door is closed. This statement assumes that the cardreader is enabled only when a card will be handled.

- Whenever the system S comes in state T, it might happen that P will hold until V holds.

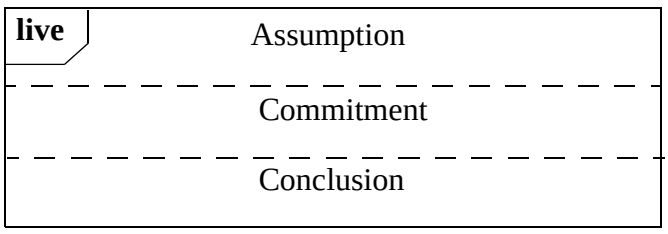
By this we interpret that there is a path of states following T such that in the start of the path P holds, but eventually V will hold.

As we see, the CTL-inspired way of describing temporal statements is declarative. It concentrates on what can be said about the situations. Our second approach concentrates on which behavioral sequences we would like to exclude or include. The two approaches can be made equal if the model of the state includes the history of inputs and outputs of the system (See also Focus (p.11-19)).

Meta-MSC Inspired by Focus (p.11-19) and Temporal logic (CTL) (p.11-17) we specify Liveness and Safety descriptions in an elaboration of MSC-96 called Meta-MSC.

Figure 11-26: Liveness in Meta-MSC

Open figure



Here we specify an expression with three operands. The idea is that whenever the Assumption holds, the Commitment will always happen before the Conclusion happens.

Our basic model is that there is an MSC document which describes the system completely. Alternatively we may assume that this MSC document could be produced by executing an object model (in SDL e.g.). The Meta-MSC diagrams are associated with this complete system of MSCs. The semantics of the MSC document is a set of simple

traces of actions. First select the subset of simple traces which starts with traces of Assumption. Corresponding to every trace in Assumption there is a subset of traces in the MSC document. Every such subset should match all traces in the commitment restricted by the conclusion.

To make the Meta-MSC more versatile we introduce a wildcard notation. A special MSC reference has the name **any** (possibly appended a set of MSC names) which means that the MSC reference may refer any sequence legal at this point in the MSC document traces.

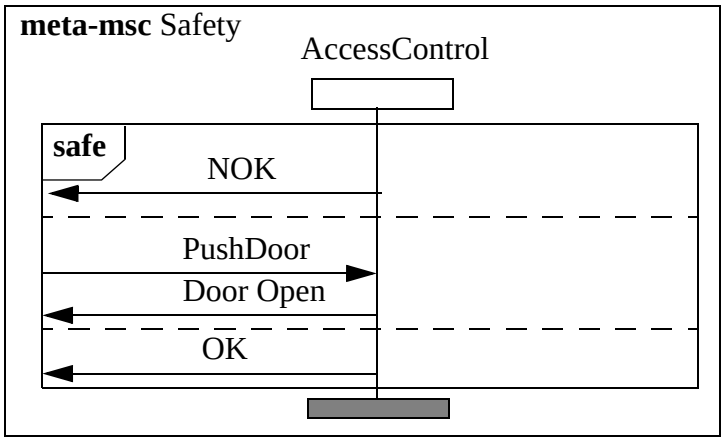
We also introduce a notation for a wildcard message: "**any** name {message set}" where the message set is optional. Finally there is a wildcard instance notation as well: "**any** name {instance set}". The names of the wildcard notation are only meant to be explanatory text. For an example see Figure 11-7 "Liveness in MSC" (p.11-12).

We also introduce a safety construct, which is identical in syntax to the liveness construct, but the operator is called **safe**. The interpretation is that whenever the assumption is true, the commitment should never happen until the conclusion happens.

More specifically, for the set of simple sequences covered by the basic MSC document which are matched by the assumption, their continuations should not be covered by the commitment of the safe-construct up to the conclusion.

Figure 11-27: Safety in Meta-MSC

Open figure



The example in Figure 11-27 "Safety in Meta-MSC" (p.11-43) shows the safety property that after a NOK response the door shall not be possible to open before another positive response (OK) has been received.

Matching of the sequences are done by matching global conditions and event sequences. Whenever a global condition is initial condition to a live- or safe-construct in Meta-MSC, it is necessary that the matching MSC sequences have that condition as initial condition. If the Meta-MSC has no initial condition, this matches any condition (or no condition). See MSC-96 Tutorial for further treatment of conditions.

