21 Dictionary for TIMe

Abstraction

(Collins): Having no reference to material objects or specific examples.

(Coad,Yourdan): The principle of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate more fully on those that are.

Abstract system

An abstract system is a system which exists in a conceptual, abstract world.

Abstract systems are composed from abstract components.

Action

According to Collins (1986) an action is:

- something done, such as an act or deed.

In TIMe an action is seen as an occurrence of an activity during some development process. It takes a specific state of input (a milestone) and produces a specific state of output (another milestone). The general rules and guidelines that should be followed during an action are described for the activity it is an occurrance of.

Active object

The purpose of active objects is to take care of transformations and control the need to *perform*. They are justified more by what they do than by what they represent. Their behaviour is often detailed and related to physical processes. A call handling process in a telephone system, is one example. It interacts with physical users and controls physical connections.

Actor

An actor is a stake holder that takes actively part in the services or work processes of a Domain.

Actual gate

The message gates are used when references to the MSC are put in a wider context in another MSC. The actual gates on the MSC reference are then connected to other message gates or instances. Similar to gate definitions, actual gates may have explicit or implicit names.

A message gate always has a name. The name can be defined explicitly by a name associated with the gate on the frame. Otherwise the name is given implicitly by the direction of the message through the gate and the message name, e.g. "in_X" for a gate receiving a message X from its environment.

<actual gate area> ::=

<actual out gate area> | <actual in gate area> |

<actual order out gate area> | <actual order in gate area>

<actual out gate area> ::=

<void symbol> [**is associated with** <gate identification>]

is attached to <msc reference symbol>

[**is attached to** { <message symbol> | <lost message symbol> }]

Note: The <actual out gate area> is attached to the open end of the <message symbol> or <lost message symbol>.

<actual in gate area> ::=

<void symbol> [**is associated with** <gate identification>]

is attached to <msc reference symbol>

[**is attached to** { <message symbol> | <found message symbol> }]

Note: The <actual in gate area> is attached to the arrow head end of the <message symbol> or <found message> symbol.

(more)

<actual order out gate area> ::=

<void symbol> [**is associated with** <gate identification>]

is attached to <msc reference symbol>

is followed by <general order area>

<actual order in gate area> ::=

<void symbol>[**is associated with** <gate identification>]

is attached to <msc reference symbol>

is attached to <general order area>

Aggregation

All non-trivial systems are composed from components. The process of putting components together to form a whole is called aggregation. Aggregation enables us to associate a single concept and a name with a composite object. This helps to simplify matters considerably when we are dealing with the object as a whole. But to build the object and use it correctly we need to understand what it consists of.

An aggregate is an object in itself and the part objects are parts of this object only. This is in contrast to aggregation just by using ordinary relations.

The opposite process of decomposing a whole into parts is called partitioning (or decomposition).

We distinguish between relation aggregation and real aggregation.

Alternative

The **alt** operator defines alternative executions of MSC sections. This means that if several MSC sections are meant to be alternatives only one of them will be executed. In the case where alternative MSC sections have common preamble the choice of which MSC section will be executed is performed after the execution of the common preamble.

Application

An application is an abstract system that provide the main services of a system and is therefore the most valuable part of a system from a user point of view.

Architecture

An architecture is an abstraction of a concrete system representing:

- the overall structure of hardware identifying at least all physical nodes and interconnections needed to implement an abstract system;
- the overall structure of software identifying at least all software nodes, software communications and relations needed to implement an abstract system (in terms of processes, procedures and data).

asterisk state

An asterisk state is a shorthand for all states except those listed in an accompanying asterisk state list.

The state names in an asterisk state list must be distinct and must be contained in other state list in the enclosing body or in the body of a supertype.

Z.100

Attribute

is an aspect of a Configuration Item that gives additional information, e.g. about its functionality.

The attribute is not part of the Identification of the item.

Attribute specification

Attributes are specified by means of a name and a type:

<attribute name> [: <type identifier>]

It is allowed to drop the type information on attributes and just give the attribute name.

In UML, types being used for defining attributes by themselves can be defined as classes, they may be predefined classes or they may be predefined simple types like Integer, Text, etc.

Click here for an example.

Attributes

Attributes of objects are “value” properties that are not covered by part objects (aggregation). Attributes are defined by a name and a type. In Domain Object Models this is informally specified, but it is still worthwhile to use a type that will be defined as an attribute type or class in the Design Object Model.

For the specification of attributes in UML, see attribute specification in UML.

For the specification of attributes in SDL, see variable definition in SDL.

Automaton

An *automaton* is an abstract machine which can be in a set of *states*. It takes a stream of *input symbols*. The consumed input symbol and the state together determines which actions the automaton takes. After the actions have been performed the automaton enters another state. The passage from one state through the consumption of an input symbol to another state is called a transition.

Finite State Machine (FSM) is an automaton where the state space and input alphabets are finite.

In our methodology, SDL uses FSMs as their theoretical base for describing interaction processes.

Baseline

is the designation of a “snap-shot” (typically in time) of a product or system, with a specification of all Identifications of all Configuration Items that are part of it.

A baseline may also have a more specific definition, implying that all the configuration items included in the baseline have a certain Status.

Behaviour associated with an object model

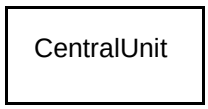
If a class of objects has been identified as part of the object modelling, then it is possible to associate behaviour with objects of this class. If some behaviour has been identified without being associated with any object or class (but a role), then it is possible to associate it with classes later or combine it with other behaviour specifications to new roles or classes.

Depending upon the nature of the behaviour that is desirable to express, it is either expressed in terms of MSC or in fragments of SDL process graphs. The latter may be applicable if the analysis is based upon existing specifications in SDL or in case it is desirable to specify behaviour properties like “instance of type AccessPoint” shall always (that is in any state) accept a Log signal and respond to the Logger with the current status of the point”.

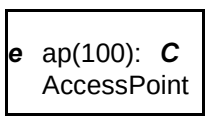
block

A block is a container of processes (or of blocks, that in turn may contain processes or blocks etc.). Processes of a block are contained in process sets that are connected by signal routes.

A block is created as part of the creation of the enclosing block or system. All blocks are created as part of the system creation, that is there is no dynamic creation of blocks.



A block is specified either directly (singular block), like *CentralUnit,* or as a block set according to a block type. The block set *ap* is not a reference (as *CentralUnit*). Instead it designates a set of block instances. The example here specifies a set of 100 blocks of type AccessPoint.



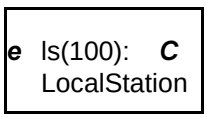
In the latter case, the AccessPoint must have been defined as a block type, as shown here:

The block *CentralUnit* is defined in a separate block diagram, while the properties of the blocks in the *ls* block set is defined by the block type *LocalStation*. A block type is defined by a block type diagram. To see a block type defined in terms of a substructure of blocks, look at block type diagram of AccessPoint with block substructure.

Z.100

block set

Type-defined blocks are contained in block sets. A block set is a fixed number of blocks with properties according to a block type.



The set of LocalStations is called ls and the number (100) designates the cardinality of the set. All the block instances within a block set typically have the same relationship with its surroundings (given by the channels).

A channel connected to a block set (via the gates e or C) will actually represent a set of channel instances.

A block set is not an array, so the thirteenth block cannot be identified by e.g. ls(13). The number of elements in a block set is determined when the system is created, all blocks in the set are created as part of the creation of the system, blocks will be permanent part (instances) of the system instance, and sets of blocks cannot be created dynamically.

Z.100

block type

A block type defines the common properties for a category of blocks.

Block types are defined in block type diagrams, and these may be referenced by means of block type references.

Block types may contain a connectivity graph of block instances connected by channels. This makes up a structure of nested blocks. At the leaves of this structure there are blocks which contain processes. In SDL, block types may not contain both blocks and processes at the same time.

In addition to containing structures of blocks or structures of processes, block types may contain other type definitions. This makes up the scoping hierarchy of SDL. Names in enclosing type definitions are the only names visible.

Block types may contain data type definitions, but no variable declarations. This follows from the fact that processes in SDL do not share data other than signal queues. They share a signal queue in the way that one process appends (output) signals to the queue (the input port), while the other process consumes (input) signals from the same queue. Appending and consuming signals are atomic, non-interruptible operations. The input port is the basic synchronisation mechanism of SDL.

Block types may contain process types, service types and procedures as well as block types and data types.

Z.100

block type diagram

A block type diagram defines the properties of a block type.

Z.100

block (type) heading

The heading of block diagrams defines the name of the block.

The heading of block type diagrams defines the name of the block type, possible formal context parameters, whether the block type is virtual or not and if it inherits from another block type.

Z.100

block type reference

Block types are defined in block type diagrams, and they are referenced by means of block type references. The block type reference indicates in which block or system scope unit the block type is defined.

Z.100

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.

channel

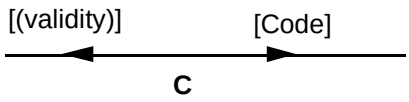
A channel is a one-way or two-way directed connection. It is characterised by the signals that it may carry; these constitute the signal list(s) of the channel. A channel has a signal list for each direction.

