Data, process and behaviour modelling in an integrated specification framework
van Hee, K.M.; Verkoulen, P.A.C.

Published: 01/01/1990

Abbreviations
Computing science notes; Vol. 9021

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Data, Process and Behaviour Modelling in an
Integrated Specification Framework

by

K.M. van Hee        P.A.C. Verkoulen

90/21

May, 1991
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
Editors: prof.dr.M.Rem
        prof.dr.K.M. van Hee
Data, Process and Behaviour Modelling in an Integrated Specification Framework

K.M. van Hee

P.A.C. Verkoulen

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513
NL-5600 MB Eindhoven
The Netherlands
phone: +31-40-472993
fax: +31-40-436685
e-mail: verkoulen@win.tue.nl

Abstract

Petri Nets provide an attractive formalism for modelling complex systems. Especially high-level nets, such as Coloured Petri Nets and Predicate/Transition Nets, have high expressive power since they supply tokens with values that are structured by some kind of type system. However, current Petri Net formalisms do not support a real data-oriented view of systems, such as is provided by modern object oriented data models. Of course, it is possible to simulate features of such data models using the aforementioned type system, but then the systems analyst has to do a lot of extra work. Therefore, we introduce in this paper an object oriented data model and we integrate it with a high-level net model. The data model consists of two levels, so we have called it the L2 model.

The first level has been inspired by the Functional Data Model. It has a partially graphical modelling language. The objects specified by this language are called simplexxes or simplex objects. They have an object identity, a number of attributes with values and some knowledge in the form of object identifiers of other simplexxes. The first level allows for modelling the global state space of a system.

The second level specifies structured sets of simplexxes, called complexes or complex objects. One of these simplexxes is called the root. The root has direct or indirect knowledge of all

---

\footnote{This research has been partially supported by the ESPRIT-II project PROOFS (EP 5342) and by the NWO project ISDF (NF-74).}
contained simplexes in the complex. Finally, a complex has a unique identity, a location, i.e. a place where it resides, and a time stamp denoting the creation- or arrival time of the complex. Each simplex belongs to exactly one complex.

The L2 data model is integrated with a high-level Petri Net model by defining each token in the net to be a complex object. The high-level model we consider is a timed net model where the firing of transitions is independent of the values of the tokens; this only depends upon the presence and the time stamps of the tokens. However, the production of new tokens is governed by the values of the consumed tokens. In contrast to the usual timed nets we have time in the tokens as a separate component. Moreover, the time-stamps are calculated from the transition time and a specified delay in the transition. This integration of data and net models is different from other approaches in literature. E.g. in [Heuser & Peres 90] it is described how an Entity-Relationship model can be extended with facts (transitions which are never enabled) in order to model dynamic constraints.

One of the issues that will be dealt with in the paper is the problem of keeping invariant the unicity of object identities in a Petri Net.

**Keywords:** CASE-tools, complex objects, conceptual modelling, data modelling, process modelling, formal specifications, software engineering.
1 Introduction

One of the basic needs of a systems analyst or designer is a framework to specify systems. The product of this activity is called a specification or a conceptual model. It forms the bridge between the user requirements and the implementation of the system. In principle, such a framework should consist of a mathematical (meta)model of systems, a modelling language and a CASE tool. These components are connected in the following way: the language has to have enough expressive power to describe all relevant aspects of any system under consideration in an abstract way. The semantics of the language are given in terms of the mathematical model in order to avoid ambiguities. Furthermore, the CASE tool guides and supports the systems analyst in specifying the system and analyzing his specification. In principle, it would be sufficient that the modeller is enabled to make abstract specifications of a system. It is not necessary to use a CASE tool for this purpose. However, a CASE tool supports the specification process and it provides facilities for simulation and analysis purposes.

The kind of systems we are interested in are distributed information systems and their environments such as offices, factories and logistic systems. These systems are often very complex. Therefore a modelling language is required that supports hierarchical decomposition of specifications and a graphical representation of (parts of) the specifications, thus supporting the analyst in getting an overview over the system. In classical database theory, you usually make the assumption that all data is accessible because there is one central database. In distributed systems, data is distributed and not available at any moment at any arbitrary location in a system.

Now that we have stated our wishes for a specification framework, we will investigate which frameworks are already available.

There are many frameworks in the information systems area that belong to the class of data models ([Tsichritzis & Lochovsky 82]). They are especially suited for modelling the static structure of a system. Examples are the Entity-Relationship model [Chen 76], the Relational model [Codd 70], the Functional model [Shipman 81] and the Binary model [Abrial 74]. These frameworks all have languages to express the static structure, also called the state space of a system. However, they lack concepts for modelling the dynamics of a system and for the way the system components cooperate (the interaction structure). Object oriented data models like GOOD ([Gyssens et al. 90]) provide in some way a solution for this, but they have not been designed primarily with this objective.

A next group of formalisms consists of the algebraic specification languages such as ACT ONE and ACT TWO [Ehrig & Mahr 85-90] and state-based specification languages like Z [Spivey 88] and VDM [Jones 86]. In these formalisms, algebraic specifications are used to specify algebraic data types in a property oriented way, which means that an internal state is modelled implicitly. Of course a central database system can be considered as an abstract data type, but in practice it is hardly possible to model realistic database systems using
algebraic specifications. The state-oriented specification languages are better candidates and there is a lot of experience in modelling quite data-intensive applications using $Z$ and VDM (cf. [Bjørner et al. 90]). In fact, $Z$ can be considered as a generalisation of the relational algebra (cf. [Van Hee & Van Diepen 90]).

The above mentioned languages are not only able to express the static structure of a system, but also operations transforming states. So there is a possibility to model some dynamic aspects of a system. However, most of these languages do not have a graphical representation (neither for data structures nor for the interaction structure of a system). A disadvantage which is less important is that most of them do not have shorthand notations for frequently occurring constraints like functional dependencies and referential integrity.

Both former groups of specification frameworks lack expressiveness to describe distribution aspects of systems. Nowadays we do not consider centralized information systems as a feasible solution for all needs. When someone is modelling large organisations, it is necessary that he considers communicating autonomous information systems. They differ from the distributed database systems in the sense that distributed databases are logically one system. However, communicating information systems are distributed on a logical level. Therefore we have to consider distribution aspects on the level of the conceptual model already.

There are many formalisms for describing the interaction between system components. We mention CCS [Milner 80], CSP [Hoare 85] and process algebra [Bergstra & Klop 86]. These formalisms have a property oriented approach to describe communication, i.e. they consider the system components as black boxes. For the purpose of specifying protocols they are very useful, but they are not very well-suited for specifying data-intensive applications. The reason for this is that they do not facilitate modelling objects with values with an arbitrarily complex structure.

A next group of formalisms which have facilities for describing dynamic aspects of a system is the group of the Petri Net models ([Reisig 85]). A Petri Net is a bipartite network of transitions and places. Transitions are active components which are connected to each other via places. A transition can communicate with transitions to which it is connected via places by sending and receiving so-called tokens over these places. Petri Nets describe distributed systems in a state based way, hence system components are considered as “white boxes”. The original Petri Net formalism by C.A. Petri does not allow tokens (the passive objects flowing through the system) to have values. However, high level net models such as Coloured Petri Nets [Jensen 91], Predicate/Transition Nets [Genrich 87] and the DES model [Van Hee et al. 89] allow tokens with arbitrarily complex values, such as database states. However, like the aforementioned group of formalisms, they do not provide the same facilities to describe complex data structures as data models do.

Petri Net formalisms do not support dynamic reconfiguration of the communication structure of the system: they have a fixed network topology. Object oriented formalisms such as the Actor model [Agha 86] and object oriented programming languages such as POOL [America & Rutten 89] and Small-Talk do allow dynamic reconfiguration. However, these
formalisms are not very well-suited for conceptual modelling of information systems.

We did not mention Dataflow Diagram techniques such as DFD-Yourdon [Yourdon 89], SADT [Marca & McGowan 88] and ISAC [Lundeberg et al. 81] yet. They are used in practice very frequently although they have a lack of mathematical semantics, causing part of the problems concerning the development of information systems in practice [Lyytinen 87].

So far we have seen several formalisms that are suited to model one of the aspects of a system: data structure, local state transitions or communication between system components. However, there are very few frameworks that integrate two or more of the aforementioned aspects and also have a supporting CASE tool. As an example we mention LOTOS [Brinksma 88], Design/CPN [Huber et al. 89] and ExSpect [Van Hee et al. 89]. LOTOS integrates CSP and algebraic specifications, while Design/CPN and ExSpect integrate Petri Nets, dataflow diagrams and a typed functional language in a consistent way.

We will treat the ExSpect model in more detail in section 4 of this paper.

Although typed functional languages such as ML [Harper 86] have powerful data modelling facilities provided by their type system, they have the same deficiencies as state based specification languages as Z and VDM. That is why we introduce an integration step in this paper: the integration of a data model within a framework that already supports complexly structured data in a primitive way, local state transitions and communication between system components.