If there is a desire to use standard MSC-96 tools to edit the Meta-MSC document then this can easily be simulated by the following simple means:

- Use msc instead of meta-msc as keyword for the headings.
- Use seq-operator and add a comment about the fact that it is actually either a live- or a safe-construct.
- Use “any_” as a prefix to wildcard message- and instance names. Define the “any” MSC as **empty**.

To combine Meta-MSC constructs, the **seq**-operator corresponds to logical **and**, while the **alt**-operator correspond to logical **or**.

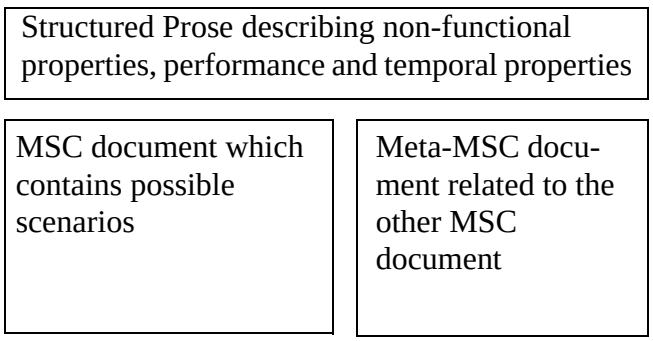
While an automatic validator fairly easily may find out whether a given MSC is possible from a given execution state, finding our about the truth of a Meta-MSC construct is much more complex. In general it requires an exhaustive simulation since it addresses all sequences matching the assumption.

Summary of property modelling methodology

Let us now summarize what submodels a property model may consist of in the SISU methodology:

Figure 11-28: Submodels of Property Modelling

Open figure



These are all described properties. We may also derive properties which e.g. could comprise structural roles used in validation and for documentation purposes.

The property models should be mostly service oriented.

On the constructive use of MSC

Our methodology advocates a combined approach where property modelling and object modelling proceed hand-in-hand achieving the optimum progress of common understanding and system development.

There are a few basic techniques to apply when the different descriptions are combined.

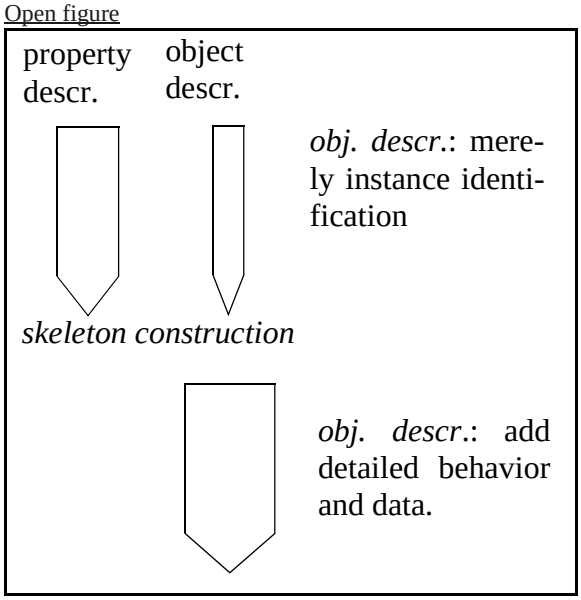
The purely property-oriented approach

The purely property-oriented approach is characterized by the idea that the property descriptions will be complete enough and that the imperative implementation-oriented object descriptions can be almost automatically derived from the property descriptions through what we may call *skeleton construction*.

We have sketched the purely property-oriented approach to modelling in Figure 11-29 (p.11-45).

Figure 11-29: Skeleton construction

Open figure



Given that we use MSC as our main property modelling language, the property descriptions cannot become entirely sufficient for the production of implementation code, but we may produce skeletons which may be extended to complete object descriptions quite easily.

The problem with the purely property-oriented approach is that the technique of producing skeletons are normally not so easily repeated during the development and maintenance phases since the skeletons have been supplemented by object-oriented descriptions and it may not be obvious where the supplements fit in with a newly generated skeleton.

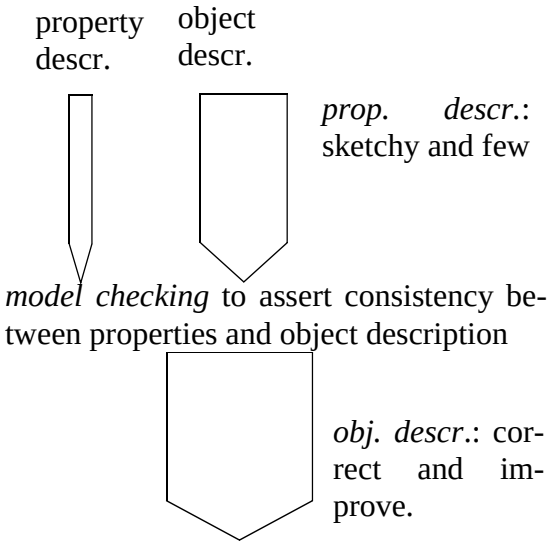
The purely object-oriented approach

The purely object-oriented approach is characterized by the emphasis on the object modelling. The property descriptions are considered important, but minor in relation to the total development. The role of the property descriptions is to serve as “check-lists” that the object descriptions should adhere to. The property model is “model checked” against the object model. The model checking will involve automatic means as well as human intervention.