One or two arrows indicate the direction(s) of the channel.

Channels connect blocks or block sets with other blocks or block sets, or with the environment of the system. It provides a (one or two way) communication path for signals. If there is no channel between two blocks, then processes in these two blocks cannot communicate by signal exchange. Processes may, however, communicate by means of remote procedure calls without channels connecting the enclosing blocks. A channel cannot connect a block or block set with itself.

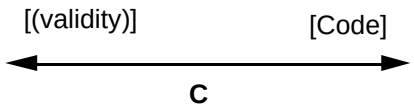
Channels may be delaying or non-delaying.

A *delaying* channel is specified by a channel symbol with the arrows at the middle of the channel:



The delay of signals is non-deterministic, but the order of signals is maintained.

A *non-delaying* channel is specified as follows, that is with the arrows at the endpoints:



Associated with each direction of a channel are the types of signals that may be conveyed by the channel. The list enclosed by the signal list symbol can be signals (as e.g. Code) or signal lists (as e.g. validity) enclosed in ().

Channels connected to the frame symbol represent the connections to the environment.

Z.100

Class diagram

A class is defined by a class diagram: a frame symbol with a heading (name and possibly inheritance specification), attributes, operations and a contents.

If it is desirable to specify the inheritance as part of the class symbol and not graphically, then the following heading is used, instead of just the class name:

<heading>::= <class name> [<inheritance>]
<inheritance>::= **inherits** <class identifier>

Example:

Class with constraints on its environment

Classes are often defined with a specific purpose in mind, and especially for the behaviour of a class (typically becoming a process type in SDL) it is necessary to know what other processes will be in the environment. This is typical for the scenario with several equally “important” objects that have to co-operate in order to do a task. It will, however, reduce the reusability of the class in other contexts where these other objects will not be. A quite different scenario is the specification of a typical “server” object class that should work in any context and where the behaviour is independent on the behaviour of the client objects.

A specification of a class with constraints on it environment contains the following elements:

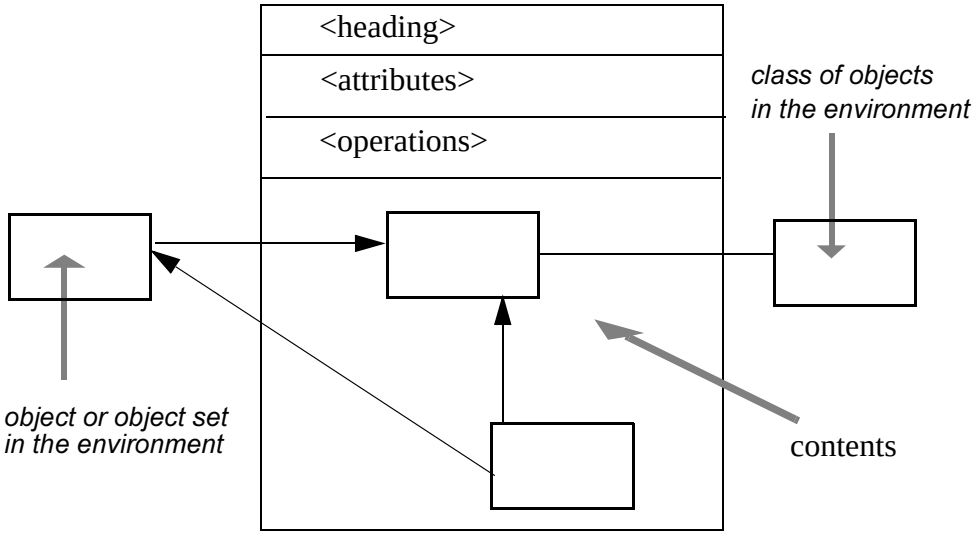
- The class definition in focus may contain a definition of the attributes of the class (the intention).
- The environment of a class is important for the understanding of its purpose and constraints. Therefore, the environment of importance has been depicted outside the class. Entities in the environment represent roles.
- When the class is instantiated there will be entities in the actual instance environment that will play the roles. Therefore, all instances must comply with the roles given to them by the other instances.

A class definition may include a prescription of what we consider a valid instance environment. The entities and relations in the environment of a class represent roles that shall be played by actors in the environment of an instance of the class.

Classes defined by means of aggregation

A crucial point in UML (and in the TIMe method as a whole) is the notion of real aggregation. Most existing notations do not support this, but only a special containment relation.

The notation for aggregation is to take the existing notation for an object or class and extend it with a contents that contains objects and object sets being related and/or connected.



Complete abstract system

A complete abstract system models as completely as practically possible the abstract functionality implemented in a concrete system.

It covers the Application and the infrastructure functionality supporting the Application. Its behaviour is a valid model of the real behaviour and its structure is similar to the structure of physical nodes in the concrete system.

Concrete system

A concrete system is a real system which is part of the physical world.

In TIMe, concrete systems are composed from physical parts and software that execute to provide services to its users.

Condition (MSC 92)

A **condition** describes either a global system state (global **condition**) referring to all instances contained in the MSC or a state referring to a subset of instances (non-global condition). In the second case the **condition** may be local, i.e. attached to just one instance.

Condition (MSC 96)

A condition describes either a global system state (global condition) referring to all instances contained in the MSC or a state referring to a subset of instances (nonglobal condition). In the second case the condition may be local, i.e. attached to just one instance.

<condition area> ::=

<condition symbol> **contains** <condition name list>

is attached to {<instance axis symbol>*} **set**

Configuration

is a particular composition of a product or (sub)system from particular components (items) with a defined functionality.

Configuration Control

The formal guidelines for

- describing the configuration of a system or product on the basis of the identification and status of each Configuration Item it consists of
- describing the derivation process and rules from source components through derived components to a complete system
- coordinating and approving changes in this description

Configuration Item

is an entity which is subjected to Configuration Management and is treated as atomic (indivisible) in this respect.

A configuration item may consist of parts, but these parts are then not managed as parts according to configuration management (e.g. a printed circuit board may be a configuration item, while the components on it are not subjected to configuration management).

A configuration item is concerned with the (syntactic) descriptions that a system consists of, and not the (semantic) building blocks in the system domain.

Is also informally called part, entity or component.

Configuration Management

The formal guidelines for

- identifying and defining the Configuration Items a system is composed of
- recording and reporting the status of entities and requests for change throughout the components life-span
- evaluating and initiating changes
- controlling the change process
- verifying the release of system versions

Configuration Control Board

A body which is

- responsible for evaluating change requests
- capable of ordering their execution
- capable of monitoring their completion

Configuration Management Plan

A document which describes how Configuration Management shall be carried out in a project or for a product. It describes:

- roles and role responsibilities
- the process for evaluating and implementing change

It defines:

- component statuses and corresponding approval criteria
- identification criteria
- methods for inspection, approval, filing etc.
- types of items to be managed
- tools to be used

Connection Point

The connection points are introduced to simplify the layout of HMSCs and have no semantical meaning.

Connection points are nodes which make it possible to reduce the number of branches since several parallel branches with the same start and end give no additional meaning.

<node area> ::=

<hmsc reference area> | <connection point symbol>

| <hmsc condition area> | <par expr area>

Connections

Objects are connected if they are involved in communication with each other. This is different from objects being related, as this will only imply that the objects may be reached by navigating along the relations.

When using SDL as the design language, connected objects will mainly be objects that will be represented by blocks or processes in SDL.

Constructive part of a description

A constructive part of a domain object or property model description is a part that may be automatically transformed into a corresponding design.

Examples are parts of object models with relations that may be transformed to database schemes; a subtype relation between two types in the domain object model that is transformed to the corresponding relation between the corresponding SDL process types.

Content

The content of an object model consists of a structure of internal entities or a behaviour. The structure may be decomposed over several aggregation levels. The structural component may be instances of types defined in other object models.

Property models associated with the content will specify properties of internal objects and interfaces.

Context

The context of an object model consists of the entity being modelled, considered as a black box, and its environment, where the environment consists of other entities that are known to or that interact with the entity being modelled. This serves to describe the environment and the interfaces as well as other external relationships.

The environment of a type consists of conceptual entities, called roles, relations and connections. The environment of an instance consists of actual entities playing the roles.

By associating property models with the context it is possible to specify the external properties that the object provides, as well as the properties it requires from its environment.

Coregion

The total ordering of events along each instance in general may be not appropriate for entities referring to a higher level than SDL-processes.

Therefore a **coregion** is introduced for the specification of unordered events on an instance. Such a **coregion** in particular covers the practically important case of two or more incoming messages where the ordering of consumption may be interchanged.

Counter counter implementation

[Keen 81]: *How to cope with* counter implementation:

1. Make sure you have a contract for change
2. Seek out resistance; treat it as a signal to be responded to
3. Rely on face-to-face contact
4. Become an insider; work hard to build personal credibility
5. Co-opt users early

Counter implementation

[Keen 81]: *How to oppose a decided change without showing your face:*

1. *Lay low*
2. *Rely on inertia*
3. *Keep things complex, hard to coordinate, and vaguely defined*
4. *Minimize the legitimacy and influence of the change agent*
5. *Exploit the lack of knowledge of the change agent*

Must be met by counter counter implementation.

create

A process may create processes in other process sets in the same block, possibly providing actual parameters to the new instance.