The data model we will describe is a two level data model. Therefore, we call it the L2 model. The first level, the so-called simplex level, resembles the binary data model. This is nice for practitioners since they are already used to this kind of models. In fact, these classical data models have already great expressive power. One thing they cannot express though is that a group of entities should be considered as one object. Of course it is possible to introduce a new entity (type) that is connected to all entities that should be aggregated, but there is still no way to express that one considers this aggregate and all its "composing" entities as one object. Therefore we introduce a second level in our data model: the complex level. A complex (object) is a set of simplexes. These simplexes have to be related in some way as we will see in section 3. A complex has the following two properties: a location or place and a time stamp. The location denotes where the complex resides in the system. The time stamp is the availability time of the complex; consider it for the moment as its time of birth.

Please note that a complex object is built from simplex objects and not (recursively) from complexes again. However, this does not mean that aggregation and disaggregation of complex objects is not possible: using the structure of a complex (as will be described in section 3) and the schema which defines the structure of the complex, (dis)aggregation is possible. We will come back to this in section 3.

Our data model deserves in some respect the predicate object oriented as it has been defined in [Atkinson et al. 89], because it supports a kind of complex objects and object identity plays an essential role. However, the objects in our model are passive: they do
not send messages, but they can be messages themselves.

We conclude this introduction with an overview of our paper. In section 2 we consider a very basic formalism for describing systems in general: *transition systems*. We show how this formalism can be used to design a formalism that is more suited for specification purposes. In section 3 we introduce a data model both in an informal and a formal way. We will illustrate our approach by means of an example and we will also say something about inheritance and constraints. In section 4 a process model will be introduced that is based upon the data model from section 3. Again, both informal and formal definitions are given. We define the formal semantics of the process model in terms of the transition systems which will be dealt with in section 2. Of course, both section 3 and 4 contain examples which show how these models can be used when developing (information) systems. In the future, we will work out some large examples in order to show the suitability of our model and, maybe, to find improvements. Section 5 contains the conclusion and some more directions for our future research.
2 Transition Systems

At a very basic level a system can be described by a so-called transition system. Such a transition system is represented by a pair \( (S, L) \), where \( S \) is a finite or countable set of states and \( L \) is a function, the transition law, that assigns to each state \( s \in S \) a set of states \( L(s) \). The meaning of this set \( L(s) \) is that if the system is in state \( s \), the next state of the system must be some element of \( L(s) \). Observe that transition systems can be non-deterministic as we leave it unspecified which element of \( L(s) \) will be the next state of the system, as long as it is some element of \( L(s) \). For this reason transition systems are also known as non-deterministic automata. The behaviour or semantics of a transition system is the set of all finite and infinite sequences of states \( (s_0, s_1, \ldots) \), called traces or paths, such that for all \( i \geq 0 \) it holds that \( s_{i+1} \in L(s_i) \). Here \( s_0 \) is called the initial state of the system.

In the literature, one often considers open transition systems with input and output sets. However, it is possible to restrict oneself to closed transition systems, as we do here. The trick is that we consider the actual system we are modelling (the target system) and its environment within one closed transition system.

Transition systems are often used to describe operational semantics of process models, e.g. in [Verkoulen 89]. We do the same in this paper: the formal semantics of the net model in section 4.2 is described in terms of transition systems. However, transition systems do not provide a suited framework for specifying complex distributed systems.

Our approach is based on the observation that many systems in reality can be seen as networks within which objects are moving. The state of a system is represented by the configuration of all objects in the system. In a state transition some of these objects disappear and some new objects are generated. Of course, such a new object can be a modification of a previously existing one. For instance, the production process in a factory can be considered as a flow of orders, materials and semi-finished products through a network of machines, means of transportation, etc. In each production step one or more of these objects are used to produce new ones.

So far we have only considered objects, but in our paradigm a system is a network of moving objects. So, next we will consider the network aspects. Such a network consists of actors which are connected to each other by locations (also called places, channels or stores) and which communicate with each other by sending objects over these locations. The actors, also known as processors or transitions are the active components in the net. They are able to perform local state transitions in the form of consumption and production of objects. These objects are passive entities. Actors are connected to each other via locations: they may only consume objects from locations and produce objects for locations to which they are connected. We assume that the network topology is static, i.e. the connection relation between actors and locations is fixed. There are also formalisms which allow the topology to change: this is called dynamic reconfiguration. Such a formalism is
the Actor model of [Agha 86]. The dynamics in our framework are close to high level Petri Nets (cf. [Genrich 87, Jensen 91]). This provides our framework with many well-studied analysis techniques. This is one of the reasons why we have adopted a Petri Net-like approach instead of a formalism allowing for a dynamic topology of a system. We think that the freedom that is given by allowing a dynamic topology of a system makes the analysis of systems very difficult, if not impossible.

We assume that complex objects that are consumed in a transition disappear, although they may return as output in the same transition. Moreover, we assume transitions to be instantaneous. All complex objects have a unique identification. When the state of a system changes, complex objects are removed, created and modified. The identities of the produced complexes are determined by but different from the identities of the consumed complexes. This means that no such identity can “survive”. This might seem contradictory to the idea of a never-changing unique identification which an object identity is meant to be. It is possible (and sometimes necessary) to consider one child of a parent as the continuation of its parent. In that case, the old identity and the new one both refer to the same real-world entity, but in different states of the system under consideration. Actually, one could say that they are elements of one and the same equivalence class, viz. the set of all object identities referring to one and the same real-world object, but in different states. This is just a way to deal with the rather philosophical question whether an object can be considered as the old one after a modification.
3 The L₂ Data Model

The main issue of this paper is the integration of process modelling with a data model. Though data modelling with a type system has a great expressive power, there is also a problem: it lacks overview and (graphical) support. It is obvious that (textual) type systems are not very suitable for describing the structure of the state space of large and complex systems properly, as they do not give a good overview of the structure of the state space and they do not give any graphical support. An example of how data modelling can be done using a type system like that of ExSpect or ML can be found in [Verkoulen 90]. Reading that paper will give some idea about the software engineering difficulties data modelling with a (textual) type system causes. Data modelling with a type system versus data modelling using a (suitable) data model can be compared with programming in a third generation language versus programming in a functional or logical language: the same things can be expressed, but it takes much more effort in the first case.

Moreover, we want to incorporate some of the recently developed ideas in object oriented research (e.g. [Atkinson et al. 89, Gyssens et al. 90, Bancilhon 88]). One of the features we want to support is the notion of object identity. This means that each object which will ever exist in the specified system has a unique identification. This is very natural: though properties of an object may change in time, the object as it is does not change. When the specification framework supports object identity, the modeller does not have to bother about this; it is guaranteed by default. Of course, all this can be simulated too in the existing framework, but this places an extra burden upon the modeller.

The same holds for other features which an object oriented specification tool should support according to [Atkinson et al. 89]: complex objects, encapsulation, types or classes and inheritance.

An example of how data modelling can be done using the ExSpect type system can be found in [Verkoulen 90]. This example illustrates the software engineering problems that occur when the only data modelling facility available is a type system.

In [Verkoulen 90] a first attempt is made to combine ExSpect with a conventional data model. However, that was only a kind of feasibility study. The data model was not integrated with the process model in that paper, but it was put “on top” of ExSpect. In this paper, we will describe a complete integration.

Therefore, we will introduce a new data model. It has been inspired by the Functional Data Model ([Shipman 81, Aerts & Van Hee 88]) and the Binary Model ([Abrial 74]), but it has also some object oriented features. Later in this paper, the data model and the process model will be combined into one framework for system specification.
3.1 Informal Description of the L2 Data Model

When we are modelling a system, one of the aspects we have to describe is its state space, i.e. the set of all possible states the system may “reach”. Such a state is characterized by (the properties of) the objects the system contains. Some of these objects are concrete such as materials and patients, others are abstract such as orders, treatments and data. All objects belong to an (object) class, which is a set of objects with the same characteristics. Here, we distinguish two kinds of object classes: simplex classes and complex classes, of which the “inhabitants” are called simplexes and complexes respectively. Complexes are built of simplexes and so there are two levels. All tokens in the net model will become complex objects. As we will see below, a complex consists of one or more simplexes.

A simplex represents a concrete or abstract entity in the real world. The properties of a simplex can be divided into three parts: an administrative part, a reference part and a value part.

The administrative part records the identity of the object and the simplex class to which it belongs.

The reference part registers the relationships of a simplex with other simplexes. Each of these relationships has a name (a label): the reference name. The reference part of a simplex contains for each reference name a set of object identities. References can be considered to represent the knowledge an entity has about other entities. Notice that this does not only represent the fact that entities have some knowledge about each other, but also which entity “possesses this knowledge”. We will come back to this aspect below.

Finally, the value part is a tuple which has an arbitrarily structured value for each attribute of the class the object belongs to.

In a schema the classes to which the simplexes in a system belong, and the relationships between those classes are described. The set of all simplex classes Simplex is partitioned into two parts: Simplex* ⊆ Simplex is the set of simplex classes of which the simplex objects in a state have to have a unique identity. These “unique” simplex classes will often be used for modelling physical entities, which will be unique in a system. The other ones are suited for modelling e.g. information objects, which can occur more than once in a system.