We have sketched the purely objectr-oriented approach to modelling in Figure 11-30 (p.11-46).

Figure 11-30: Object orientation

Open figure

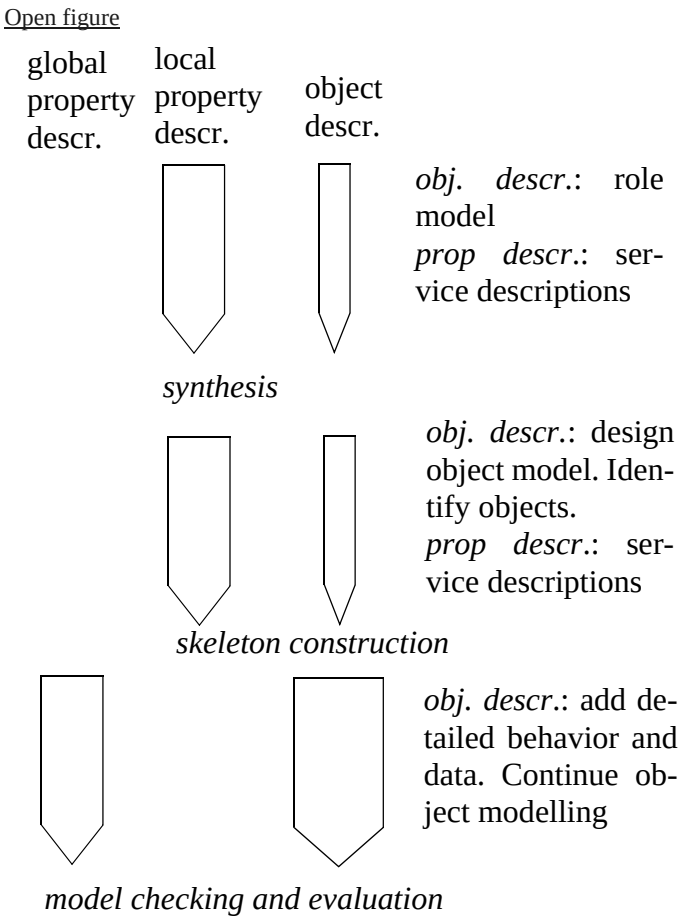


The result of the property evaluation may be that the object model is changed and improved.

An important specialization of the purely object-oriented approach is the *evaluative approach* where the property description is a set of general properties which are common to a large domain of systems. They appear as guidelines for how the system should be specified. The model checking amounts to checking that the guidelines are followed in the object model. The evaluation results in warnings of possible problems.

The combined approach

Figure 11-31: Combined method



The combined approach depicted in Figure 11-31 (p.11-47) tries to reap the benefits of both the property-oriented and the object-oriented approches. While local property descriptions (like MSC) serve well as basis for skeleton construction, more global property models (e.g. with temporal logic) perform better as input to the model checking activity.

The combined approach also combine well with projected approaches like role modelling (see also Summary of property modelling methodology (p.11-44))

Construction of SDL Skeletons from MSC

Here we present a simple, but effective way to produce an object model skeleton (in SDL) from property descriptions (in MSC). The technique will be illustrated by a simple example.

Resolve aliasing of instances

If role modelling or other projection oriented techniques have been applied, the synthesis of roles should be performed. This means that the instance aliases (their roles) should be identified and the MSC diagrams changed such that the aliases (role names) become subordinate to the instance names (e.g. as comments).

Group different MSCs

This activity could also be called: make MSC-96 diagrams which represent the different central behavioral patterns. In the example we consider only simple sequencing and alternatives.

Identify omitted parts

Sketchy property descriptions often have parts that are omitted. A normal assumption for skeleton construction is that no important control information is omitted. Our point here is that a minimum effort is to make the omissions explicit. Omitted parts can be included in the MSCs as comments or actions.

Add (local) conditions

Adding (local) conditions will make the construction more effective and omitted parts may be circumvented.