The create line (dashed line with arrowhead) indicates possible creations. Create lines are optional.

Z.100

dashed entity

A dashed entity is the graphical way of representing an entity that is inherited from a supertype and which needs to be used in the definition of the subtype. There are dashed block sets and process sets, services and gates.

The Z.100 terminology is *existing entity*.

An existing block set/block may be connected by channel, and these will then be there in addition to those specified in the super type.

An existing process set/service may be connected by signal routes, and these will then be there in addition to those specified in the super type.

An existing gate can have constraints in terms of signals on the endpoints of the gate specified, and these are then added to the inherited gate and will then apply in addition to those of the inherited gate.

In the PR version of a specification, inherited entities are simply identified by name.

Z.100

decision

A decision transfers the interpretation to the outgoing path whose range condition contains the value given by the interpretation of the question.

Z.100

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.

Description

A description is a statement or account that describes. It is a symbolic representation that enablse communication and reasoning about some subject. Descriptions may be expressed on a variety of media using a variety of languages and notations.

In TIMe, descriptions are contrasted with documents, which are considered as the physical carriers of descriptions.

Design oriented development

An approach to system development where systems are understood and maintained mainly in terms of abstract design description in some notation or language.

Design oriented development is at a lower process maturity level than Property oriented development, but higher than implementation oriented development, where “the code documents the system”.

diagram heading

In the upper left-hand corner of the first page of diagrams, we find the heading.The heading defines the name of the entity, it may contain definition of formal parameters, context parameters, it may specify if a type inherits from another type and the virtuality of atype (virtual, redefined or finalised).

The heading of the first page of a diagram must be a full heading of the form:

<heading> ::= <kernel-heading> [<additional-heading>]

while

the following pages only need a kernel heading:

<kernel-heading>::= [<virtuality>] [exported]

<diagram-kind> [<qualifier>] <diagram-name>

The kernel heading depends upon the diagram kind, see

- system (type) heading
- block (type) heading
- process (type) heading
- service (type) heading
- procedure heading

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.

Document

A document is a piece of paper, a booklet, etc.; providing information esp. of an official nature. In TIMe Documents are physical *carriers* of information. This information may be local to that document, or it may be fetched from descriptions and models (whole or partial models). Documents are often made for specific occasions and audiences, e.g. a contract, a review document, a user manual.

A description or model may appear in several documents, therefore the descriptions or models should be maintained separately from the documents.

A document may be seen as a “snapshot” at a particular point in time. As such it need not be maintained, although it may be.

Domain

The (problem/application) domain models a part of the real world having similar needs and terminology, and where a system instance may be a (partial) solution to some need (the problem). It is not specific to a particular system or system family, but rather to a market segment. It covers common phenomena, concepts and processes that need to be supported to solve the problem, irrespective of particular system solutions.

Note that the domain is like a type; it is a generalised concept covering the common features of many domain instances. Hence the Domain is not a set of occurrences, but a general pattern for one occurrence.

Domain auxiliary descriptions

Domain Auxiliary descriptions will often be informal text and illustrations used to help reading the other Domain Descriptions.

Domain descriptions

A domain description describes a (problem/application) domain.

In TIMe domain descriptions are organised in:

- domain models;
- domain statements;
- domain dictionaries;
- domain auxiliary.

Domain dictionary

A Domain Dictionary is dictionary over common domain terminology.

Domain model

A Domain Model is a formal definition of a Domain expressed in terms of Object Models and Property Models (collections of classes with attributes and relations and associated properties). To fully define a Domain it is possible to use Domain Models on several abstraction levels.

In TIMe, Domain Models are expressed using OMT/UML, MSC and (possibly) SDL.

Domain statement

A Domain Statement is a concise statement about a Domain, and is normally expressed in prose.

Domain Statement

A (problem) domain statement is a concise description of the problem domain with focus on stakeholders and their needs, the essential concepts, functions and work processes, rules and principles. It should also clearly state the nature of the problem, i.e. what one wants to achieve.

entity kinds

SDL defines the folowing different kinds of entities:

- packages
- system
- system types
- blocks

- block types
- channels
- signal routes
- signals
- gates
- timers
- block substructure
- channel substructures
- processes
- process types
- services
- service types
- procedures
- remote procedures
- variables (and formal parameters)
- synonyms
- literals
- operators
- remote variables
- data types
- generators
- signal lists and
- views.

Environment

An MSC describes the communication between a number of system components, and between these components and the rest of the world, called *environment*. It is assumed that the environment of an MSC is capable of receiving and sending messages from and to the Message Sequence Chart; no ordering of message events within the environment is assumed. Although the behaviour of the environment is non-deterministic, it is assumed to obey the constraints given by the Message Sequence Chart.

environment

The environment consists of a set of SDL processes that may send signals to the system and which may receive signals from the system.

Z.100

Environment

Environment is the surroundings of an MSC. When the MSC is placed in a wider context my using MSC references, the communication with the environment from inside the MSC diagram should match the communication with the MSC reference which references it.

The environment is represented by the diagram frame.

Communication with the environment goes through gates.

Event

The instance definition provides an **event** description for message inputs and message outputs, actions, shared and local conditions, timer, process creation, process stop. Outside of coregions a total ordering of **events** is assumed along each instance-axis. Within coregions no time ordering of **events** is assumed.

Expressiveness

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

From Webster:

expressive: 3: full of expression : SIGNIFICANT

Family descriptions

A system family description describes a system family.

In TIMe, family descriptions are organised in:

- family models;
- family implementations;
- family statements;
- family dictionaries;

- auxiliary descriptions.

finalised input

A finalised input is a redefinition of a virtual input transition that cannot be redefined in further subtypes. A virtual input is a special case of a virtual transition.

Z.100

finalised process type

is a finalised redefinition of the corresponding virtual process type in the super block type, and it is **not** virtual, so that it can **not** be redefined in further subtypes of this block type.

A final redefinition of the process type must be a subtype of the type identified in the virtuality constraint.

Z.100 (virtual types)

Formal Semantics

Formal semantics means explaining the meaning of the MSC description by referring to a definition of the language in mathematical (logical) terms.

The formal semantics of MSC-92 is expressed in a process algebra. The point of describing the semantics mathematically is that proofs may be performed automatically and stringently.

Framework

A *framework* is an abstract system or a collection of (large) system component with two parts:

- a redefinable application;
- a configurable infrastructure that takes distribution into account, and contains all additional behaviour and supporting functionality needed to support the application in the concrete system.

Functional property

A functional property is a property which is measurable in an abstract system.

Functional properties characterise the behaviour of abstract systems, and can be measured by observing the abstract system.

gate

A gate is a potential connection point for channels/signal routes when connecting sets of blocks/processes/services. The same symbol is used in all cases.

Gate are defined in block/process/service types and used when connecting sets or instances of these with channels/signal routes.

The signal list associated with the endpoints represents constraints (on incoming/outgoing signals) the gate.

Z.100

General order relation

A general order relation is a binary relation between two message events. It defines a sequencing between the two events which otherwise would not have been defined.

General order relations may also be completed via gates. An order gate connects general order relations of an MSC diagram with an event of another MSC diagram. Order gates must be explicitly named.

Generalisation/specialisation

For classification of concepts we have the notions of generalisation and specialisation. Generalisation is a means to focus on similarities between a number of concepts and to ignore their differences. To generalize is to form a concept that covers a number of more special concepts based on similarities of the special concepts.

The intension of the general concept is a collection of properties that are all part of the extension of the more special concepts. The extension of the general concept contains the union of the extensions of the more special concepts. The inverse mechanism is to specialise: to form a more special concept from a general one.

Note that the exact meaning of specialisation will only be given when it is applied in a formal language. When using specialisation in the domain object modelling it is recommended to use it in a way that will not be very different from the meaning in the design.

Helpers

These are general tools that are used by the actors to provide the services of a Domain. Examples are communication systems, radar equipment and keys.

HMSC start

The graph describing the composition of MSCs within an HMSC is interpreted in an operational way as follows. Execution starts at the <hmsc start symbol>. Next, it continues with a node that follows one of the outgoing edges of this symbol.

Identification

is an unambiguous designation of a Configuration Item that is part of a system or product.

The identification can consist of a name, type, revision number and variant.

The identification can not be changed during the life-span of the item.

Physical items will bear the identification.

identifier

An identifier contains an optional qualifier in order to denote the scope unit in which the entity is defined:

<identifier>::= [<qualifier>] <name>

where qualifier defines the path:

<qualifier>::= <path-item>{‘/‘<path-item>;* |

‘<<‘ <path-item>{‘/‘<path-item>* ‘>>‘

The qualifier gives the path from either the system level, or from the innermost level from where the name is unique, to the defining scope unit.

Each path-item have this form:

<path-item>::= <scope-unit-kind>{<name> | <quoted-operator>}

where scope-unit-kind is one of

- package,
- system type,
- system,
- block,

- block type,
- substructure,
- process,
- process type,
- service,
- service type,

(more)

- procedure,
- signal,
- type, or
- operator.

A definition in an inner scope unit overrides definitions with the same name in outer scope units. Qualifiers may be used in order to identify overridden entities.