We define all this in a schema using a labeled directed graph. The nodes of the graph are the simplex classes (the unique ones are labeled with an asterisk) and the attributes. Relationships between classes are represented by labeled directed arcs. An arc from a class D to a class R denotes that to each object in D belongs a set of objects in R. So the functions are multi-valued, as in [Shipman 81]. Finally, attributes are connected to the class they belong to by an unlabeled and undirected edge.

In our schemas, we will denote simplex classes by boxes and attributes by ovals.

An instance of a class is a set of objects belonging to that class. We will provide facilities for formulating constraints which allow only certain instances of a class. Typical constraints for the library case we will see in example 3.1 are: the amount of books lent by a member
may not exceed the number of 10 at any moment in time. Also the due date (ddate) of a lending must fall after its lending date (ldate). We will come back to this in section 3.4.

Now consider figure 1 which shows a simplex schema represented graphically. This schema represents the state space structure of a simple library system. The library possesses books, which have a title. Two books may have the same title. Furthermore, the members of the library are registered. They may lend books and they may reserve titles (when making a reservation, they are only interested in the title, not in one specific copy).

Example 3.1

![Diagram of a library simplex schema]

Figure 1: Library simplex schema.

We will now say something about complexes. In conventional database theory, what we call a complex object is often seen as a view on the database. But when systems are
to be modelled where the flow of entities and distribution aspects are important, complex objects play another role. Examples of such applications are distributed databases, message handling systems, logistic systems, etc. In this kind of applications it is necessary to be able to describe that some piece of information is (only) available during a certain period of time at a certain spot in the system.

For example, we want to be able to treat a reservation, the book which is reserved and the person reserving it as one (complex) object.

A complex class consists of a set of simplex classes; one specific simplex class in this set is called the root class or simply the root. We require that each complex class is connected, i.e. every simplex class in a complex class can be reached from the root via an undirected path of references to simplex classes that belong to the complex class. Complex classes may overlap. Complex classes may have names that are also names of simplex classes.

We will illustrate the definition of complex classes in figure 2 using the library case from example 3.1. Here, we do not indicate attributes and reference labels because the picture would become too crowdy then. The complex classes are represented by dashed curves around their composing simplex classes and a dashed line leading from this curve to the root.

Example 3.2
We assume that the library consists of two filials and one central department. Each filial has its own member administration (though these administrations do not have to be disjoint). They may also have titles in common. However, books, reservations and lendings are unique in the library system. These conventions are denoted by the asterisks in the schema.

The complex schema below consists of three "singleton complex classes" containing only one simplex class, and two non-atomic complex classes: request and lending. Complex objects of classes lending and book can not intersect. However, complexes of classes member and request resp. title and request can.

The elements of a complex class are called complex objects. The properties of some complex object c consist of three parts: an administrative part, a root simplex and a set of contained simplexes.

The administrative part contains the unique identity of the complex, the location where the complex resides, the availability time of the object at that location and the complex class to which the complex belongs.

The third constituent part of a complex is a set of contained simplexes. We require that the root simplex of a complex is an element of this set of contained simplexes and that the contained simplexes are connected to the root via references.

1These graphical conventions are preliminary. We are currently developing the graphical language for complex classes.
Figure 2: Complex classes.
We will now extend our notion of a schema in order to account also for complex classes: a schema defines both simplex and complex classes. The simplex classes are defined as described before. The complex classes are defined by assigning a root simplex and a set of contained simplex classes to each complex class. As we will see below, we impose some constraints upon a schema. One of them is the constraint that each simplex class has to be part of at least one complex class.

A complex object must have a structure in accordance with the structure of the complex class it belongs to. This means that a complex object can only contain simplex objects that belong to the contained simplex classes of the complex class of the object. At each moment, a system is in a certain state (which is a set of complexes satisfying some constraints). In such a state, all simplices must be part of some complex.

In our paradigm, the objects that move through a network will be complex objects. So only complexes can exist on their own, simplices only exist within complexes. Simplices inherit the location and the time stamp from the complex they are part of.

The simplex level is enough to model state spaces of systems, the complex level is important to describe the transition laws of complex dynamic systems. Now it is clear why we call this approach a two level structure: there are two levels, viz. the simplex and the complex level. A complex is built from simplices, and not (recursively) from complexes. The structure of a complex is represented by the structure of the references between the simplices that are contained in the complex. This has several advantages; among others it avoids recursive types with fix-point semantics and (therefore) it is also possible to have objects which are represented by an arbitrary graph, not only by trees.

Despite this definition of complexes, hierarchical decomposition of complexes is possible. We will provide operations for this in the object manipulation language we are developing at the moment. We will illustrate this in the next example.

**Example 3.3**

In figure 3 we see a schema and a complex object according to that schema. The schema contains two complex classes `ctrain` and `cbag`. The complex object `c1` is an element of `ctrain`. It contains six simplex objects: `t1` of class `train`, `b1` and `b2` of class `bag` and `l1`, `l2` and `l3` of type letter.

Our object manipulation language will contain an operation for inducing the two dashed complex objects of type `cbag` with root simplices `b1` and `b2` respectively from the `c1` complex. So to the outside, it looks like the `c1` complex contains two subcomplexes, instead of six simplices.
Another reason why we call this approach a two-level structure is that there are two stages in the state space modelling. First, the simplex classes, references and attributes are defined. Then, the complex classes are defined by clustering simplex classes.

3.2 Formalisation of the $L_2$ data model

We distinguish schemas and instances. A schema describes the structure of objects belonging to classes in the schema; an instance of a class is a set of objects belonging to that class. A set of instances of all classes in a schema is called a state. All simplex objects in a state have to have different identities. Complex objects inherit their identity from their simplex root, so they have a unique identification too. Therefore, we will start by formalising the assignment of object identities in section 3.2.1. Afterwards, we will give a formal definition of a schema and an instance of the $L_2$ data model.

3.2.1 Identification Structure

Part of an $L_2$ schema is a so-called identification structure. Such an identification structure consists of a countable set $I$ (an identification set), a set $A \subseteq I$ (the atomic or initial identities) and the functions $\mathcal{F}$ and $\mathcal{G}$ which are defined upon $I$. The $\mathcal{F}$ function assigns to each identity $i$ a set $\mathcal{F}(i) \in \mathcal{P}(I)$ of identities which are "children" of $i$. For different $i$ and $j$, the sets of children $\mathcal{F}(i)$ and $\mathcal{F}(j)$ must be disjoint. This implies that no identity can "survive": an identity $i$ cannot be an element of $\mathcal{F}(i)$ as this would violate the aforementioned disjointness constraint (cf. lemma 3.2). This might seem contradictory to the idea...
of a never-changing unique identification. We solve this as will be described below. It is possible (and sometimes necessary) to consider one child of a parent as the continuation of its parent. In an identification structure, this is done by defining the continuation function $\mathcal{G}$. In that case, $i$ and $\mathcal{G}(i)$ refer to the same real-world entity, but in different states of the system under consideration. Actually, one could say that $i$ and $\mathcal{G}(i)$ are elements of one and the same equivalence class, viz. the set of all object identities referring to one and the same real-world object, but at different points in time. That is why in an auxiliary definition we will introduce the predicates $\equiv^\mathcal{G}$ (equality modulo $\mathcal{G}$) and $\in^\mathcal{G}$ (membership modulo $\mathcal{G}$). Intuitively, two identities are equal modulo $\mathcal{G}$ if one of them is the transitive $\mathcal{G}$-continuation of the other.

**Definition 3.1 Identification Structure**

An identification structure is a tuple $(I, \mathcal{F}, \mathcal{G}, A)$ where $I$ is a countable set called the identification set, $\mathcal{F}$ is called the child function, $\mathcal{G}$ is called the continuation function and $A \subseteq I$ is called the initial set. It has to satisfy the following conditions:

1. $\mathcal{F} \in I \rightarrow \mathcal{P}(I \setminus A)$
2. $(\forall i, j \in I :: i \neq j \Rightarrow \mathcal{F}(i) \cap \mathcal{F}(j) = \emptyset)$
3. $(\forall i \in I :: (\exists j \in A :: (\exists n \in \mathbb{N} :: i \in \mathcal{F}^n(\{j\})))$\footnote{We define $\mathcal{F}^n$ for $n \in \mathbb{N}$ as follows:
For all $n \in \mathbb{N}$, $\mathcal{F}^n \in \mathcal{P}(I) \rightarrow \mathcal{P}(I)$ is such that

$(\forall B \in \mathcal{P}(I) :: \mathcal{F}^n(B) = B \land (\forall k \in \mathbb{N} :: \mathcal{F}^{k+1}(B) = \{j \in I \mid (\exists i \in \mathcal{F}^k(B) :: j \in \mathcal{F}(i))\}$)$. We will write $\mathcal{F}^n(x)$ instead of $\mathcal{F}^n(\{x\})$.