Produce SDL transitions from MSC instance axis segments

From one instance axis of the grouped MSC, we may produce an SDL process which will produce that MSC (given that all the other instances behave properly).

Table 11-2: MSC to SDL translation (sdl(x))

R	MSC constructs: X	SDL counterparts: sdl(X)
0	sdl(Initial condition (name1)•X)	Start • State(name1)• sdl1[name1](X)
1	sdl1[name1](input • {output}* • Condition (name2) • X)	input • {output}* •State (name2) • sdl1[name2](X)
2	sdl2[name1](input •X)	State(newname) • input • sdl2[newname](X)
3	sdl1[name1](set • X)	set[a] • State(name1)• sdl(X)
4	sdl2[name1](timeout • X)	State(newname) • input of timer • sdl2[newname](X)
5	sdl2[name1](reset • X)	reset • sdl2[name1](X)

Table 11-2: MSC to SDL translation (sdl(x))

R	MSC constructs: X	SDL counterparts: sdl(X)
6	sdl1[name1](**alt** (input1•X, input2•Y))	State(newname) • (((input1• sdl2[newname](X), (input2• sdl2[newname](Y))
7	sdl2[name1] (**alt**(output1•X, output2•Y))	Decision(**any**) • (sdl2[name1] (output1•X), sdl2[name1](output2•Y))
8	sdl1[name1](output • X)	output[b] • State(name1)• sdl1[name1](X)
9	sdl2[name1](output • X)	output • sdl2[name1](X)
10	sdl2[name1] (**alt**(Condi-tion(name2)•X), Condition(name3)•Y)	Decision(name2,name3)[c]• (sdl2[name1](X), sdl2[name1](Y))
11	sdl1[name1](proccall • X)	proccall1[d] • State(name1)• sdl1[name1] (input•proccall2•X)
12	sdl2[name1](proccall • X)	proccall • sdl2[name1](X)
13	sdl2[name1](Condi-tion(name2) • X)	State(name2) • sdl1[name2](X)

a. The ‘set’ must be placed before every occurence of State name1.
b. The ‘output’ must be placed before every occurrence of State name 1.
c. The decision is on a variable ranging over the condition names found in alternatives directly following the alternatives’ start.
d. Proccall is the sequence of Proccall1•input•Proccall2. There are no inputs in Proccall1.

Legend for Table 11-2 (p.11-48)***:***

The rules are numbered. Each rule takes an MSC construct and transforms it into the corresponding SDL. The MSC constructs differ in their prefixes. Every argument has an unknown tail X (and sometimes Y) which normally appears also in the SDL such that the definition is recursive. The dot operator “•” means sequencing, curly brackets and a star “{xxx}*” means that xxx is repeated 0 or more times, “alt(v,u)” represents the MSC-96 alternative expression. Otherwise names of constructs have been used such as Condition, State, input, output and decision.

Resolve indeterminacy on transitions

The straight forward algorithm will often result in a process where transitions are not uniquely determined, i.e. the pair (state, input) is not sufficient. The indeterminacy will be resolved either by an SDL decision or by joining anonymous states. Anonymous states are those which have received their name through the translation of Table 11-2 (p.11-48) through rules 2 or 4.

The idea is of course that there is an underlying SDLprocess which has only one transition where the algorithm indicates several. We must “unify” the alternatives. We will give an informal algorithm which unifies two alternatives. The procedure may be repeated if there are more than two alternatives originally.

We will compare the two alternatives from the input and produce a combined version

Table 11-3: Transition fragment unification (unify(X,Y))

R	Transition 1 X	Transition 2 Y	Unified transition unify(X,Y)
I	v • X	v • Y	v • unify(X,Y)[a]
II	task 1 • X	task 2• Y	(task 1,task 2[b]) • unify(X,Y)
III	output1 •X	output2 • Y	decision(any) • (output1 •X, output2 •Y)
IV	State1 • input1• X	State2 •input2•Y	State• (input1•X, input2•Y)[c]
V	set • State1•input1•X	State2 • input2 •Y	set• unify (State1• input1•X, State2• input2• reset• Y)

a. The v represents a common preamble of the two transitions. That there is a common preamble of at least the input is the assumption of this unification.

b. The two tasks (which originate from informal MSC actions and therefore merely contains informal text) should be informally joined.