Qualifiers may be omitted if not needed in order to identify the right entity in the right scope unit.

States, connectors and macros cannot be qualified. States and connectors are not visible outside their defining scope unit, except in a subtype definition.

Illustrative part of a description

An illustrative part of a domain object or property model description is a part that is not automatically transformed into a corresponding design.

Implementation

Implementations are detailed and precise descriptions of the hardware and the software that a concrete system is made of. They define the physical construction of systems in a system family. The software part will be expressed in programming languages such as C++ or Pascal, while the hardware part will be expressed in a mixture of hardware description languages such as circuit diagrams, cabinet layout diagrams or VHDL.

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.

Incomplete messages (lost and found)

The loss of a message, i.e. the case where a message is sent but not consumed, may be indicated by a black hole.

Symmetrically, a spontaneously found message, i.e. a message which appears from nowhere, can be defined by a white hole.

<incomplete message area> ::=

{ <lost message area> | <found message area> }

{ **is followed by** <general order area> }*

{ **is attached to** <general order area> }*

<lost message area> ::=

<lost message symbol> **is associated with** <msg identification>

[**is associated with** { <instance name> | <gate name> }]

is attached to <message start area>

NOTE: The <lost message symbol> describes the event of the output side, i.e. the solid line starts on the <message start area> where the event occurs. The optional intended target of the message can be given by an identifier associated with the symbol. The target identification is written close to the black circle, while the message identification is written close to the arrow.

<found message area> ::=

<found message symbol> **is associated with** <msg identification>

[**is associated with** { <instance name> | <gate name> }]

is attached to <message end area>

NOTE: The <found message symbol> describes the event of the input side (the arrowhead) which should be on a <message end area>. The instance or gate which supposedly was the origin of the message is indicated by the optional identification given by the text associated with the circle of the symbol. The message identification should be written close to the arrow part.

input

An input allows the consumption of the specified input signal instance. The consumption of the input signal makes the information conveyed by the signal available to the process. The variables associated with the input are assigned the values conveyed by the consumed signal.

The values will be assigned to the variables from left to right. If there is no variable associated with the input for a sort specified in the signal, the value of this sort is discarded. If there is no value associated with a sort specified in the signal, the corresponding variable becomes “undefined”.

The sender expression of the consuming process is given the PId value of the originating process, carried by the signal instance.

Z.100

Input event

An input event designates the consumption of a message. Normally there is a corresponding output event. The input event follows after the corresponding output event in time.

<message in area> ::= <message in symbol>

is attached to <instance axis symbol>

is attached to <message symbol>

<message in symbol> ::= <void symbol>

The <void symbol> is a geometric point without patial extension. The <message in symbol> is actually only a point which is on the instance axis. The end of the message symbol which is the arrow head is also pointing on this point on the instance axis.

Instance

A Message Sequence Chart is composed of interacting instances of entities. An instance of an entity is an object which has the properties of this entity. Related to SDL, an entity may be an SDL-process, block or service. Within the instance heading the entity name, e.g. process name, may be specified in addition to the instance name.

Instance

An instance is an interacting entity of an MSC. Events are on instances and they are ordered according to their position on the instance from top to bottom. An instance has an instance head and an instance end or a stop. Between these there is the instance axis which may be either a single vertical line or a column defined by two vertical lines.

<instance area> ::=

<instance head area> **is followed by** <instance body area>

<instance head area> ::= <instance head symbol>

is associated with <instance heading>

[**is attached to** <createline symbol>]

<instance heading> ::=

<instance name> [[:]<instance kind>][decomposition>]

<instance body area> ::= <instance axis symbol>

is followed by {<instance end symbol>|<stop symbol>}

Instance descriptions

An instance description describes a system instance.

In TIMe, system instance descriptions are organised in:

- Instance models that formally define the system instance, usually by configuration of some family;
- Implementations which are the instance specific implementations, such as configuration files;
- Auxiliary descriptions that provide supplementary documentation, for instance a test suite.

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.

Language

By a systems engineering language we mean a formal description technique (FDT). This means that not only the alphabet (notation) must be defined, but that both syntax (grammer) and semantics (meaning) of the language must be defined.

Examples of systems engineering languages are SDL, MSC, LOTOS, ESTELLE.

Contrast to Notation.

Liveness Property

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

local variables

Local variables of a procedure become parts of the procedure instance when the procedure is called, and they cease to exist when the procedure returns.

The local variables will get default initial values if nothing else is specified.

Z.100 (variable definition)

Localisation (nesting)

Some phenomena and concepts are only meaningful within the *context* of a specific phenomenon or concept. Localisation of definitions supports this and gives rise to nesting of definitions. Scope rules and binding rules determine how nested definitions may use entities defined in enclosing definitions.

Loop (HMSC)

A loop in HMSC occurs when branches and nodes form a cycle. There are no restrictions on how such cycles should appear.

Message

A message within an MSC represents exchange of information between two instances or one instance and the environment.

A message exchanged between two instances can be split into two events: the message input and the message output. Messages coming from the environment are represented by a message input, messages sent to the environment by a message output. To a message, parameters may be assigned between parentheses. The declaration of the parameter list is optional for the message input.

Method

A method is systematic way of producing some result.

In systems engineering a method provides guidelines for structuring and using descriptions in given notations.

Contrast to Methodology.

Methodology

A methodology is a collection of methods and guidelines for when and how to use them to produce a result.

In systems engineering most results take the form of descriptions expressed using some notation or language. A systems engineering methodology therefore prescribes a set of descriptions and associated methods.

A systems engineering methodology is used by an organisation in an attempt to achieve right quality, short lead times and low cost.

Model checking

Given a model which is (typically described by automata), decide whether a given logical statement (typically describe in some temporal logic) is valid.

In our methodology we use model checking to determine the consistency between an SDL model and an MSC temporal specification.

MSC diagram

A Message Sequence Chart, which is normally abbreviated to MSC, describes the message flow between instances. One Message Sequence Chart describes a partial behaviour of a system.

An MSC describes the communication between a number of system components, and between these components and the rest of the world, called environment. For each system component covered by an MSC there is an instance axis. The communication between system components is performed by means of messages. The sending and consumption of messages are two asynchronous events. It is assumed that the environment of an MSC is capable of receiving and sending messages from and to the Message Sequence Chart; no ordering of message events within the environment is assumed.

<msc diagram> ::=

<msc symbol> **contains**

{ <msc heading> { <msc body area> | <mscexpr area> } }

MSC document

A Message Sequence Chart document is a collection of Message Sequence Charts, and sub Message Sequence Charts, optionally referring to a corresponding SDL-document.

MSC heading

The Message Sequence Chart heading consists of the Message Sequence Chart name and (optionally in the textual form) a list of the instances being contained in the Message Sequence Chart body.

MSC heading

The Message Sequence Chart heading consists of the Message Sequence Chart name.

<msc heading> ::=

msc <msc name>

MSC reference

MSC references are used to refer to other MSCs of the MSC document. The MSC references are objects of the type given by the referenced MSC.

MSC references may not only refer to a single MSC, but also to MSC reference expressions. MSC reference expressions are textual MSC expressions constructed from the operators alt, par, seq, loop, opt, exc and subst, and MSC references.

The actual gates of the MSC reference may connect to corresponding constructs in the enclosing MSC. By corresponding constructs we mean that an actual message gate may connect to another actual message gate or to an instance or to a message gate definition of the enclosing MSC. Furthermore an actual order gate may connect to another actual order gate, or an orderable event or an order gate definition.

<msc reference area> ::= <msc reference symbol>

contains { <msc ref expr> [<actual gate area>*] } **set**

is attached to { <instance axis symbol>* } **set**

is attached to { <actual gate area>* } **set**

MTTF

MTTF is an acronym for Mean Time To Failure. The definition is as follows:

$$\text{MTTF} = \frac{\text{Toal execution tme}}{\text{Number of failures observed}}$$

Total execution time is the execution time accumulated over all installations that run the same product. Thus, if one site has run the product for two months on two computers and another site has run the product for one month on one computer, the total execution time is 2 x 2 + 1 x 1 = 5 months of execution time.

A simple example will show how it works:

$$\text{MTTF} = \frac{40 \text{ months}}{10} = 4 \text{ months}$$

One should avoid the use of MTTF as a reliability measure if the failure rate has varied much over the total execution time.

Non-functional property

A non-functional property is a property which is not measurable in an abstract system.

Non-functional properties can be related to the handling of abstract systems, for instance that they are flexible. More often they are related to the concrete system, and express physical properties such as size, weight and temperature.

Performance, real-time responses and reliability are considered to be non-functional properties in TIMe, since they cannot be measured in the abstract systems.

Notation

A systems engineering notation consists of symbols (an alphabet) that can be used to model or describe a concept or entity.

A notation is less formal than a Language, in that the syntax and/or the semantics are not formally defined.

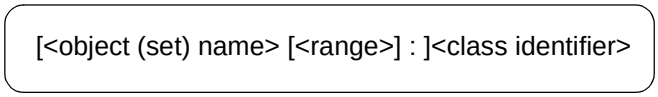
Examples of notations are OMT, UML, ROOM, SA/SD, SADT.

