Modelling object-oriented systems by Transformation Systems

Diplomarbeit im Fach Informatik,TU-Berlin
Vorgelegt von Andi Rayo Kniep

01. Januar 2005

1. Gutachter:
Prof. Hartmut Ehrig
2. Gutachter:
Dr. Gabriele Taenzer

Die selbstständige und eigenhändige Anfertigung dieser Arbeit versichere ich an Eides statt.
Berlin, den 01. Januar 2005

_____
Andi Rayo Kniep

Zusammenfassung

In der vorliegenden Arbeit werden “Objekt-Orientierte Transformations-Systeme” als semantischer Rahmen für die Integration von heterogenen objekt-orientierten Systemen eingeführt. Objekt-Orientierte Transformations-Systeme ermöglichen die Modellierung von Instanzen dieser Systeme durch eine mathematisch-formale Semantik und dadurch deren Vergleich und Konsistenzprüfung.
Um den Begriff “Objekt-Orientierung” abzugrenzen und für diese Arbeit zu definieren, werden die Merkmale von heutigen objekt-orientierten Systemen identifiziert und als Charakteristiken der Objekt-Orientierung herausgearbeitet.
Ein Ergebnis der Arbeit ist der Beweis, dass Objekt-Orientierte Transformations-Systeme zu einer “Concrete Institution” erweitert werden können und daher eine Instantiierung generischer Transformations-Systeme von M. Große-Rhode darstellen. Dies ermöglicht die Anwendung bereits vorhandener und bewiesener Theoreme wie der Komposition von Transformations-Systemen und verschiedener Entwicklungsrelationen auf Objekt-Orientierte Transformations-Systeme.

Abstract

In this paper, “Object-Oriented Transformation Systems” are introduced as a general semantical framework for the integration of possibly heterogeneous object-oriented systems. Instances of these systems can be modelled by Object-Oriented Transformation Systems within a formal mathematical semantic enabling comparison and consistency check between them.
In order to clarify the term “object-orientation” the characteristics of today’s object oriented systems are identified.
As a result of this paper Object-Oriented Transformation Systems are proven to form a concrete institution and thus constitute an instantiation of the generic Transformation Systems by M. Große-Rhode. Hence existing and proven theorems like the composition of Transformation Systems and development relations can be applied to instances of Object-Oriented Transformation Systems.

Contents

Chapter 1

Introduction

1.1. Motivation

In software development a large number of people work together on implementing a software system. In order to co-ordinate their activities the targeted system is specified centrally during the whole process and particularly before its beginning. It is this specification to which the developers' implementation has to conform to and can be checked against.
The importance of a universal specification appears in the existence of countless specification techniques, led by the UML (Unified Modeling Language), the most established one. However the latter is not a specification language itself, but an accumulation of so-called viewpoint specification techniques, i.e. every technique is used to model a certain aspect of a system. One system is described by several specifications, possibly created by several persons. Apparently, the specifications may happen to be inconsistent: far too often developers fail to know existing drafts of the same system and therefore are not aware of their specification conflicting with others.

The most relevant question in this context is: "Is there a system-model that satisfies all the requirements that are imposed by the various specifications?"
The latter is a question the DFG[1]-project IOSIP[2] deals with, as the integration of (possibly heterogeneous) specification techniques is one of the project's main goals. In a first approach it was M. Große-Rhode, who developed Transformation Systems (see [Gro01]), offering to be a reference semantic for integrating the semantics of various specifications into a combined model. Being based on algebraic structures and techniques Transformation Systems provide a formal mathematical environment for the description of the desired system.[3]
If the various specifications can successfully be integrated in a Transformation System, a so-called combined model will be found, being a formal description of the targeted software system and thus proving the consistency, or rather compatibility, of the integrated specifications.
Transformation Systems by M. Große-Rhode are designed for specification languages in general. In order to translate a specification into a Transformation System both all structures and all data need to be expressed by given algebraic structures or new structures in the form of a data space signature and so-called data states have to be defined.

---

1 DFG = „**D**eutsche **F**orschungs**g**emeinschaft" (German Research Council)

2 IOSIP = **I**ntegration of **o**bject-oriented software **s**pecification techniques and their application-specific extension for **i**ndustrial **p**roduction systems, a project supported by the DFG.