4. $\mathcal{G} \in I \rightarrow I$ such that $(\forall i \in \text{dom}(\mathcal{G}) :: \mathcal{G}(i) \in \mathcal{F}(i))$\footnote{An injective function $f$ is generalized for sets as follows: $f(\{x_1, \ldots, x_n\}) = \{f(x_1), \ldots, f(x_n)\}$.

$\mathcal{F}(i)$ and $\mathcal{F}(j)$ are disjoint. The second one says that an identity can never be an $\mathcal{F}$-descendant of itself.

In order to be able to prove lemma 4.1 in section 4.2, we need two auxiliary lemmata. The first one is in fact a reformulation of the requirement that for two different identities $i$ and $j$ the sets of their $\mathcal{F}$-descendants ($\mathcal{F}(i)$ and $\mathcal{F}(j)$) are disjoint. The second one says that an identity can never be an $\mathcal{F}$-descendant of itself.

**Lemma 3.1**

Given an identification structure $ID$ with identification set $I$ as in definition 3.1, it holds that

$(\forall i, j \in I :: (\forall X \subseteq I :: (j \in \mathcal{F}(i) \land j \in \mathcal{F}(X)) \Rightarrow (i \in X \land \mathcal{F}(i) \subseteq \mathcal{F}(X)))$.
Proof

Let \( i, j \in I \) and \( X \subseteq I \) such that \( j \in \mathcal{F}(i) \) and \( j \in \mathcal{F}(X) \). We prove \( i \in X \) by induction to \( X \).

If \( X = \{x\} \), then \( j \in \mathcal{F}(i) \land j \in \mathcal{F}(X) \) implies \( x = i \) because of clause 2 of definition 3.1, giving \( i \in X \).

If \( X = \{x\} \cup Y \), then there are two possibilities:
1. If \( x = i \), then \( i \in X \) holds.
2. If \( x \neq i \), then \( j \notin \mathcal{F}(x) \), because this would contradict clause 2 of definition 3.1. So in this case, \( j \in \mathcal{F}(Y) \) holds. But then (using the induction hypothesis) \( i \in Y \) holds, so \( i \in X \) holds too.

Now that we have proved \( i \in X \), it is trivial that \( \mathcal{F}(i) \subseteq \mathcal{F}(X) \) as \( \mathcal{F}(\bigcup_{x \in X} x) = (\bigcup_{x \in X} \mathcal{F}(x)) \).
\( \square \)

Lemma 3.2

Given an identification structure \( ID \) with identification set \( I \) as in definition 3.1, it holds that
\[
(\forall i \in I : (\forall n \in \mathbb{N} : n > 0 : i \notin \mathcal{F}^n(i))).
\]

Proof

Because of definition 3.1, we know that for all \( i \in I \) it holds that \( i \in \mathcal{F}^m(a) \) for some \( a \in A \) and \( m \in \mathbb{N} \).

Let \( a \in A \). We will prove by induction to \( m \) that for each \( i \in \mathcal{F}^m(a) \) the assumption \( i \in \mathcal{F}^n(i) \) for some \( n > 0 \) leads to a contradiction.

Let \( i \in \mathcal{F}^m(a) \) and suppose \( i \in \mathcal{F}^n(i) \) for some \( n > 0 \). There are two cases:
1. \( m = 0 \), which means \( i = a \). But then \( a = i \in \mathcal{F}^n(i) \) for some \( n > 0 \) gives that there is some \( Y \in \mathcal{P}(I) \) with \( \{a\} \cup Y \in \text{rng}((\mathcal{F})) \), which is contradictory to clause 1 of definition 3.1.
2. \( m > 0 \), so some \( k \in \mathcal{F}^{m-1}(a) \) and \( j \in \mathcal{F}^{n-1}(i) \) exist with \( i \in \mathcal{F}(k) \land i \in \mathcal{F}(j) \). There are again two cases:
   1. \( j \neq k \), but this contradicts clause 2 of definition 3.1.
   2. \( j = k \), but this means \( k \in \mathcal{F}(k) \) as \( k \in \mathcal{F}^{n-1}(i) \land i \in \mathcal{F}(k) \), using the transitivity of \( \mathcal{F} \). But as \( k \in \mathcal{F}^{m-1}(a) \) this gives a contradiction using the induction hypothesis.
\( \square \)

In the definition below we will introduce the notion of “equality modulo \( \mathcal{G} \)”. This is necessary because of the following: we will see that no complex object identity can “survive” a state transition (cf. lemma 3.2). We have introduced the \( \mathcal{G} \) function to account for this. However, this would mean that we would have to update all references to an object when the identity \( i \) of that object would become \( \mathcal{G}(i) \). This would mean a global check of all objects in the system. This would destroy our local state transition paradigm. Therefore, we define the notion of equality modulo \( \mathcal{G} \). Intuitively, this means that the object identities \( i \) and \( \mathcal{G}(i) \) can be considered the same.
Definition 3.2 \( \equiv \) (equality modulo \( G \))
Let \( ID \) be an identification structure with identification set \( I \) and continuation function \( G \). Then the predicates \( \equiv \) and \( \not\equiv \) are defined as follows:
1. For \( i, j \in I \), we define \( (i \equiv j) \overset{\text{def}}{=} (\exists n \in \mathbb{N} :: i = G^n(j) \lor j = G^n(i)) \);
2. For \( i \in I \) and \( X \subseteq I \), we define \( i \not\equiv X \overset{\text{def}}{=} (\exists j \in X :: i \not\equiv j) \).
\( \square \)

We conclude this section with an example of an identification structure. We will prove that this construction satisfies the constraints of definition 3.1.

Example 3.4 Construction Identification Structure
We propose the following construction of an identification structure \( ID \).
- \( I = \mathbb{N}^+ \), the finite (non-empty) sequences of natural numbers.
- \( A = \mathbb{N} \).
- For \( i \in I \), \( F(i) \subseteq \{j \cdot i \mid j \in \mathbb{N}\} \), so new identities are created from old ones by adding a natural number in front of the old identity.
- For \( i \in I \), it holds that \( G(i) = 0 \cdot i \).
- For \( i, j \in I \) it holds that \( i \equiv j \) if and only if the following predicate is satisfied:
  \( (\exists n \in \mathbb{N} :: i = 0^n \cdot j \lor 0^n \cdot i = j) \).
  Informally, this means that all identities which are equal after deleting (some) leading zeros refer to the same real-world entity.

We will now prove that this construction satisfies the constraints from definition 3.1:
1. It is obvious that \( \text{rng}(F) \subseteq P(I) \). Furthermore, it is obvious that there are no sets in \( \text{rng}(F) \) which contain elements of \( A \), for all elements of sets in \( \text{rng}(F) \) have at least length 2 by the construction of \( F \) whereas the elements of \( A \) are all of length 1.
2. Let \( i, j \in I \) with \( i \neq j \). Let furthermore \( x_i \in F(i) \land x_j \in F(j) \). Then \( x_i = \alpha \cdot i \) and \( x_j = \beta \cdot j \), for some \( \alpha, \beta \in \mathbb{N} \). But then \( x_i \neq x_j \), because of the definition of equality of lists and the assumption \( i \neq j \).
3. It is obvious that all non-empty finite sequences of natural numbers can be made by starting with the last element of the sequence and concatenating numbers in front of it until the sequence has been constructed.
4. Let \( i \in I \). Then \( G(i) = 0 \cdot i \in \{j \cdot i \mid j \in \mathbb{N}\} = F(i) \).
\( \square \)

3.2.2 Formalisation of L2 Schemas and Instances

We now give a formal definition of a schema in our data model. We have added some informal comments to the formal definitions in order to make them more easy to understand.
Definition 3.3 Schema

A schema is a tuple \((CL, RF, AT, SC, DM)\) where:

- \(CL = \langle \text{Simplex}, \text{Complex}, \text{Simplex} \rangle\); Simplex and Complex are sets denoting the names of the simplex and complex classes. Furthermore, \(\text{Simplex}^* \subseteq \text{Simplex}\) is the set of simplex classes of which the simplex objects in a state have to have a unique identity.
- \(RF = \langle RN, RS \rangle\): the reference structure; Here \(RN\) is the set of reference names and \(RS \in \text{Simplex} \rightarrow (RN \leftarrow \text{Simplex})\) assigns to each simplex class and each suitable reference name a simplex class.
- \(AT = \langle AN, AS, D \rangle\): the attribute structure; Here \(AN\) is the set of attribute names and \(AS \in \text{Simplex} \rightarrow \mathcal{P}(D)\) is a function assigning to each simplex class and each suitable attribute name the set of all possible attribute values, which is a subset of the attribute domain set \(D\). For \(s \in \text{Simplex}\) and \(a \in \text{dom}(AS.s)\), we call \(AS.s.a\) the attribute domain of attribute \(a\) of class \(s\).
- \(SC = \langle RT, CS \rangle\) defines the structure of the complex classes:
  - \(RT \in \text{Complex} \rightarrow \text{Simplex}\) is called the root function. It assigns to each complex class the simplex class that is the root of the complex.
  - \(CS \in \text{Complex} \rightarrow \mathcal{P}(\text{Simplex})\) assigns to each complex class a set of contained simplex classes, such that
    \[
    \forall c \in \text{Complex} :: RT.c \in CS.c \land \\
    \forall s \in CS.c :: (\exists n \in \mathbb{N} :: (\exists s_0, \ldots, s_n \in CS.c :: s_0 = RT.c \land s_n = s \land \\
    (\forall i : 0 \leq i < n : s_{i+1} \in \text{rng}(RS.s_i))))
    \]
    This means that for any simplex class \(s\) that is part of a complex class \(c\) there must be a path of references within \(c\) from the root of \(c\) to \(s\).
    It should hold that \((\cup_{c \in \text{Complex}} CS.c) = \text{Simplex}\) (no "dangling" simplex classes).
- \(DM = \langle ID, LN, TP \rangle\) defines the domains:
  - \(ID\) is an identification structure;
  - \(LN\) is the set of locations where (complex) objects may reside;
  - \(TP\) is the set of time points; it is a totally ordered set with an addition operator \(+\).

\[
\Box
\]

The structure of the simplex and complex objects within an instance is described by a so-called object universe.