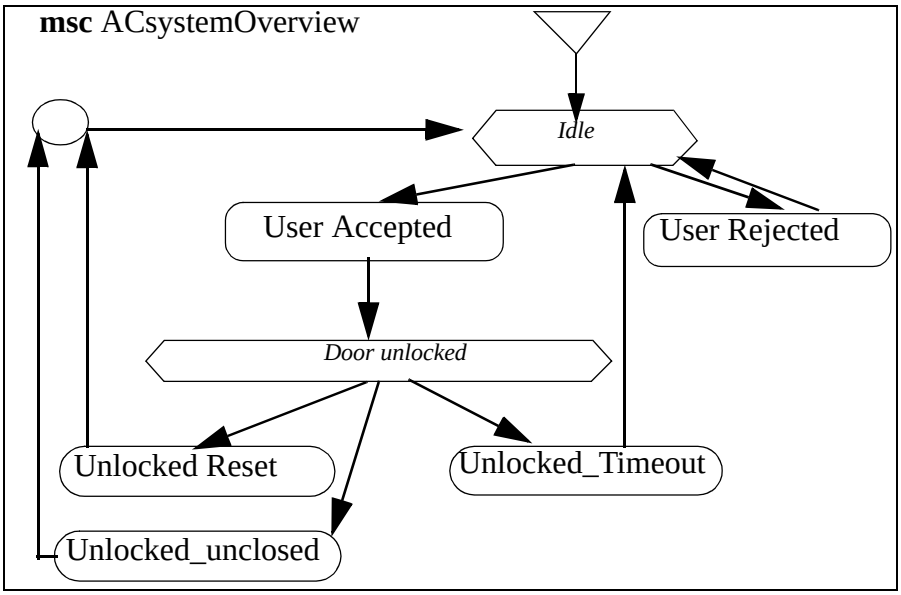
c. State1 and State2 are both “anonymous” and therefore unified to one anonymous name. If one of the states has a given name (coming from a condition in MSC), the unification results in that name.

A worked out example

We take as our starting point the MSCs of the MSC-92 methodology. Their combination is given by the road map of Figure 11-32 (p.11-51).

Figure 11-32: AC System Overview

Open figure



The individual simple MSCs are given by Figure 11-33 "User accepted" (p.11-51) and Figure 11-34 "User rejected" (p.11-52), with the continuations Figure 11-35 "Unlocked reset" (p.11-52), Figure 11-36 "Unlocked timeout" (p.11-53) and Figure 11-37 "Unlocked unclosed" (p.11-53)