3 See paragraph 3.1.1. for a detailed description of Transformation Systems.

Today, the object-oriented approach to software development has prevailed: large software-systems are written in object-oriented programming languages like C++, Java, C#, but also SmallTalk and Eiffel. Even today's scripting languages[4] use the principles of object-orientation by supporting the definition and instantiation of classes. For this reason major specification languages like the UML are object-oriented.
Hence the need for a reference semantic of software systems requires an object-oriented framework.

1.2. Goals

It is this paper's intention to define Object-Oriented Transformation Systems as a formal reference semantic for general object-oriented systems. Furthermore, it is intended to create a framework that is comprehensible and thus usable for persons that are familiar with current object-oriented systems.

One of the main purposes is the possibility of translating (or rather constructing) the semantics of object-oriented systems into Object-Oriented Transformation Systems, in order to combine the semantics[5] to a complete system-model or to compare them with each other (consistency check).

Though Object-Oriented Transformation Systems should be specialised in aspects of object-orientation[6], they should be general and open enough to allow the integration of various object-oriented systems.

Nevertheless Object-Oriented Transformation Systems must be capable particularly of major object-oriented systems of the present. The expression "major object-oriented systems of the present" refers to programming languages like Java and C++ on the one hand and specification techniques like the ones of the UML on the other hand. It will be used throughout this paper in order to refer to this meaning.
Since Object-Oriented Transformation Systems should subserve persons that are familiar with these systems, its structure should be geared to major object-oriented systems of the present by being the smallest common denominator of object-orientation. The advantage of the formal resemblance in structure will facilitate the integration of the mentioned systems into Object-Oriented Transformation Systems, as, for a large part, the translation work can be accomplished based on existing knowledge[7].

As mentioned above Transformation Systems by M. Große-Rhode are designed as a reference semantic for different specification techniques. In [Gro01] M. Große-Rhode presents a rich selection of theoretical results for Transformation Systems; these include the composition of Transformation Systems as well as development relations like "extension" and "reduction" along data space signature morphisms.

---

[4] Scripting languages are not as powerful as system programming languages/computer languages: also known as batch or job control languages they usually can not be compiled to machine-code, do not support typing and are executable only within certain programs. Hence their power is limited by the program they are interpreted by. Examples for this are PHP, Perl, VisualBasic, JavaScript and several other Java-derivates that are used in different programs to subsume sequences of instructions.

[5] The semantics of several specifications/systems.

[6] A special case of Transformation Systems, see below.

[7] Coming apparent in terms of intuition.

In order to take advantages of these theorems the resulting framework has to be based on Transformation Systems by M. Große-Rhode: Object-Oriented Transformation Systems have to be defined as an instantiation of the generic Transformation Systems[8].

The goals that have been described before have to be achieved with the following subgoals:

- **General semantical framework (G1)**[9]
  OOTS offer a general and common semantical framework for the interpretation of object-oriented systems.
- **Instantiation of generic Transformation System (G2)**
  OOTS are an instantiation of generic Transformation Systems by M. Große-Rhode.
- **Consistency check and comparison (G3)**
  OOTS can be used for consistency check among and comparison between different possibly heterogeneous object-oriented systems.
- **Object-oriented structure (G4)**
  OOTS have an object-oriented structure, that is adapted from today's object-oriented systems[10]. OOTS themselves constitute an object-oriented system.
- **Translation of object-oriented systems (G5)**
  OOTS allow the translation, i.e. semantical integration, of heterogeneous object-oriented systems into a common semantical domain.
- **Easy to comprehend and understand (G6)**
  OOTS have a structure and a layout that is easy to comprehend and understand, especially for people familiar with object-oriented systems.
- **Presentation of the dynamic semantics of object-oriented systems (G7)**[11]
  OOTS illustrate a comprehensible presentation of object-oriented systems and their actual processes, i.e. dynamic semantics.

There are two related approaches with similar goals, both instantiating Transformation Systems of M. Große-Rhode as well:
The first involves the reports of J. Tenzer and D. Parnitzke, both describing UML class diagrams by modified versions of Transformation Systems (see [Ten01], [Par01]). M. Große-Rhode later took up these approaches in an example of use for Transformation Systems (see section 3.2.1.).
The second by B. Braatz, M. Klein and G. Schroeter approaches from another perspective: viewpoint-specifications are the main issue of their paper (see [BKS04]). In order to constitute a framework for a semantical integration of viewpoint-specification object-oriented Transformation Systems[12] are defined, claiming to be a reference semantic for object-oriented systems in general; a pretension that resembles the goals of this paper (see section 3.2.2.).