Definition 3.4 Object Universe

An object universe \(U\) is a tuple \((S, C, \Sigma)\). Here \(\Sigma\) is a schema as in the definition above. Furthermore, \(S\) is the set of all possible simplex objects according to \(\Sigma\). Such a simplex object \(s\) is of the form \((id, sim, ref, val)\), where

---
5We consider tuples to be entities upon which component functions \(c_1, \ldots, c_n\) are defined. We say \("x is a tuple \((c_1, \ldots, c_n)\)" when there can be no misunderstanding, but we should have denoted this as \((c_1(x), \ldots, c_n(x))\) actually.

19
• \( id \in I \), the identity of the simplex object \( s \);
• \( sim \in \text{Simplex} \), the simplex class to which \( s \) belongs;
• \( ref \in (RN \leftarrow \mathcal{P}(I)) \), a function assigning to each suitable reference \( r \) a set of object identities of simplexes which are related to \( s \) "via" \( r \);
• \( val \in (AN \leftarrow D) \), a function assigning to each attribute name an attribute value.

Finally, \( C \) is the set of all possible complex objects according to \( \Sigma \). Such a complex object \( c \) is of the form \((id, com, loc, at, root, cont)\), where

• \( id \in I \), the identity of \( c \);
• \( com \in \text{Complex} \), the complex class to which \( c \) belongs;
• \( loc \in LN \), the location of \( c \);
• \( at \in TP \), the availability time of \( c \);
• \( root \in I \), the identity of the root simplex of \( c \);
• \( cont \in \mathcal{P}(S) \), the set of contained simplexes of \( c \).

We define the auxiliary predicate \( \text{path}(c,s_1,s_2) \) in the context of an object universe. This predicate holds iff there is a directed path of references from simplex \( s_1 \) to simplex \( s_2 \) in some complex \( c \in C \).

**Definition 3.5 path**

Suppose an object universe \((S,C,\Sigma)\) is given and let \( c \in C \) and \( s_1, s_2 \in \text{cont}(c) \). Then the predicate \( \text{path}(c,s_1,s_2) \) is defined by

\[
\text{path}(c,s_1,s_2) \equiv \\
(\exists n \in \mathbb{N} : (\exists x_0, \ldots, x_n \in \text{cont}(c) : x_0 = s_1 \land x_n = s_2 \land \\
(\forall k \in \mathbb{N} : 0 \leq k < n : \text{id}(x_{k+1}) \in \text{rng}(\text{ref}(x_k))))))
\]

An object universe satisfying some conditions regarding object identities and the structure of references, attributes and complex objects will be called valid. The exact conditions will be given in the following definition.

**Definition 3.6 Valid Object Universe**

An object universe as described in definition 3.4 is called valid iff the following constraints are satisfied:

1. \((\forall s \in S : (\forall r \in \text{dom}(\text{ref}(s)) : r \in \text{dom}(RS.sim(s))))\)
   which means that \( s \) has only references of the simplex class \( s \) belongs to;
2. \((\forall s \in S : (\forall a \in \text{dom}(\text{val}(s)) : a \in \text{dom}(AS.sim(s)) \land \text{val}(s).a \in AS.sim(s).a))\),
   which means that each simplex has an associated tuple of attribute values that fits into the attribute structure of its simplex class;
3. \((\forall c \in C : \text{sim}(\text{id}^{-1}(\text{root}(c))) = RT.com(c) \land (\forall x \in \text{cont}(c) : \text{sim}(x) \in CS.com(c)))\).
   This means that the root class of a complex object \( c \) has to be equal to the root class
of the complex class of \( c \) in the schema, and that the simplex classes of all contained simplexes have to be an element of the contained classes of the class of \( c \);

4. \( (\forall c \in C :: (\exists r \in \text{cont}(c) :: \text{id}(r) = \text{root}(c) \wedge (\forall x \in \text{cont}(c) :: \text{path}(c, r, x)))) \).

This means that the root simplex \( r \) of a complex \( c \) is an element of the contained simplexes of \( c \) and that the contained simplexes are connected to the root simplex through zero or more references within the complex.

\[ \square \]

From now on, we will only consider valid object universes, unless explicitly stated otherwise. In the next definition, the state space with respect to a (valid) object universe will be defined.

**Definition 3.7 State Space**

We define the state space \( SS_U \) with respect to a valid object universe \( U \) to be a set-valued set, of which the elements are sets of complexes which satisfy some additional constraints; the complexes have to have different identities; the same holds for simplexes from “unique” simplex classes (Simplex’). The simplexes in these complexes have to satisfy the reference structure of the schema of \( U \):

\[
SS_U = \{ \sigma \in \mathcal{P}(C) | (\forall c_1, c_2 \in \sigma :: c_1 \neq c_2 \Rightarrow id(c_1) \neq id(c_2)) \wedge \\
(\forall s_1 \in c_1, s_2 \in c_2 :: \text{sim}(s_1) = \text{sim}(s_2) \in \text{Simplex}^* \wedge s_1 \neq s_2 : id(s_1) \neq id(s_2)) \wedge \\
(\forall s \in \text{cont}(c) :: (\forall r \in \text{dom}(\text{ref}(s)) :: (\forall x \in \text{cont}(c) :: \\
id(x) \in \text{ref}.s.r \Rightarrow \text{sim}(x) = RS.\text{sim}(s).r))))} \}
\]

\[ \square \]

### 3.3 Inheritance

In this section, we will say something about *inheritance*. For the moment, we have a very pragmatic view on the notion of inheritance. Inheritance is not another primitive in the data model, but it is a property that can be derived from a schema. We do not record it directly in the formal representation of the schema. However, we will make it possible to use the concept of inheritance to obtain a more compact specification. So it can be used on the syntactic level of the specification language to facilitate the definition of a schema.

Informally, a simplex class \( s_1 \) may inherit attributes and/or references from another simplex class \( s_2 \). In addition to this, \( s_1 \) may have attributes and references of its own. So \( s_1 \) has the same or more attributes and for each reference from \( s_2 \) there is a reference from \( s_1 \) to the same simplex class or to a simplex class that inherits from a simplex class connected to \( s_2 \). In this case \( s_1 \) is called a *specialisation* of \( s_2 \) and \( s_2 \) is called a *generalisation* of \( s_1 \). Similarly, a complex class may inherit from another complex class.

As a class can be a specialisation of two or more other classes, *multiple inheritance* is allowed.
Definition 3.8
Let $\Sigma$ be a schema. For all simplex classes $s_1, s_2 \in \text{Simplex}$ it holds that $s_1$ inherits from $s_2$ (notation $s_1 \prec s_2$) iff the following constraints hold:
1. $\text{dom}(RS.s_2) \subseteq \text{dom}(RS.s_1)$
2. $\text{dom}(AS.s_2) \subseteq \text{dom}(AS.s_1)$
3. $(\forall r \in \text{dom}(RS.s_2) :: (RS.s_2.r = RS.s_1.r) \lor (RS.s_1.r \prec RS.s_2.r))$
4. $(\forall a \in \text{dom}(AS.s_2) :: AS.s_2.a \subseteq AS.s_1.a)$

For all complexes $c_1, c_2 \in \text{Complex}$ it holds that $c_1$ inherits from $c_2$ (notation $c_1 \preceq c_2$) iff the following constraints are satisfied
1. $RT.c_1 \prec RT.c_2$
2. $(\forall s \in CS.c_2 :: (\exists s' \in CS.c_1 :: s' \prec s))$

$\Box$

3.4 Constraints

In realistic practical cases, the state space of a system is very large. Usually, you want to limit the set of all possible states by defining constraints. Constraints can be defined using predicates.

Definition 3.9
Let $SS_U$ be the set of all possible states based on some valid object universe $U$. Then a function $\varphi \in SS_U \rightarrow \{\text{true, false}\}$ is called a constraint.