Object classes with attributes, relations and connections

This aspect of object modelling has to do with identification of classes without considering how many instances there will be in a given system and also without considering how they are used in the design of specific systems or other instances.

Objects and Object Sets

While traditional notations only represent set of objects through cardinalities of relations, an object/class in UML may be defined to consist of sets of objects independent of relations, see also aggregation.



An object symbol may represent a single object or it may represent a set of objects. The text inside an object symbol consists of two parts separated by a colon. The first part, which is optional, contains a local object name followed by the number of objects. The second part contains the class name.

The number of objects in a set is specified by a low and a high limit placed inside parentheses (range). For example, (1,100) which means at least one and a maximum of 100 or (3,3) which means exactly 3. The special symbols * and + represent zero or more and one or more, respectively. If no range is specified, then a single object is specified.

(min, max) at least min, at most max

(min,) at least min, no upper bound

(+) at least one, no upper bound

(*) any number, zero or more, no upper bound

Object model

An object model defines static object structures in terms of objects, classes (types), associations and connections, and dynamic object behaviour in terms of signals and state transitions.

These are models that describe how a system or component is composed from objects, connections and relationships, and how each object behaves.

The term object model is a bit misleading, as object models normally describe general types (sometimes called classes) and object sets rather than individual objects. A type is a concept. According to the classical notion of a concept, it is characterised by:

- extension, the collection of phenomena that the concept covers;
- intention, a collection of properties that in some way characterise the phenomena in the extension of the concept;
- designation, the collection of names by which the concept is known.

Representing concepts by types and phenomena by instances of these types follows this pattern: the instances belong to the extension, the type definition gives the intention and the type name represents the designation. The term object model as we use it in TIMe covers objects as well as types.

Object models are constructive in the sense that they describe how an entity is composed from parts, be it abstract or concrete.

In TIMe, every object model should have associated property models.

Operations

Operations are specified by a name and an optional signature:

<operations>::= <operation>*
<operation>::= <operation name>[: <signature>]

Operator

The **alt** operator defines alternative executions of MSC sections. This means that if several MSC sections are meant to be alternatives only one of them will be executed. In the case where alternative MSC sections have common preamble the choice of which MSC section will be executed is performed after the execution of the common preamble.

The **par** operator defines the parallel execution of MSC sections. This means that all events within the parallel MSC sections will be executed, but the only restriction is that the event order within each section will be preserved.

The **loop** construct can have several forms. The most basic form is "**loop <n,m>**" where n and m are natural numbers. This means that the operand may be executed at least n times and at most m times. The naturals may be replaced by the keyword inf, like "**loop <n,inf>**". This means that the loop will be executed at least n times. If the second operand is omitted like in "**loop <n>**" it is interpreted as "**loop <n,n>**". Thus "**loop <inf>**" means an infinite loop. If the loop bounds are omitted like in "loop", it will interpreted as "**loop <1,inf>**". If the first operand is greater than the second one, the loop will be executed 0 times.

The **opt** operator is the same as an alternative where the second operand is the empty MSC.

The **exc** operator is a compact way to describe exceptional cases in an MSC. The meaning of the operator is that either the events inside the <exc inline expression symbol> are executed and then the MSC is finished or the events following the <exc inline expression symbol> are executed. The exc operator can thus be viewed as an alternative where the second operand is the entire rest of the MSC.

output

An output generates a signal of the specified signal type, containing the specified actual parameters, and send this signal instance to the specified destination.

Stating a <process identifier> in <destination> indicates the destination as any existing instance of the set of process instances indicated by <process identifier>. If there exist no instances, the signal is discarded.

If no signal route identifier is specified and no destination is specified, any process, for which there exists a communication path, may receive the signal.

If an expression in the list of actual parameters is omitted, no value is conveyed with the corresponding place of the signal instance, i.e. the corresponding place is “undefined”.

The PId value of the originating process is also conveyed by the signal instance.

Z.100

Output event

An output event designates the output of a message. Normally there is a corresponding input event. The output event must come before the corresponding input event in time.

<message out area> ::= <message out symbol>

is attached to <instance axis symbol>

is attached to <message symbol>

<message out symbol> ::= <void symbol>

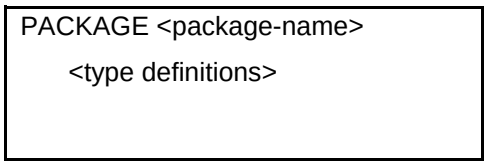
The <void symbol> is a geometric point without patial extension. The <message out symbol> is actually only a point which is on the instance axis. The end of the message symbol which has no arrow head is also on this point on the instance axis.

package

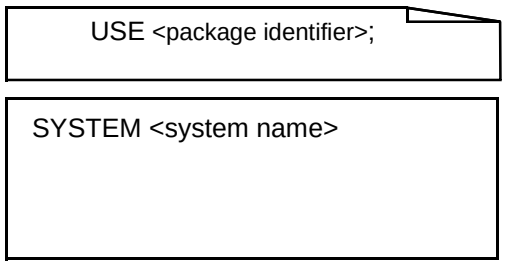
A package is a collection of types. A package is defined by a package diagram. Packages can be provided (that is defined) together with a system diagram (or together with another package diagram) or they can be used by means package identifiers.

A package may contain definitions of types, data generators, signal lists, remote specifications and synonyms. Definitions within a package are made visible to a system definition or other package definitions by a package-reference-clause (use clause). All (or selected) definitions of packages provided in this way will be visible in the system definition (or in the new package).

A package diagram has this form:



A package can be used either either in the definition of a new package, or as here, a system. This is done by the use clause.



Z.100

package reference clause

A package reference clause specifies that a system diagram or package diagram use the definitions of other packages. The names following the "/" after the package name denotes the subset of the definitions that are used.

Z.100

page numbering

A diagram may be split into a number of pages. In that case each page is numbered in the rightmost upper corner of the frame symbol. The page numbering consists of the page number followed by (an optional) total number of pages enclosed by (), e.g. 1 (4), 2 (4), 3 (4), 4 (4).

Passive object

The purpose of passive objects is to *represent* something we need to *know* about. Descriptions of passive objects will abstract from physical details of the entities they represent and model only what we need to know about them. The behaviour of passive objects will normally be very different from the actual behaviour of the objects they represent. A passive object representing a person has a simple behaviour concerned with updating of attributes and relationships (data), while the real person itself has an extremely complex behaviour.

Physical node

A physical node is a distinct physical entity, such as a computer, that implements one or more abstract system objects.

A physical node operates concurrently with other physical nodes.

Physical nodes may be aggregated and decomposed, but always in such a way that abstract objects are contained within physical nodes.

procedure

Procedures define patterns of behaviour that processes/services may execute at several places or several times during their life-time. The behaviour of a procedure is defined in the same way as for processes (that is by means of states and transitions), a procedure may have (local) variables, and in addition it may have IN, OUT, IN/OUT parameters.

Procedures are defined by procedure diagrams.

Z.100

procedure call

A procedure call transfers the interpretation to the procedure definition referenced in the call, and that procedure graph is interpreted.

The interpretation of the transition containing the procedure call continues when the interpretation of the called procedure is finished.

The actual parameter expressions are interpreted in the order given.

If an <expression> in <actual parameters> is omitted, the corresponding formal parameter has no value associated, i.e. it is “undefined”.

Z.100

procedure heading

The procedure-heading of a procedure diagram has this format:

<procedure heading> ::=

[<virtuality>] [<export-as>] **procedure** <procedure-name>
[<virtuality-constraint>] [<specialisation>]
[<procedure-formal-parameters>]
[<result>]

<procedure-formal-parameters> defines the formal parameters of the procedure and have the format:

<procedure-formal-parameters> ::=
fpar [in`/`out | **in**] <typed-parameters>
{, [in`/`out | **in**] <typed-parameters> }*

where <typed-parameters> have the format

<typed-parameters> ::
<variable-name> {‘,‘ <variable-name>}* <data-type-identifier>

<typed-parameters> is a list of parameter names followed by a data type name.

<result> has the format:

<result> ::= **returns** [<variable-name>] <data-type-identifier>

where <data-type-identifier> gives the data type of the value returned by the procedure. The optional <variable-name> can be used to name the result. The result can either be stated as an expression next to the return symbol, or as an assignment in a task to the variable introduced in result.

procedure reference

A procedure reference specifies that there is a procedure in the enclosing entity and that the properties of this procedure are defined in a separate (referenced) procedure diagram outside this diagram.

Z.100

procedure start

Z.100

Process

According to Collins (1986) a process is:

1. a series of actions which produce a change or development;

2. a method of doing or producing something.

In TIMe a process is an ordered series of actions that produce change or development of descriptions and systems. The particular states of descriptions and systems that are produces are called milestones, and the periods of time when they are developed are called phases. Hence a process is an ordered series of actions and milestones related to phases.