---

[8] "Generic Transformation Systems", in contrast to "ordinary Transformation Systems", are based on (concrete) institutions constituting the system's data models and thus open for arbitrary data spaces (ordinary Transformation Systems are based on algebras, representing data states).

[9] If this goal is referred to within this paperthe shortform "goal G1" will be used.

[10] In particular systems like Java, C++ and the UML.

[11] As a side-effect of their object-oriented structure Object-Oriented Transformation Systems should offer a comprehensible representation of the processes of a modelled object-oriented system: often the actual steps of a software system and its methods remain hidden. Since Object-Oriented Transformation Systems also model the dynamic semantics, there is a way to represent and thus understand these processes.

[12] Other than Object-Oriented Transformation Systems as presented by this paper: B. Braatz, M. Klein and G. Schoeter used the same term to name their framework.

Existing approaches vs. targeted goals

Both approaches agree with this paper on some of the goals that where enumerated above, but contradict others. They will be considered in detail in this paper (see chapter 3), identifying lacking features, but also qualities that should be adopted by Object-Oriented Transformation Systems.

1.3. Structure of the paper

The task of defining a framework that is able to constitute the semantics of object-oriented systems in a clear manner as simple and comprehensible as possible yields the following questions:

- What are object-oriented systems? (What systems are embraced by this term?)
- What is and what makes object-orientation?

The lack of a definition evokes the problem of the term "object-orientation" being understood differently by different persons and in different times.

Therefore in **chapter 2** the expression "object-orientation" is investigated for the scope of this paper by identifying object-oriented concepts of nowadays. The latter are then grouped into so-called "characteristics of object-orientation". Every system that supports or is able to express these characteristics is then called object-oriented. At the end of chapter 2 these terms will have been clarified including the tasks of Object-Oriented Transformation.

Subsequently, **chapter 3** presents existing related approaches and checks on the identified object oriented characteristics, i.e. checking the approaches' ability of presenting them. In this context Transformation Systems by M. Große-Rhode are introduced as well, being both the basis of this paper and of related approaches. Beyond this, in chapter 3 the intention is to determine which qualities/advantages of existing approaches could be taken over or should be adopted into Object-Oriented Transformation Systems.

Chapter 4 then contains the actual definition of Object-Oriented Transformation Systems. The definition is split into three parts: a syntactical one, introducing the data space signature, class signatures and data space signature terms. The semantical part contains the definition of system states. A system state denotes a snapshot of the represented system, i.e. being an instantiation of the data space signature. The last part introduces actions w.r.t. a given data space signature. Actions describe transitions between system states, so-called transformations.
Following the actual definition it is proven that OOTS are an instantiation of generic Transformation Systems by showing that they form a concrete institution in the sense of [GB84] and [GB92].
After that the basic framework is extended to concurrent Object-Orient Transformation Systems. Therefore, some parts of the original definition have to be adjusted and a new kind of actions will be introduced.

Chapter 5 implements a concrete example: on the basis of an exemplary object-oriented system, an Object-Oriented Transformation System is instantiated and presented. Furthermore, since Object-Oriented Transformation Systems claim object-orientation and the ability of describing object-oriented systems, chapter 5 checks the framework's claim by means of the characteristics, identified in chapter 2.

1.4. Methodological remarks

At last some methodological decisions which were taken when writing this paper are pointed out and otherwise might be misunderstood or overlooked:

Boldface

There is no semantical meaning by the use of boldface-text, e.g. the terms **OOTS** and OOTS are semantically equal. Boldface-text passages are used to emphasize the subject of a paragraph, e.g. the focus of a definition or the point of a remark.

Abbreviations

In order to refer to certain definitions abbreviations are introduced: in particular constraints that have to be satisfied by every Object Oriented Transformation System will be referred to as OTCxx, where xx stands for a short term of the constraint's intention, e.g. constraint OTCsorts requires every Object-Oriented Transformation System to hold the data sort "bool" $\in$ DS (see definition 1).