For any set of constraints $\mathcal{X} \subseteq (SS_U \rightarrow \{\text{true, false}\})$, the state space with respect to $\mathcal{X}$ is denoted by

$S_{\mathcal{X}} = \{s \in SS_U | (\forall \varphi \in \mathcal{X} :: \varphi(s))\}$

$\Box$

We have classified the constraints into six types:
1. Attribute constraints, only involving one attribute of a simplex class;
2. Tuple constraints, involving two or more attributes of one simplex class;
3. Reference constraints, involving only references. We distinguish
   3.1 Cardinality constraints;
   3.2 Key constraints;
   3.3 Exclusiveness constraints;
   3.4 Global reference constraints: all other reference constraints.
4. Simplex constraints, involving only one simplex class;
5. Complex constraints, involving only one complex class;

We are still working on how to express constraints graphically in a schema. We expect that
we will not be able to express all constraints graphically. However, cardinality constraints can be represented graphically. We will give an example of that in figure 4.

![Figure 4: Cardinality Constraints.](image)

In figure 4, $r$ is the name of the reference and $k$, $l$, $p$ and $q$ are natural numbers, including the infinity symbol $\infty$. The arrow heads on the arcs denote which simplex "knows" and which simplex "is known". So, $r$ is a reference of $A$, which means that an $A$ simplex knows $B$'s while $B$'s have no knowledge of the $A$'s that refer to them.

The intervals in figure 4 represent a cardinality constraint: figure 4 means that to each $A$-simplex a a set $\{b_1, \ldots, b_n\}$ of $B$-simplexes which are "$r$-known" by a belongs where the cardinality $n$ of this set is at least $k$ and at most $l$. In case $l = \infty$ there is no limit to the number of references, but is should be finite. Analogously the numbers $p$ and $q$ determine bounds to the number of $A$ simplexes that may refer to one and the same $B$-simplex.

We will introduce symbols at the ends of the arcs as shorthands for these labels.
4 The OONET Model

In this section, we will start with an informal treatment of the DES process model and the proposed extensions to that model, which will result in a new model: the OONET model. Afterwards, the OONET model will be formalized. In this formalisation, the process model is integrated with the L2 data model from section 3.

4.1 Informal Description of the OONET Model

A network in the process model consists of two kinds of components, viz. processors and channels, corresponding to transitions and places in Petri Net terminology. The processors are active components which can communicate with each other as they are interconnected by channels. They communicate by sending tokens to each other over these channels. Tokens are passive objects. So, each channel contains at each moment zero or more tokens which must "fit into" the type of that channel. Moreover, each token has a time stamp, denoting the (availability-) time before which it can not be 'consumed'. The state of a system is represented by the bag of all tokens present in the system.

Each processor has one or more input channels and zero or more output channels. An assignment of tokens to channels such that each input channel of exactly one processor has exactly the number of consumable tokens as prescribed by the specification is called an event. This processor can then execute (fire). Processors are eager to start, i.e. they will fire as soon as possible. The event time at which an event e may occur is the maximum of the time stamps of the tokens in e. The transition time of a system in a certain state is the minimum of the event times of all possible events. Being in a certain state, an event will be selected of which the event time is equal to the transition time of the system. This event will then be executed. This causes a state transition: the consumed tokens are removed from the state and the generated tokens are added to it. This all happens instantaneously. The time stamps of the produced tokens are equal to the sum of the transition time and a specified delay. This delay is non-negative, so the transition times of successive events will be ascending.

Processors may fire at the same moment (concurrently). However, when giving the formal semantics we will assume that only one processor is involved in an event; we work with interleaving semantics: if two processors fire concurrently, some non-deterministic ordering is made. In [Van Hee et al. 89] it is proved that the set of all reachable states in this definition is equal to the set of reachable states which is obtained when you do not work with interleaving semantics. The difference is that in the former case some states can only be reached via another one.

The ExSpect language allows for defining processors, channels, stores and systems. A store is a special kind of channel which contains always exactly one token. Whenever this token is 'consumed' by any processor, it is replaced by another token without delay,
either explicitly by the processor or implicitly by putting the old one back. A system is an aggregate of processors, channels, stores and systems. A system may contain subsystems, which makes it possible to structure a specification hierarchically.

Stores and systems are not part of the formal model; they are considered as sugar in the language. It is possible to give a formal semantics of the dynamic part of ExSpect without introducing stores and systems in the process model, as has been shown in [Verkoulen 89].

So far this is almost the same as the DES model in [Van Hee et al. 89]. We will now describe the aspects in which the OONET model differs from the DES model. It is possible to translate an instance of the net model into an instance of the OONET model and vice versa.

In the OONET model, tokens will be complex objects and so they contain much more structure than they used to in the DES model and in CPN. This is necessary for the integration of data modelling within the model and for encapsulating object oriented ideas. So in the following we consider an OONET in the context of some valid object universe $\mathcal{U}$ (cf. definition 3.6).

As all tokens are distinguishable because of their unique identifications, the state of a system can now be represented by the set of all present tokens, instead of using a bag.

The connections between channels and processors are described as follows: Each processor has two sets of so-called connectors: the input- and the output-connectors. This defines the interface of the processor. Whenever a processor is ‘plugged in’ somewhere in a network, channels have to be connected to those connectors. One could think of the connectors as formal channel parameters of a processor. Which channels are connected to which processors in an OONET is described by a function called $\text{Match}$.

### 4.2 Formal Definition of the OONET Model

In this section, we will give the formal definition of the OONET model. After that, the formal semantics of the OONET model are described in terms of finite state machines. Finally, two theorems are proved. The first one implies that the identities of complexes in an OONET will always be different provided that the initial state of the OONET satisfies some condition: it has to be a so-called one-generation state (cf. definition 4.2). The second theorem states that the time concept of the OONET model is well-defined in the sense that time cannot go back.

**Definition 4.1**

Let $\mathcal{U}$ be a valid object universe with state space $SS_\mathcal{U}$ and identification structure $ID$ with child function $\mathcal{F}$.

We define an OONET to be a tuple $(R, V, Con, Match, \mathcal{U})$, where

- $PI = \text{dom}(R)$: the processor indices;
- $CI = \bigcup_{p \in PI}(\text{dom}(\text{Match}_p))$: the connector indices;
• $\text{Con} = (\text{ConI}, \text{ConO})$, where
  
  * $\text{ConI} : \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{I})$: for $p \in \mathcal{P}$, $\text{ConI}_p$ is the set of input-connectors of $p$;
  
  * $\text{ConO} : \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{I})$: for $p \in \mathcal{P}$, $\text{ConO}_p$ is the set of output-connectors of $p$.

  The following constraints have to be satisfied:
  
  * $(\forall p \in \mathcal{P} :: \text{ConI}_p \cap \text{ConO}_p = \emptyset)$: input- and output connector sets are disjoint;
  
  * $(\forall p \in \mathcal{P} :: \text{ConI}_p \neq \emptyset)$: each processor has at least one input connector.

• $\text{Match} : \mathcal{P}(\mathcal{I}) \rightarrow (\mathcal{I} \rightarrow \mathcal{L})$: for each $p \in \mathcal{P}$, $\text{Match}_p$ is a function assigning to each connector $c \in \text{ConI}_p \cup \text{ConO}_p$ a channel $x \in \mathcal{L}$ to which $c$ is connected;

• $V \in \mathcal{L} \rightarrow \mathcal{C}$, assigning to each channel $c$ the complex class denoting which complexes can be contained by $c$.

It has to hold in a state that each channel contains only complex objects that are element of the complex class of that channel. Formally:

$$(\forall \tau \in \mathcal{S} \Rightarrow (\forall \sigma :: \text{com}(\tau) \leq V(\text{loc}(\tau))))$$

• For $c \in \mathcal{P}(\mathcal{I})$ and $p \in \mathcal{P}(\mathcal{I})$, $R_p \in \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{I})$ is such that

  * $\text{dom}(R_p) = \{ e \in \mathcal{P}(\mathcal{I}) | (\forall c \in \mathcal{L} :: \# \{ \tau \in e | \text{loc}(\tau) = c \} = \# \{ x \in \text{ConI}_p | \text{Match}_p(x) = c \} \}^6$

  Intuitively, this means that for each processor $p$ and for each channel $c$, the number of tokens in $e$ with channel identification $c$ is equal to the number of input connectors of $p$ to which $c$ is connected. This is the so-called firing condition, which has to be satisfied for a processor to be enabled.

  * $(\forall \tau \in R_p(e) :: \text{loc}(\tau) \in \{ \text{Match}_p(x) | x \in \text{ConO}_p \})$.

  Intuitively, this means that only tokens are generated for output channels to which $p$ is connected.

  * $(\forall \tau \in e :: (\forall \tau' \in R_p(e) :: \text{at}(\tau') \geq \text{at}(\tau)))$

  This means that the time stamps of the generated tokens are equal to or larger than the largest time stamp of all consumed tokens.

  * $(\exists \tau \in R_p(e) \text{ id}(\tau) \in (\exists \tau \in R_p(e) \text{ id}(\tau)))$

  This means that the way identities are assigned to generated complexes satisfies the conditions of definition 3.1.