Figure 11-33: User accepted

Open figure

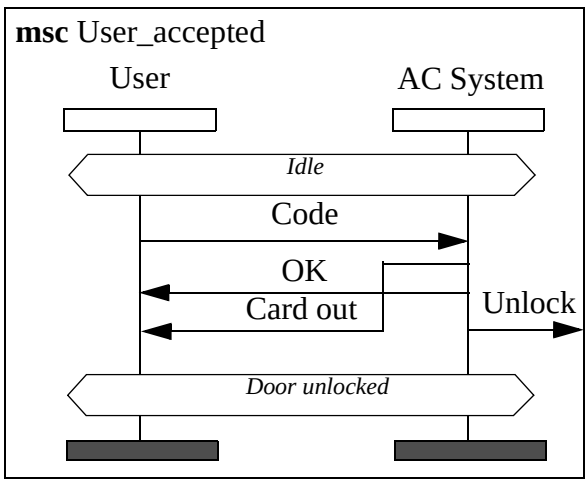


Figure 11-34: User rejected

Open figure

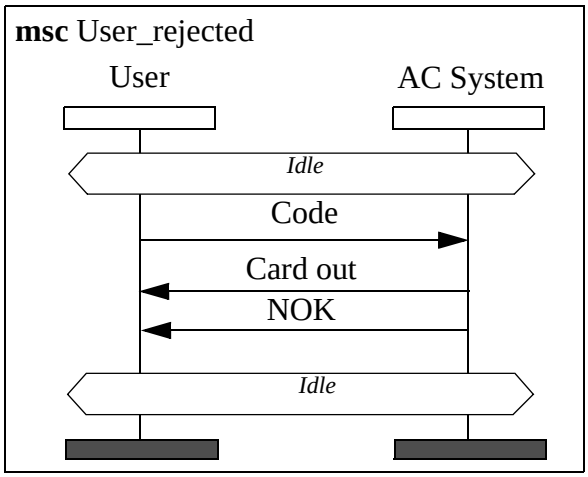


Figure 11-35: Unlocked reset

Open figure

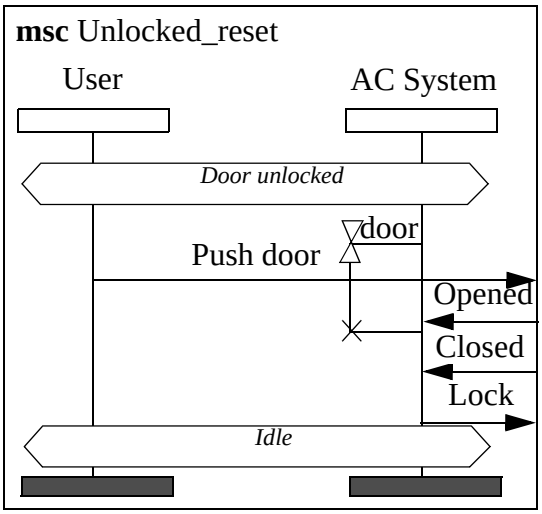


Figure 11-36: Unlocked timeout

Open figure

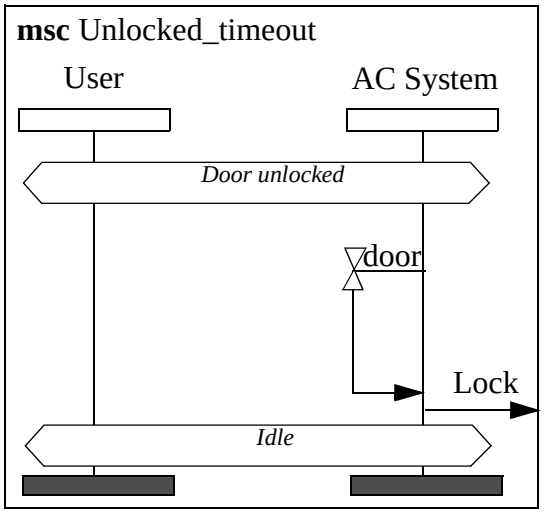
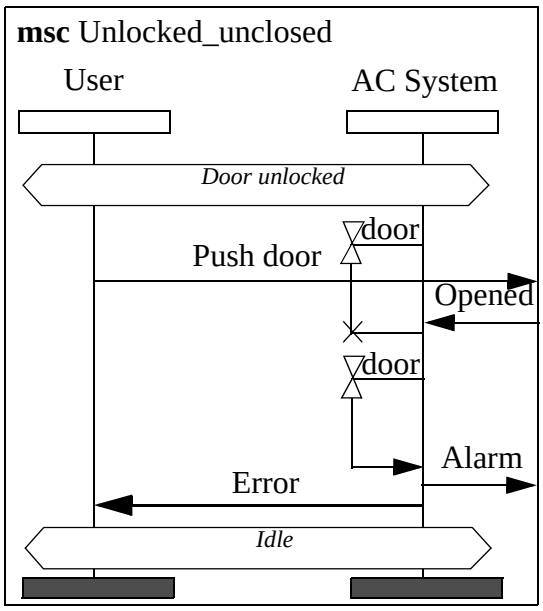


Figure 11-37: Unlocked unclosed

Open figure



By using the rules given in Table 11-2 "MSC to SDL translation (sdl(x))" (p.11-48) we reach the SDL graph Figure 11-39 "SDL process AC system" (p.11-55) which we see has two occurences of ambiguous transitions (Idle/Code) and (Door Unlocked/ Opened). These occurences must be resolved by the rules of Table 11-3 "Transition fragment unification (unify(X,Y))" (p.11-50).. The result of the unifications are given in Figure 11-38 "Unified SDL process AC system" (p.11-54).