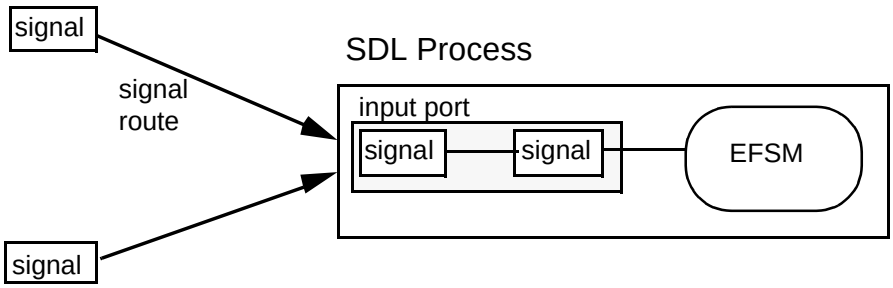
process

A process instance is part of a process set, which in turn is part of a block.

The properties of a process is either defined by a process diagram or it is defined by a process type diagram.

Each process consists of the input port and an extended finite state machine (EFSM) with a sequential behaviour defined by a process graph, which is a sort of state transition diagram. The finite state machine fetches signals from the input port in strict FIFO order except when the order is modified by the save operator (see below). For each signal it performs one transition which will take a short but undefined time.

Signals are messages that the finite state machine consumes. Each signal has a signal type identification which the FSM uses to select the next transition action. In addition, the signal carries the sender identity and possibly some additional data.



An SDL process with signal instances in the input port

process diagram

A process diagram defines the properties of a process set, where each of the process instances in the set have the specified properties.

The behaviour of processes may be defined either by means of a procedure graph (states and transitions) or by means of a substructure of services connected by signal routes. The behaviour of each of the services is defined by means of states and transitions.

Z.100

process (reference)

A process reference specifies that there is a process in the enclosing block and that the properties of this process are defined in a separate (referenced) process diagram outside this diagram.

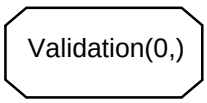
Z.100

process set

A process set defines a set of processes according to a process type.

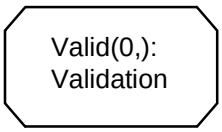
Just like we have the distinction between block reference, block type and block set according to type, we have the distinction between process reference, process type and process set according to a type. Our recommendation is that process sets should be described with reference to a process type.

Process reference:
Process set without
any associated type.



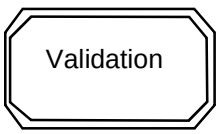
This is both a specification of a process set as part of the enclosing block and a reference to the corresponding process diagram, which defines the properties of the processes in the set.

Process set according
to a process type
(Validation)



The numbers in parentheses after the process set name specify the number of instances in the process set. As defined in above, there are initially no processes, and there is no limit on the number of instances that may be created.

A process set according to a type requires that the corresponding process type is defined:



Z.100

process type

A process type defines the common properties of a category of process instances. A process type is defined by a process type diagram.

process type diagram

A process type diagram defines the properties of a process type.

Z.100

process (type) heading

The heading of process diagrams (defining a process set directly without any process type) is a <process heading>, defining the name of the process set and the initial/maximum number of instances in the set.

The heading of process type diagrams is a <process type heading>, defining the name of the process type, its virtuality (and constraint), its formal context parameters and if it inherits from another process type.

Formal parameters are variables of the process instances. They get values as part of the creation of the process instance.

When a system is created, the initial processes are created in arbitrary order. The formal parameters of these initial processes have no associated values; i.e. they are undefined.

If the initial number is omitted, then the (default) value is 1. If the maximum number is omitted, then there is no limit on the number of instances.

Z.100

Proof

A proof is a systematic sequence of statements aimed at establishing the truth of some given sentence. A proof is often supported by mathematical notation, and based upon formal inference rules. Proofs may also be performed automatically by a computer program, or semi-automatically by the use of proof assistants.

Property

A property is a quality or characteristic attribute, such as the strength or density of a material.

In TIMe we speak of functional/abstract properties and non-functional/concrete properties associated with objects.

Properties are not components that can be used to build systems. They are measures we use to characterise and evaluate systems by. Let us compare to a brick: the brick itself is an object we can use to build something with (e.g. a fireplace), its physical measures are properties we may use to select the particular type of brick and to plan the fireplace, but not to build with. Thus, properties are not components to be used in constructions, but means to understand, select and plan constructions.

Property

a characteristic trait or quality

(American Heritage Dictionary)

Property model

A property model is a model that states properties of a system, a component or a single object without prescribing a particular construction. Property models are not constructive, but used to characterise an entity from the outside. There are many kinds of properties: behaviour properties, performance properties, maintenance properties, etc. This is the perspective preferred by users and sales persons. It is also the main perspective in specifications.

In TIMe properties will be expressed mainly using text and MSCs.

Property oriented development

Property oriented development is characterized by an integration of:

- better product planning through focus on the early stages of system development, in particular domain analysis and requirements specification;
- emphasis on system families, evolution and reuse;
- formal expressions of required and provided properties;
- quality-by-construction through integration of methods for verification, validation and design synthesis.

Property oriented development is at a higher process maturity level than Design oriented development.

redefined process type

is a redefinition of the corresponding virtual process type in the super block type, and it is virtual, so that it can be redefined in further subtypes of this block type.

A redefinition of the process type must be a subtype of the type identified in the virtuality constraint.

Z.100 (virtual types)

Reference expression

MSC references may not only refer to a single MSC, but also to MSC reference expressions. MSC reference expressions are textual MSC expressions constructed from the operators **alt**, **par**, **seq**, **loop**, **opt**, **exc** and **subst**, and MSC references.

The **alt**, **par**, **loop**, **opt** and **exc** operators are described in definition of operator. The **seq** operator denotes the weak sequencing operation where only events on the same instance are ordered.

The **subst** operation is a substitution of concepts inside the referenced MSC. Message names are substituted by message names, instance names by instance names and MSC names by MSC names.

Real aggregation

Real aggregation is supported by UML.

Real aggregation implies:

- that the part object is only part of one object, and
- that possible relations specified with the part object (class) as endpoint only hold for the part object and not for all objects of this class.

UML adorns the association with a filled diamond and calls it *composition*.

Refinement

By refinement we mean that the refinement is a system where all behaviors are also behaviors of the refined, but not necessarily conversely.

Relation aggregation

This is the form of aggregation where the part objects are just related to the composite object with a special relation, but still just a relation. This was the only form of aggregation supported by OMT.

UML adorns the association with a hollow diamond and calls it *aggregation*.

Relations

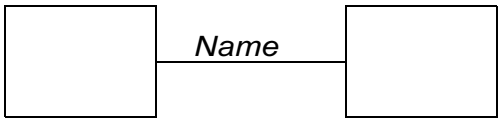
A relation represents *application specific* relationships between objects of the involved classes. Instances of a relation are called links and consist of tuples of object references. Structural “relations” such as subclass-of and part-of are *not* regarded as relations, but as separate constructs.

Relations can be used either as the basis for automatic generation of the corresponding part of functional design (e.g. a database part of the design) - that is as *constructive* parts of the conceptual model, or as *illustrations* of properties that will be “implemented” in some way in the design.

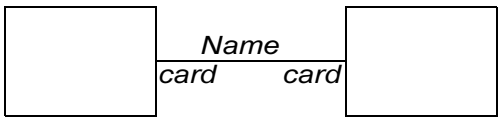
Relations

The basic (underlying/chosen) notation will most certainly have support for relations with classes as endpoints.

With UML as basis we have the following cases of relations:



Relation, maximum and minimum cardinality 1 in both directions.



Relation, maximum and minimum cardinality *card* in both directions.

Cardinality (also called multiplicity) is a text string comprising a comma-separated sequence of integer intervals in the format:

min .. max at least min, at most max

min..* at least min, no upper bound

* any number, zero or more, no upper bound

When defining an object/class by means of aggregation, we may in some case need to express that there are relations to the objects/object sets comprising the aggregation. The same symbols as for relations with classes as endpoints are used, but the semantics is different: the set denoted by the relation is a subset of the related object (set). See also relation aggregation and real aggregation.

Reliability

According to IEEE, reliability for software is defined as follows:

> The probability that the software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and the use of the system as well as a function of the existence of faults in the software.

There exists, however, an alternative version which is also used:

> The ability of a program to perform a required function under stated conditions for a stated period of time.

remote procedures

The remote procedure mechanism consists of four interdependent language constructs:

1. *The exporting of a procedure*. A procedure which is made visible by other processes is marked with the keyword **exported** preceding the procedure heading, e.g. "**exported procedure** Validate ..." from a process within the CentralUnit. The exporting process can control in which states it will accept the remote request. It may also specify to save the request to other states. The controlling of the acceptance is done by using input and save symbols with the remote procedure name preceded by the keyword **procedure**.
2. *The importing of a procedure*. When a process, service or procedure wants to import a remote procedure, it must specify the signature of this procedure in an "imported procedure specification" in a text area. The specification in our case would read: "**imported procedure** Validate; **returns** integer;" where the integer returned would give the result of the validation.
3. *The specification of remote procedure*. In SDL all names must be defined in a specific scope. Thus, the names of remote procedures must be defined in the context in which the actual definition of the procedure and the calls will be contained. In our case the definition of the procedure Validate is within the CentralUnit and the call is in Controller of the AccessPoint. The scope unit enclosing all these is the system itself. There we will find a text area with the following text: "**remote procedure** Validate; **returns** integer;".
4. *The calling of a remote procedure*. The calling of the remote procedure is indistinguishable from local procedure calls unless the caller explicitly states which process it will request the procedure executed by. This can be done by a **to**-clause with a PId following the procedure name of the call.