  * $(\exists \tau \in R_p(e) :: (\forall \sigma \in \text{cont}(\tau) :: (\text{sim}(\sigma) \in \mathcal{S} \Rightarrow \text{id}(\sigma) \in \mathcal{F}(\text{id}(\bigcup_{\tau \in e} \text{cont}(\tau)))) \wedge$

    $(\text{sim}(\sigma) \notin \mathcal{S} \Rightarrow (\text{id}(\sigma) \in \mathcal{F}(\text{id}(\bigcup_{\tau \in e} \text{cont}(\tau)))) \vee (\sigma \in (\bigcup_{\tau \in e} \text{cont}(\tau)))))))$

  This means that the way identities are assigned to generated simplexes of which duplicates in a state are not allowed satisfies the conditions of definition 3.1; simplexes of which duplicates are allowed get an identification according to definition 3.1 or they are a copy of a simplex in $e$.

  * $(\forall c \in R_p(e) :: (\forall s \in \text{cont}(c) :: (\exists i \in I :: i \notin \text{ref}(s) ::$

    $(\exists \sigma \in \bigcup_{\tau \in e} \text{cont}(\tau) :: \text{id}(\sigma) \equiv i) \vee$

    $(\exists \sigma \in \bigcup_{\tau \in R_p(e)} \text{cont}(\tau))) :: \text{id}(\sigma) \equiv i)$

  Here the inverse function is generalised such that $f^{-1}(y) = \{ x | f(x) = y \}$
Intuitively, this means that each simplex $s$ in a generated complex $c$ can only refer to some simplex $x$ with identity $i$ if $x$ was involved in the production of $c$, if $x$ is referred to by some other object involved in the production of $c$ or if $x$ is produced together with $c$.

In the definition given above we have given a minimal set of constraints each OONET should satisfy. We think the OONET model is well-defined this way without having imposed constraints which are not strictly necessary for each system which will be specified. It is easy to think of other constraints one may want to impose upon an OONET and give necessary and sufficient conditions for an OONET to satisfy these constraints. The way these constraints will be specified and validated in a specific case is a subject for further research.

We will give the formal semantics of the OONET model below, but before we are able to do so, we need an auxiliary definition.

**Definition 4.2** One-generation state

Let $O$ be an OONET as in definition 4.1. Then a state $\sigma \in SS_{\mathcal{U}}$ is called a one-generation state iff

$$(\forall \tau, \tau' \in \sigma :: \tau \neq \tau' \Rightarrow \neg(\exists n \in \mathbb{N} :: id(\tau) \in F^n(id(\tau'))))$$

This means that in a one-generation state the identity of a complex cannot be the descendant of the identity of another complex in the same state. As a consequence, all complexes generated from complexes in a one-generation state will have different identities from the remaining complexes in the state, because $ID$ is an identification structure. If a state would contain complexes $x$ and $y$ and $x$ is a descendant of $y$, then in a state transition $y$ could generate another descendant with the same identity as $x$. In a one-generation state, such an "identification clash" cannot occur.

In particular, a one-generation state cannot contain two different complexes $c_1$ and $c_2$ with the same identifications, because this would mean $c_1 \neq c_2 \land id(c_1) = F^0(id(c_2))$.

For the identities of simplexes form a class in $Simplex^*$, similar definitions and theorems can be formulated.

The following lemma states that if an OONET is in a one-generation state, it will remain in a one-generation state if $ID$ is an identification set as in definition 3.1. This implies that all complex identities in the state will remain different.
Lemma 4.1 Invariance of Unicity

Let $O$ be an OONET and $\sigma$ a one-generation state of $O$. Then the following holds:

$$\forall p \in PI : (\forall e \in F(\mathcal{C}) : e \in \text{dom}(R_p) \cap \sigma \Rightarrow \sigma' := (\sigma \setminus e) \cup R_p(e) \text{ is a one-generation state})$$

Proof

Let $O$ be an OONET according to definition 4.1. Let $p \in PI$ and assume that $O$ makes a state transition from a one-generation state $\sigma \in SS_U$ to another state under the firing of some processor $p$ which consumes a set of complex objects $\alpha \in \text{dom}(R_p) \cap \sigma$. We now have to prove that it holds that $\sigma' = (\sigma \setminus \alpha) \cup R_p(\alpha)$ is again a one-generation state.

Let $c, c' \in \sigma'$. In order to prove that $\sigma'$ is a one-generation state, we have to prove the following:

- $c \neq c' \Rightarrow \neg(\exists n \in \mathbb{N} : \text{id}(c) \in F^n(\text{id}(c')))$.  

Suppose that $c \neq c'$. Suppose furthermore that $\text{id}(c) \in F^n(\text{id}(c'))$ for some $m \in \mathbb{N}$. We will derive a contradiction now. We will do this by distinguishing three cases:

1. Suppose $c$ and $c'$ are both elements of $\sigma \setminus \alpha$. But in that case, $\sigma$ is not a one-generation state. This contradicts our earlier assumption.
2. Suppose $c$ and $c'$ are both elements of $R_p(\alpha)$. In that case, according to definition 4.1, there exist some $v, w \in \alpha$ such that $\text{id}(c) \in F(\text{id}(v))$ and $\text{id}(c') \in F(\text{id}(w))$.
   
   We have assumed moreover $\text{id}(c) \in F^n(\text{id}(c')) = F(F^{m-1}(\text{id}(c')))$. Thus it holds that $\text{id}(c) \in F(F^{m-1}(\text{id}(c')))$. Using the transitivity of $F$, $v \neq w$ holds (which follows from $\text{id}(v) \in F^n(\text{id}(w))$ for some $m > 0$ and lemma 3.2), this contradicts the assumption that $\sigma$ is a one-generation state.

3. In the third case one of the complexes $c$ and $c'$ belongs to $R_p(\alpha)$ and the other one to $\sigma \setminus \alpha$. We assume that $\text{id}(c) \in F^m(\text{id}(c'))$ for some $m \in \mathbb{N}$, so $c'$ was generated before $c$, so in this case we have $c \in R_p(\alpha)$ and $c' \in (\sigma \setminus \alpha)$. Here $m > 0$ holds for the same reasons as in clause 2 of this proof.
   
   It holds that $\text{id}(c) \in F(\text{id}(v))$ for some $v \in \alpha$, because of definition 4.1. On the other hand, we have assumed $\text{id}(c) \in F^m(\text{id}(c')) = F(F^{m-1}(\text{id}(c')))$. Therefore it holds that $\text{id}(c) \neq \text{id}(c')$, which contradicts our earlier assumption.

$\square$
**Theorem 4.2**
Let $\mathcal{O}$ be an OONET with initial state $\sigma_0$ and let $\sigma_0$ be a one-generation state. Then all reachable states of $\mathcal{O}$ are one-generation states.

*Proof*

Follows directly from lemma 4.1.

As has been mentioned on page 27, theorem 4.2 implies that all complex object identities are different in any reachable state from the initial state from an OONET, provided that this initial state is a one-generation state.

It might seem that the way we assign identities to objects is quite cumbersome, whereas we could simply have chosen a solution with one or more memory cells recording “the first free number”. However, we do not want to introduce memory cells for this assignment of identities. Moreover, we want to maintain our local state transition paradigm: when a processor fires, the state change which is a consequence of this firing can be determined by removing the consumed tokens and adding the generated ones.

The formal semantics of an OONET are defined by its process space. This will be defined in definition 4.4, which is a well-defined definition because of lemma 4.1. However, we will start by formalising some notions regarding time in OONETs which we have mentioned already in section 4.1.

**Definition 4.3 Event Set, Event and Transition Time**

Let $\mathcal{O}$ be an OONET with valid object universe $\mathcal{U}$ and $\sigma \in SS_\mathcal{U}$.

The *event set* $E(\sigma)$ of $\mathcal{O}$ in state $\sigma$ is defined by:

$$E(\sigma) = \{(p, \alpha) \in PI \times \mathcal{P}(\mathcal{C}) | \alpha \in \text{dom}(R_p) \cap \sigma\}$$

The *event time* of an event $e$ is defined by:

$$(\forall e = (p, \alpha) \in E(\sigma) :: etime(e) = \max\{at(\tau) | \tau \in \alpha\})$$

The *transition time* of a state $\sigma$ is defined by:

$$ttime(\sigma) = \min\{etime(e) | e \in E(\sigma)\}$$

Now we are able to give a formal definition of the *transition law* of an OONET $\mathcal{O}$. The transition law of $\mathcal{O}$ determines which states are reachable by $\mathcal{O}$ from a certain state.

The behaviour of $\mathcal{O}$ is determined by its *process space*: a possibly infinite sequence of states $(\sigma_0, \sigma_1, \ldots)$ where for each $i \in \mathbb{N}$ it holds that $\sigma_{i+1}$ is an element of the set $L(\sigma_i)$ of allowed states after the state $\sigma_i$. 
Definition 4.4 *Transition Law and Process Space*

Let $\mathcal{O}$ be an OONET. Then the transition law $L$ of $\mathcal{O}$ is defined by:

$$
L = \{ (\sigma, S) \in SS_H \times \mathcal{P}(SS_H) \mid
\forall \sigma' \in S :: (\exists e = (p, \alpha) \in E(\sigma) :: (\forall e' \in E(\sigma) :: etime(e') \geq etime(e)) \land
\sigma' = (\sigma \setminus \alpha) \cup R_p(\alpha)) \}\}
$$

The process space of $\mathcal{O}$ is the set of all finite or infinite sequences $(\sigma_0, \sigma_1, \ldots)$ such that $(\forall n \in \mathbb{N} :: \sigma_{n+1} \in L(\sigma_n))$ and $\sigma_0$ is a one-generation state, called the initial state of $\mathcal{O}$.