Figure 11-38: Unified SDL process AC system

Open figure

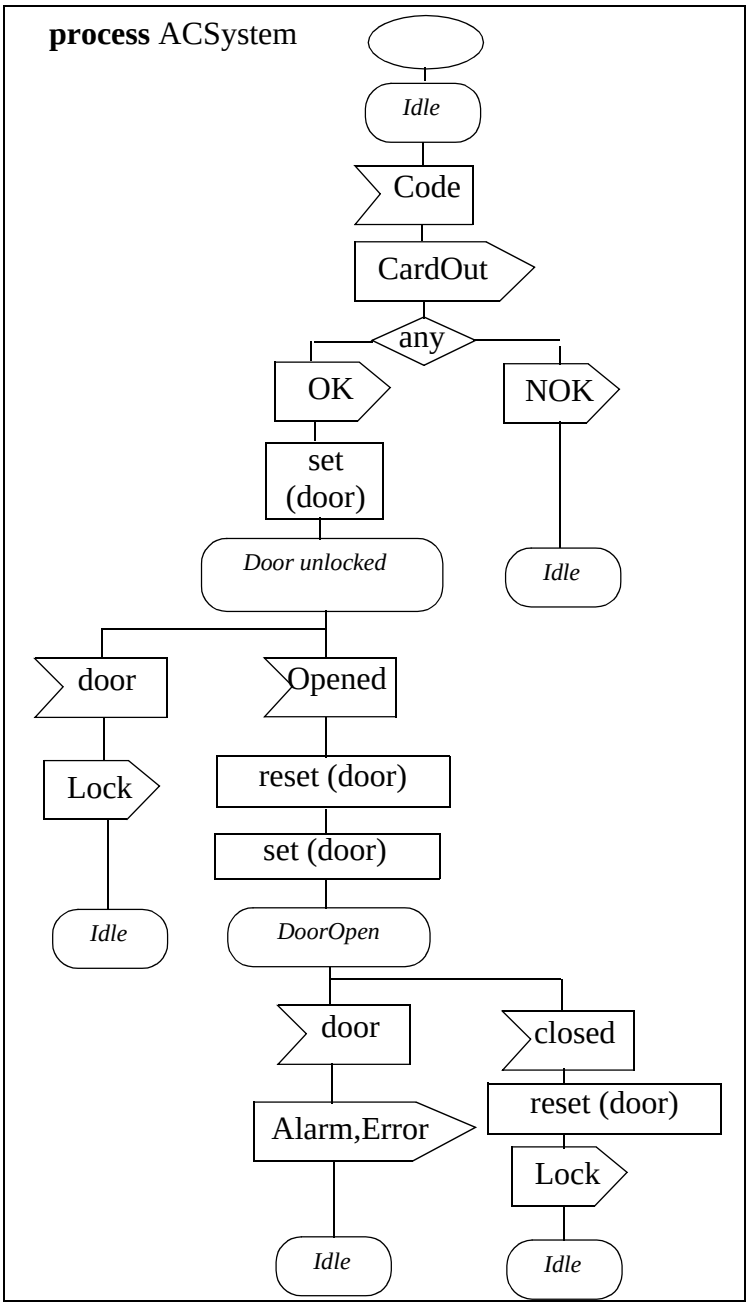


Figure 11-39: SDL process AC system

Open figure

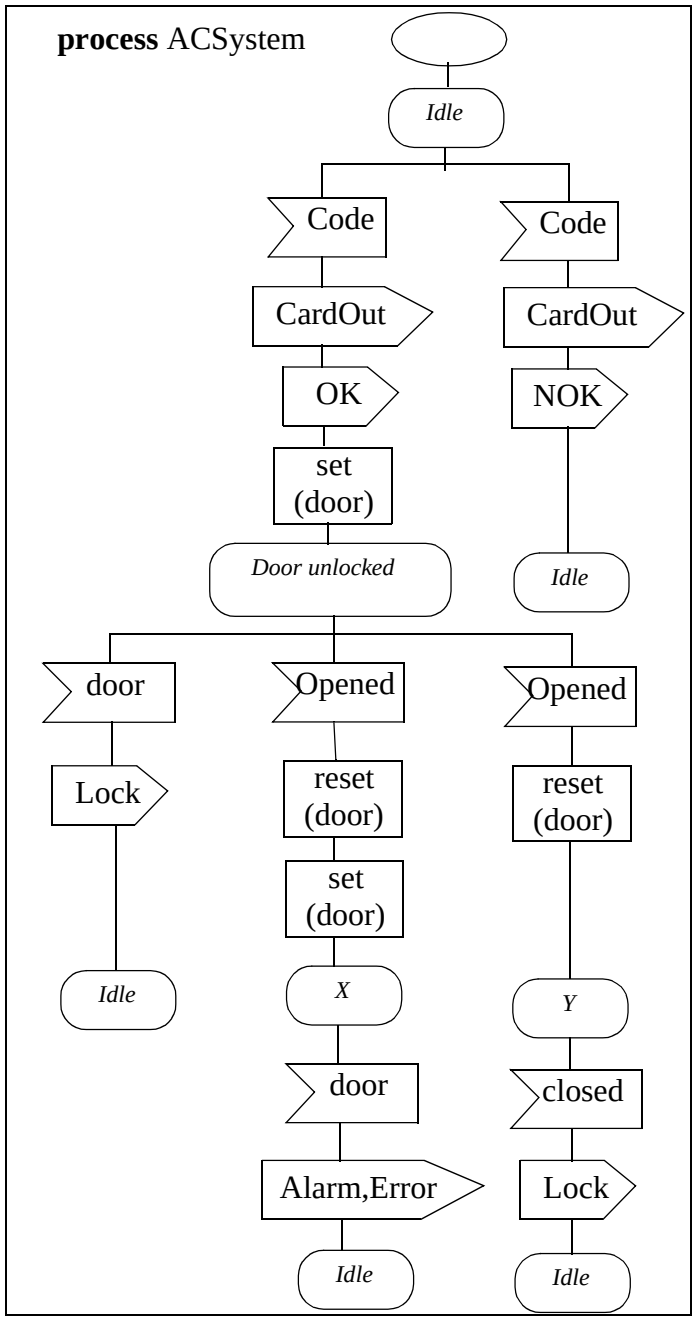


Figure 11-40: We may go through this transformation in more detail:

1. To resolve the two transitions (Idle/Code), we first apply rule I where the common preamble is (input(Code) • output(CardOut)).
2. We are now left with unifying (input(OK),set(door), State(Door unlocked)) and (input(NOK), State(Idle)). The difference lies in different output and we apply Rule III introducing a decision. This condludes the unification of these transitions.

3. Then we want to unify the two transitions given by (Door unlocked, Opened). The common preamble is (input(Opened), reset(door)) and we apply Rule I.
4. We are then left with unification of (set(door), state(X), input(door), output(alarm,error), State(Idle)) and (state(Y), input(closed), output(lock), state(Idle)). We apply Rule V and subsequently Rule IV. This concludes our transformation.

We notice some points during and after the production of the SDL process from the MSC.

- Since MSC does not have any formal data concept, the only construction we get when data is involved is the decision containing “any”. In our case, the resolution of the (Idle/Code) ambiguity results in a decision on “any”. In SDL, however, we know that this can be done formally by using data. When we elaborate the skeleton we will introduce a parameter on Code which we will involve in the decision.
- We also notice (more in the SDL than in the MSC perhaps) that when the timeout appears after the door is opened, the alarm triggers, but the state to which the system returns is the *Idle* state. This is an error in the specification since the door is still open and the system should not accept more cards until the door is closed. This acquired insight should be brought back to the MSC description.

What have we achieved when we have completed the SDL process skeleton?

1. We have gained understanding and improved the MSC document accordingly;
2. We have produced an SDL process graph which is executable and can be used for simulation (in this case: early prototyping);
3. We have a combined description (MSC/SDL) which is internally consistent and contains two different perspectives. The description is complete on its level of detail;
4. We have acquired bits and pieces of information which may be practical to apply in the more detailed descriptions to come in the system development.

List of figures

List of definitions

Casting

Casting is the process of associating roles with their acting objects.

The origin of the word is in theatres where roles are played by actors and they comprise the cast of a performance.

Declarative

An declarative description is a description which focuses on how things *are* rather than how they are *achieved*.

From Webster:

declarative: making a declaration : DECLARATORY

See also imperative.

Distillery

Distillery is originally where hard liquor is being made. To *distill* means to separate some substance from some other substance. It may also mean to purify.

Here we use *description distillery* to mean the process of purifying the description through separating the precise whole from its constituents.

Expressiveness

means that the language can describe the important aspects of the system.

From Webster:

expressive: 3: full of expression : SIGNIFICANT

Imperative

An imperative description is a description of the sequence of actions in a procedural and command-like manner.

From Webster:

imperative: 1. Expressing a command or plea; peremptory. 2. Having the power or authority to command or control.

See also declarative.

Interface Role

is a projection of an object behavior onto an interface (a communication line).

From Webster:

Interface: 1. A surface forming a common boundary between adjacent regions. 2. a. A point at which independent systems or diverse groups interact.

Liveness Property

Informally a *liveness property* expresses that something (good) will eventually happen.

Property

a characteristic trait or quality

(American Heritage Dictionary)

Role

is a behavioral pattern which describes how one acting object performs a set of related services.

From Webster:

- 1a: a character assigned or assumed
- 1b: a part played by an actor or singer
- 2: Function

Roles are used to describe properties, and are related to object designs by projection. Roles are used to link properties and objects. Projections are used for synthesis of new objects and for documenting existing objects.

Safety Property

Informally a *safety property* expresses that something (bad) will never happen.

Service

is a unit of behavior which characterizes what a system (or component) provides for the user. A service is normally given a name. Services may be interleaved in time.

From Webster:

4b: useful labor that does not produce a tangible commodity - usu. used in pl. {charge for professional ~s}

Transparency

means that the descriptions can be easily understood without excessive training and study.

From Webster:

Transparent: 4. Easily understood or detected; obvious: transparent lies.

Verify

means to ascertain that a property is true also in the running system (or relative to another description)

From Webster:

1: to confirm or substantiate in law by oath 2: to establish the truth, accuracy, or reality of something