Remote procedures may be value returning (as in our example above) and they may be virtual. Z.100

Restrictive condition

Four static restrictions are related to conditions in HMSCs:

•If an <msc reference> is immediately preceded by a <condition symbol>, with an associated set of <condition name>s, then this set must be a subset of the set of initial conditions of the <msc ref expression> associated with the <msc reference>.

•If an <msc reference> is immediately followed by a <condition symbol>, with an associated set of <condition name>s, then this set must be a subset of the set of final conditions of the <msc ref expression> associated with the <msc reference>.

•If an <par expr area> is immediately preceded by a <condition symbol>, with an associated set of <condition name>s, then this set must be a subset of the set of initial conditions of the <par expr area>.

•If an <par expr area> is immediately followed by a <condition symbol>, with an associated set of <condition name>s, then this set must be a subset of the set of final conditions of the <par expr area>.

return

A return represents the the completion of a call of a procedure.

A return is interpreted in the following way:

a)All variables created by the interpretation of the procedure start will cease to exist.

b)The interpretation of the procedure-graph is completed and the procedure instance ceases to exist.

c)Hereafter the calling process, service (or procedure) interpretation continues at the node following the call.

Z.100

Revision

is a Version of a component that is derived from an earlier version, and that is designed to replace the earlier version. The difference between two succeeding revision is usually a “small” improvement (error correction or enhancement in functionality). The latest revision is the version one intends should be used (“latest and greatest”).

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.

save

A save specifies that the signals in the save symbol are retained in the input port in the order of their arrival.

The effect of the save is valid only for the state to which the save is attached. In the following state, signal instances that have been “saved” are treated as normal signal instances.

Asterisk save implies that all signals are retained in the input port.

Z.100

scope units

The following kinds of definitions form scope units:

- package
- system type
- system
- block
- block type
- block substructure
- channel substructure
- process
- process type
- service
- service type
- procedure
- signal
- operator, and
- type.

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}

service

A service is a state machine being part of a process instance, and cannot be addressed as a separate objects. It shares the input port and the expressions self, parent, offspring and sender of the process instance.

Only one service at a time is executing a transition. Services alternate based on signals in the input port of the process.

Z.100

service (reference)

A service symbol specificies that a service is part of the containing process (type), and that the definition of the service can be found in a separate service diagram.

Process behaviour by means of services is an alternative to process behaviour by means of a process graph through a set of services. Each service may cover a partial behaviour of the process.

Z.100

service (type) heading

The heading of service diagrams is:

<service-heading> ::= service [<qualifier>] <service-name>

while service type diagrams have the following heading:

<service-type-heading>::=
[<virtuality>]
service type [<qualifier>] <service-type-name>
[<formal-context-parameters>]
[<virtuality-constraint>][<specialisation>]

signal definition

A signal definition defines a set of types of signals. Signal definitions are part of text symbols.

Signals may be defined in system and block diagrams, and these may then be used for communication between the blocks of the system or the processes of the block. Signals may also be defined in process (type) diagrams, but then they can only be used for communication between processes of the same set. Often signal definitions are collected in packages.

Z.100

signal list

Associated with each arrowhead of channels and signal routes or signal lists, that specifies the allowed signals in that direction.

Signallists are defined in text symbols.

Z.100

signal route

A signal route represents a communication path between process sets and between process sets and the environment of the enclosing block/block type.

Z.100

Software node

A software node is a distinct software entity, such as a software process (a concurrent thread), that implements one or more abstract system objects.

A software node will often operate concurrently with other software nodes, but not always.

Software nodes may be aggregated and decomposed, but always so that abstract objects are contained within software nodes.

specialisation

A type may be defined as a specialisation of another type. This is done by the following construct:

<specialisation>::= **inherits** <type-expression> [**adding**]

Specialisation applies to system, block, process, service, data types, and to signals and procedures, and the same semantics apply in all cases:

- All definitions of the supertype are inherited:

- The formal context parameters of a subtype are the unbound, formal context parameters of the supertype definition followed by the formal context parameters added in the <specialisation>.
- The formal parameters of a specialised process type or procedure are the \formal parameters of the process supertype or procedure followed by the \formal parameters added in the <specialisation>.
- The complete valid input signal set of a specialised type is the union of the complete valid input signal set of the<specialisation> and the complete valid input signal set of the supertype.
- A specialised signal definition may add (append) data type identifiers to the \data type list of the supertype.
- A specialised partial type definition may add properties in terms of operators, literals, axioms, operators and default assignment.

• Definitions and transitions (where appropriate) may be added in subtypes.

• Virtual \transitions and types in the supertype may be redefined in the subtype, but for virtual types only to subtypes of their constraint.

A virtual type or procedure is defined by prefixing the keyword of the diagram (e.g. **process** or **procedure**) by one of the keywords **virtual**, **redefined** and **finalized**.

(more)

virtual is used when a type is introduced as a virtual type. A virtual type must be a type defined locally to another type; the implication is that it can be redefined in types that inherit from the enclosing type.**redefined** is used when the redefinition of a virtual type is still virtual. **finalized** is used when the redefinition is not virtual.

Z.100

Specification

A specification covers those aspects of a model that are relevant for its external representation and use. The context part is often sufficient as a specification, but if parts of the content are important it may be included in the specification. Specifications are associated with the abstractions they belong to.

Stake holder

A stake holder is someone or something holding an interest in something.

In TIMe, a stake holder is any person, institution or system with direct or indirect interest in the Domain, a Family or a System instance.

Typical examples are companies, users, operators, owners, and systems in the environment.

start

There is only one start symbol for a process. The transition from the start takes place when the process is generated. A process may be generated either at system start-up or as a result of a create request from another process.

Z.100

state

A state represents a particular condition in which a process may consume a signal resulting in a transition. If the state has neither spontaneous transitions nor continuous signals, and there are no signal instances in the input port, otherwise than those mentioned in a save, then the process waits in the state until a signal instance is received.

Z.100

State

A *state* is a well defined situation which a system or component of a system can *be in*. A state can be defined by a unique name or the values of a set of variables, or through a set of constraints.

A *system state* is normally used for the state of a whole system. A *process state* or *basic state* refers to a state in the finite set of defined, named states in an SDL process (or equivalent). A *complete state* of an SDL process will include values of all local variables and the value of the input port and save queue.

Status

is an Attribute of a Configuration Item that qualifies it, e.g. in terms of formal approval or what quality criteria it fulfils.

As apposed to the Identification of the item, the status will change.

Subject entities

These are entities that are subject to manipulation, representation or control in the Domain. They may be materials in the case of a material transformation domain, e.g. moulding, or they may be entities represented in an information system, e.g. flights and seats, or they may be controlled machinery, e.g. a paper mill.

SubMSC (MSC 92)

An instance of an MSC may be decomposed in form of a sub Message Sequence Chart (sub MSC), thus allowing a top-down specification.

A sub MSC essentially has a structure analogous to an MSC. It is distinguished from the MSC by the keyword **submsc**. Characteristic for a sub MSC is its relation to a decomposed instance containing the keyword decomposed and having the same name as the sub MSC. The relation is provided by the messages connected to the exterior of the sub MSC and the corresponding messages sent and consumed by the decomposed instance.

[In MSC-96 there will be an extension to the decomposition phrase such that any MSC can be specified as the sub MSC.

Synthesis

In TIMe, synthesis is an activity that produces a design from a specification.

Two basic techniques are used to synthesize a design:

1. Transformation. A source description is transformed to a target description according to well defined rules. One example is to generate code from an SDL design.
2. Composition. The content is decomposed into parts (top down) and/or composed from parts (bottom up) using a mixture of manual and automated techniques. TIMe seeks to reuse existing types as much as possible, and to make new types that might be needed reusable. Thus, *design with reuse* and *design for reuse* is part of TIMe. Design with reuse involves:
   - searching for existing types having some desired properties;
   - adapting the properties to fit the particular application.

System

A system is a part of the world that a person or group of persons during some time interval and for some purpose choose to regard as a whole. A system consists of interrelated components, each component being characterised by properties that are selected as being relevant to the purpose.

System family

The System family contains generalised system and component concepts that can be adapted (configurated) and instantiated to fit into a suitable range of user environments. They represent the product base from which a company can make a business out of producing and selling system instances.

The idea is to focus development and maintenance effort mainly on the families in order to:

1. reduce the cost and time needed to produce each particular instance
2. reduce the cost and time needed to maintain and evolve the product base.

In TIMe, system families are formally defined as (collections of) types or classes. Where practical, system types/classes will be defined from which complete system instances may be generated. In addition the system family contains the component types/classes that are used to compose the system types/classes.

System family statement

The system family statement is a concise description of the system family with emphasis on specifications, i.e. the external properties.

System instance

A system instance is a (real) system which can perfom behaviour and provide services.