Definition 4.4 is correct because of theorem 4.2, as this theorem guarantees that the process space will only contain one-generation states provided that the initial state is a one-generation state.

It is trivial to prove that time is not going back in an OONET, which is expressed by the following theorem.

**Theorem 4.3**

Let $\mathcal{O}$ be an OONET according to definition 4.1. Suppose $\sigma' \in L(\sigma)$. Then $ttim(\sigma) \leq ttim(\sigma')$.

**Proof**

As $\sigma' \in L(\sigma)$, we have $\sigma' = (\sigma \setminus e) \cup R_p(e)$ for some $p \in PI$ and event $e$, by definition 4.4. But again using that definition and definition 4.3, we know that $etime(e) = ttim(\sigma)$. Application of definition 4.1 and again definition 4.3 then gives that for all $\tau \in R_p(e)$ it holds that $at(\tau) \geq etime(e)$. So events that were already possible at the moment that $p$ fired have at least event time $etime(e)$ because of definition 4.4 and events that are now possible have at least event time $etime(e)$ because all new tokens in $\sigma'$ have an availability time which is at least $etime(e)$ and using definition 4.3.

□

We will conclude this section by giving an example. We will use the aforementioned library example for this purpose.

**Example 4.1**

In the network on the next page, we see the two filials, the lenders and the central department. Filials each have a shelf-store with books. They get requests over their req-channels and returned books over their inbook-channels. They lend out via their outbook-channels. Moreover, they may send requests to the centre for a title when a member wants to obtain a title which is not present at that moment on their shelf. The centre orders such titles from another filial, which sends them by mail to the member who wants to lend it.
The only thing we left unspecified here is the functional behaviour of the processes. However, we do not want to go into detail about this here. One can read the ExSpect literature in order to learn the exact properties of the ExSpect language.

Figure 5: The library process model.

A formal representation of this net in terms of the OONET model is rather straightforward, but describing it is annoying clerical work.

☐
5 Conclusion and Future Research

We have seen how a net model and an object oriented data model can be integrated. We have given motivations and informal descriptions. We have also formalized the models. This is in our view essential to fully define and comprehend the models and to be able to implement them.

There are several aspects we have not dealt with in this paper. Some of them need more research, other ones have been investigated thoroughly but fall outside the scope of this paper. We want to mention some of these topics.

We have not dealt with the operations that can be applied onto simplexes and complexes. We will also integrate these operations in the ExSpect language. The same holds for constraints. There should be short-hand notations (probably graphically in the schema definition) to formulate constraints.

The process model gives new challenges for analysis methods: constraints formulated in the data model of a system should be invariants of the processors. So in a complete specification of a system one has to formulate such constraints and then verify their invariance. Powerful analysis methods are required in order to facilitate this verification process.

The OONET model is only suited for representing closed systems. However, we are currently working on an extension for the description of open systems too.

We have not yet gathered much experience with the use of these models. We expect that they will appear to be very useful, but we will have to validate their use in making some case studies. We are thinking of case studies in the fields of logistic systems, distributed databases and electronic data interchange (EDI). This will certainly give us new ideas. It may also result in some method which can be used when modelling distributed (information) systems.

Finally, we are currently extending the ExSpect CASE tool to deal with the new ideas that have been described in this paper.
Acknowledgement

The authors want to thank Jan Paredaens and Geert-Jan Houben for the inspiring discussions we have had together and for their useful remarks, which certainly have improved both our insights and the quality of this paper.
References


36
In this series appeared:

<table>
<thead>
<tr>
<th>No.</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>89/1</td>
<td>E.Zs.Lepoeter-Molnar</td>
<td>Reconstruction of a 3-D surface from its normal vectors.</td>
</tr>
<tr>
<td>89/2</td>
<td>R.H. Mak, P.Struik</td>
<td>A systolic design for dynamic programming.</td>
</tr>
<tr>
<td>89/3</td>
<td>H.M.M. Ten Eikelder, C. Hemerik</td>
<td>Some category theoretical properties related to a model for a polymorphic lambda-calculus.</td>
</tr>
<tr>
<td>89/4</td>
<td>J.Zwiers, W.P. de Roever</td>
<td>Compositionality and modularity in process specification and design: A trace-state based approach.</td>
</tr>
<tr>
<td>89/5</td>
<td>Wei Chen, T.Verhoeff, J.T.Udding</td>
<td>Networks of Communicating Processes and their (De-)Composition.</td>
</tr>
<tr>
<td>89/6</td>
<td>T.Verhoeff</td>
<td>Characterizations of Delay-Insensitive Communication Protocols.</td>
</tr>
<tr>
<td>89/7</td>
<td>P.Struik</td>
<td>A systematic design of a parallel program for Dirichlet convolution.</td>
</tr>
<tr>
<td>89/9</td>
<td>K.M. van Hee, P.M.P. Rambags</td>
<td>Discrete event systems: Dynamic versus static topology.</td>
</tr>
<tr>
<td>89/10</td>
<td>S.Ramesh</td>
<td>A new efficient implementation of CSP with output guards.</td>
</tr>
<tr>
<td>89/11</td>
<td>S.Ramesh</td>
<td>Algebraic specification and implementation of infinite processes.</td>
</tr>
<tr>
<td>89/12</td>
<td>A.T.M.Aerts, K.M. van Hee</td>
<td>A concise formal framework for data modeling.</td>
</tr>
<tr>
<td>89/14</td>
<td>H.C.Haesn</td>
<td>ELDA, data manipulation syntax.</td>
</tr>
<tr>
<td>89/15</td>
<td>J.S.C.P. van der Woude</td>
<td>Optimal segmentations.</td>
</tr>
<tr>
<td>89/16</td>
<td>A.T.M.Aerts, K.M. van Hee</td>
<td>Towards a framework for comparing data models.</td>
</tr>
<tr>
<td>89/17</td>
<td>M.J. van Diepen, K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra.</td>
</tr>
<tr>
<td>Page</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>90/2</td>
<td>K.M. van Hee P.M.P. Rambags</td>
<td>Dynamic process creation in high-level Petri nets, pp. 19.</td>
</tr>
<tr>
<td>90/4</td>
<td>A. Peeters</td>
<td>Decomposition of delay-insensitive circuits, p. 25.</td>
</tr>
<tr>
<td>90/5</td>
<td>J.A. Brzozowski J.C. Ebergen</td>
<td>On the delay-sensitivity of gate networks, p. 23.</td>
</tr>
<tr>
<td>90/8</td>
<td>M.B. Josephs</td>
<td>Receptive Process Theory, p. 16.</td>
</tr>
<tr>
<td>90/9</td>
<td>A.T.M. Aerts P.M.E. De Bra K.M. van Hee</td>
<td>Combining the functional and the relational model, p. 15.</td>
</tr>
<tr>
<td>90/10</td>
<td>M.J. van Diepen K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).</td>
</tr>
<tr>
<td>90/11</td>
<td>P. America F.S. de Boer</td>
<td>A proof system for process creation, p. 84.</td>
</tr>
<tr>
<td>90/12</td>
<td>P.America F.S. de Boer</td>
<td>A proof theory for a sequential version of POOL, p. 110.</td>
</tr>
<tr>
<td>90/14</td>
<td>F.S. de Boer</td>
<td>A proof system for the language POOL, p. 70.</td>
</tr>
<tr>
<td>90/15</td>
<td>F.S. de Boer</td>
<td>Compositionality in the temporal logic of concurrent systems, p. 17.</td>
</tr>
<tr>
<td>90/16</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>A fully abstract model for concurrent logic languages, p. 23.</td>
</tr>
<tr>
<td>90/17</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>91/02</td>
<td>R.P. Nederpelt, H.C.M. de Swart</td>
<td>Implication. A survey of the different logical analyses &quot;if..., then...&quot;, p. 26.</td>
</tr>
<tr>
<td>91/03</td>
<td>J.P. Katoen, L.A.M. Schoenmakers</td>
<td>Parallel Programs for the Recognition of P-invariant Segments, p. 16.</td>
</tr>
<tr>
<td>91/05</td>
<td>D. de Reus</td>
<td>An Implementation Model for GOOD, p. 18.</td>
</tr>
<tr>
<td>91/06</td>
<td>K.M. van Hee</td>
<td>SPECIFICATIEMETHODEN, een overzicht, p. 20.</td>
</tr>
<tr>
<td>91/07</td>
<td>E.Poll</td>
<td>CPO-models for second order lambda calculus with recursive types and subtyping, p.</td>
</tr>
</tbody>
</table>