The system instance area of concern contains system instances produced from system families.

system (type) heading

The heading of system diagrams, that is a system-heading is as follows:

<system-heading> ::= **system** <system-name>

while system type diagrams have system-type-headings:

<system-type-heading>::=
system type [<qualifier>] <system-type-name>
[<formal-context-parameters>]
[<specialisation>]

As indicated in the syntax rule above, a system type can have formal context parameters and it can be a specialisation (of a more general system type).

task

A task may contain a sequence of <assignment statement>s or <informal text>. The <assignment statement>s or <informal text>s are executed in the specified order.

A task is part of a transition.

Z.100

text symbol

Text symbols are used in order to have textual specifications as part of diagrams, especially for specification of signal types, data types and variables.

There is no limit to the number of text symbols that may occur in a diagram. Text symbols are not connected to other symbols by flow lines.

The text symbol is also used for the graphical representation of a use clause, see package.

Z.100

Timeline (instance axis)

No global time axis is assumed for one Message Sequence Chart. Along each instance axis the time is running from top to bottom, however, we do not assume a proper time scale. If no coregion is introduced a total time ordering of events is assumed along each instance axis.

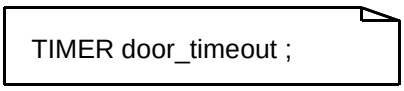
Timer

In MSCs either the setting of a **timer** and a subsequent timeout due to **timer** expiration or the setting of a **timer** and a subsequent timer reset (time supervision) may be specified.

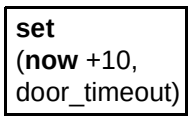
timer

The notion of timers provides a mechanism for specifying time-related matters. Timers are just like alarm clocks. The process waiting for a timer is passively waiting since the process needs not sample them. Timers will issue time-out signals when their time is reached. There may well be several different timers active at the same time. Active timers do not affect the behaviour of the process until the timer signal is consumed by the process.

A timer is declared similarly to a variable.



Timers are **set** and **reset** in tasks. When a timer has not been **set**, it is inactive. When it is **set**, it becomes active.

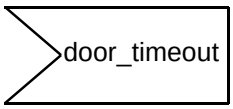


A timer is set with a **time** value. **time** is a special data type and is mainly used in connection with timers. The expression "**now**+10" is a **time** value and it adds the **time** expression **now** and the duration 10 (here: seconds). **now** is an operator of the **time** data type and it returns the current real **time**. Duration is another special data type and it is also mainly used in connection with timers. You may add or subtract duration to **time** and get **time**. You may divide or multiply duration by a real and get duration. You may subtract a **time** value from another **time** value and get duration.

(more...)

The semantics of timers is this: a time value is **set** in a timer and it becomes active. When the time is reached, a signal with the same name as the timer itself will be sent to the process itself. Then the timer becomes inactive.

The timer signal can be input in the same way as ordinary signals:



A timer may be **reset**. It then becomes inactive and no signal will be issued. (If an inactive timer is **reset**, then it remains inactive.) A **reset** will also remove a timer signal instance already in the input port. This happens when the timer has expired, but the time-out signal has not been consumed.

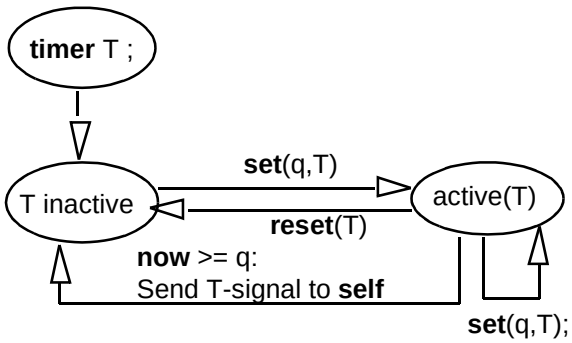
If an active Timer is **set**, the **time** value associated with the timer receives a new value. The timer is still active. If a timer is **set** to a **time** which is already passed, the timer will immediately issue the time-out signal.

There is an operator **active** which has a timer as a parameter and which returns a Boolean that can be used to check whether a certain timer is active or not.

Timer signals may contain data as other signals may contain data. Different parameter values in **set** means generation of several timer instances. **reset** must match these parameter values to eliminate the correct timer instance.

(more...)

The following is a sketch of a finite state machine of the behaviour of a timer.



Z.100

The Jante Law

The Jante Law according to Aksel Sandemose:

1. Du skal ikke tro at du **er** noe. Thou shalt not presume that thou **art** anyone [of notice].
2. Du skal ikke tro at du er like så meget som **oss**. Thou shalt not presume that thou art as good as **us**.
3. Du skal ikke tro at du er klokere en **oss**. Thou shalt not presume that thou art any wiser than **us**.
4. Du skal ikke innbille deg du er bedre enn **oss**. Thou shalt never indulge in the conceit of imagining that thou art better than **us**.
5. Du skal ikke tro du vet mere enn **oss**. Thou shalt not presume that thou art more knowledgeable than **us**.
6. Du skal ikke tro du er mere enn **oss**. Thou shalt presume that thou art more than **us** [in any way]
7. Du skal ikke tro at **du** duger til noe. Thou shalt not presume that **thou** amount to anything.
8. Du skal ikke le av **oss**. Thou art not entitled to laugh at **us**.
9. Du skal ikke tro at noen bryr seg om **deg**. Thou shalt not presume that anyone cares about **you**.
10. Du skal ikke tro at du kan lære **oss** noe. Thou shalt not suppose that thou can teach **us** anything.

The Jante Law (Janteloven) is from the novel “En flygtning krysser sitt spor” (‘A refugee crosses his tracks’) by the Norwegian/Danish author Aksel Sandemose. The book takes place in an imaginary Danish small town called Jante, based on Sandemose’s hometown Nykøbing Mors. The book is about the ugly sides of Scandinavian smalltown mentality, and the term has come to mean the unspoken rules and jealousy of such communities in general.

Transactions

These are entities representing transactions or events in the dynamic behaviour of the Domain, e.g. the purchase of a car, or a user passing a door.

transition

A transition performs a sequence of actions. During a transition, the data of a process may be manipulated and signals may be output.

Actions may be:

- task,
- output,
- set,
- reset,
- export,
- create request,
- procedure call, or
- remote procedure call

The transition will end with the process entering a:

- next state,
- with a stop,
- with a return or
- with the transfer of control to another transition.

Z.100

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.

Validation

to establish the fitness or worth of a software product for its operational mission (from the Latin *valere*, “to be worth”).

variable definition

Variables can be defined in processes, services and procedures.

Variables of process are created as part of the creation of the process instance.

Variables of services are created when the service is created as part of the creation of the containing process instance.

Local variables of a procedure become parts of the procedure instance when the procedure is called, and they cease to exist when the procedure returns.

Variables will get default initial values if nothing else is specified.

Z.100

Variant

is a Version of a component that is designed to co-exist in parallel with other versions of a component, as an alternative. One variant of a component is seldom “better” than another, but offers different alternative functionality (e.g. for different computer platforms).

Verification

to establish the truth of correspondence between a software product and its specification (from the Latin *veritas*, “truth”).

Version

is a common term for Revisions and Variants.

Is also used to denote an identified product configuration with a defined status, typically indicating a larger change than a new revision (e.g. version 5.0 of FrameMaker).

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

virtual process type

A virtual process type is a process type that can be redefined in a subtype of the enclosing block type.

The virtuality is specified in the process type heading or by <virtuality> in the corresponding process type reference symbol.

A redefinition of the process type must be a subtype of the type identified in the virtuality constraint.

Z.100 (virtual types)

virtuality

The virtuality of a type defines whether the type is virtual (so that it can be redefined in a subtype of the enclosing type), redefined (a redefined type, but still virtual), or finalised, that a redefinition that cannot be further redefined.

<virtuality>::= **virtual** | **redefined** | **finalized**

- **virtual** is used when a type is introduced as a virtual type. A virtual type must be a type defined locally to another type; the implication is that it can be redefined in types that inherit from the enclosing type.
- **redefined** is used when the redefinition of a virtual type is still virtual.
- **finalized** is used when the redefinition is not virtual.

virtuality constraint

A constraint on a virtual type has the form of a virtuality\-constraint:

<virtuality-constraint> ::= **atleast** <identifier>

where <identifier> identifies a type (which is called the constraint type) of the appropriate kind (block, process, service or procedure).

The implication of a constraint is that a redefined or finalized definition of the virtual type must be a type definition that inherits from the constraint type. In case of no constraint specified, the definition of the virtual type itself is the constraint.

virtual (input) transition

A virtual input transition is a special case of a general notion of virtual transition (virtual priority input, virtual start, virtual spontaneous transition). In addition SDL has virtual save.

Redefinition of virtual transitions/saves corresponds closely to redefinition of virtual types.

- A virtual start transition can be redefined to a new start transition.
- A virtual priority input or input transition can be redefined to a new priority input or input transition or to a save.
- A virtual save can be redefined to a priority input, an input transition or a save.
- A virtual spontaneous transition can be redefined to a new spontaneous transition.

Z.100

Walkthrough

By “walkthrough” we mean an activity which will make a group of people responsible in solidarity for a document by their joint scrutiny of the document.