Apart from constraints so-called paper conventions will mostly be abbreviated by PCONxx[13]: conventions are conditions (requirements) that do not have to be satisfied by an Object-Oriented Transformation Systems, but their satisfaction will be assumed within this paper to ease the specification-work at some points of this paper, i.e. not to detract from the points of definitions/remarks.
Hence existing conventions are valid throughout the paper. Eventually, since they conform to the conditions of major object-oriented systems like Java, C++ and the UML, paper conventions do not restrict Object-Oriented Transformation Systems in the majority of applications.

The third and last kind of abbreviations that are used in this paper refer to the object-oriented characteristics that are identified in chapter 2. Since these characteristics constitute the paper's effective definition of object-orientation they will be referred to throughout this paper (see appendix A).

In order to keep track of all the abbreviations within this paper they will all be listed in appendices A – C, along with section- and page number of first appearance.

Special terms

Finally, there are some terms appearing in this paper that should be briefly explained in order to avoid confusion. The term "dynamic semantics" specifies the actual processes of a running system. Since the whole paper deals with the semantics of object-oriented systems, the semantics of transformation during its execution have to be treated as well. This includes mainly the actual steps of running methods, i.e. their modifications done to the system (system's state).
The term "dynamic semantics" refers to these actual processes that are rarely[14] considered when specifying a system on the one hand, but nevertheless have to be expressed by a reference semantic on the other.

---

[13] Where xx is a short-term referring to the convention's purpose.

[14] System specifications focus mainly on the effects and results of methods, by specifying pre- and post-conditions to give an example, but less on how these goals are achieved in a running system.

The term “system state” is used synonymously for “data state”. The latter refers especially to the state of all data, provided by the values of attributes. A system state is also a term that is used when defining Object-Oriented Transformation Systems, constituting a detailed description of the system’s actual state (see definition 3).

The term “Object-Oriented Transformation Systems”, short “OOTS”, refers to the semantical framework that is defined in this paper. Since it is a proper name it is written capitalised in contrast to object-oriented Transformation Systems, a term that does not refer to the definition of this paper, but to any object-oriented version of Transformation Systems. In chapter 3 this term is frequently used to denote the approach to an object-oriented reference model by Braatz, Klein & Schroeter (see section 3.2.1.).
The term “Transformation Systems” is capitalised throughout this paper since it is a proper name for the approach by M. Große-Rhode that constitutes the basis of this paper as well as the existing related approaches (see [Gro01]).

Chapter 2

Characteristics of object-oriented Systems

Since it is the goal of this paper to introduce a framework that is able to describe object-oriented systems, Object-Oriented Transformation Systems have to embody all features these systems provide. The following chapter should identify and determine which characteristics make up an object-oriented system of nowadays. The latter offer many more features than the ones originally intended by object-oriented concepts. However with the time object-oriented systems became understood to provide these additional properties as well.
Eventually, it remains indefinite which characteristics are meant by talking about object-orientation.

2.1. The principle of object-orientation

The fundamental question is: "what is object-orientation?" The question itself indicates a first answer: the "object" resides in the focus of an object-oriented design, being the building block every object-oriented system is constructed out of. An object constitutes an entity of a software system, having a current state and a behaviour, described by data values (attributes) and operations that it can perform, respectively. More precisely an object packs attributes and logically linked methods to a unit. This concept is called "data encapsulation".
Furthermore every object has a distinct identity, underlining the object's nature of being an individual system's entity. There is a fourth component, known as an object's interface, describing what messages an object can understand, i.e. handle. As objects communicate with each other, messages are sent among objects and can be understood as the invocation of visible method, also known as "routines". The object's routines that are not "visible" are hidden from the outside. Usually only routines that serve an object's purpose are visible. Others serve internal purposes that may only be clear to the object itself; thus they remain "invisible".
The concept of hiding an object's part that does not support the object's purpose, i.e. its external use, is known as data- or information-hiding. Since an object's data can only be accessed through the interface, making the data itself abstract to the outside, it is also spoken of "data abstraction".

Eventually in object-orientation the object takes centre stage. Not the actions (methods) that manipulate data states, but the objects calling these methods centre the consideration of an object-oriented system. Not footloose data, but unique objects that can pass off and be created on the fly (at runtime), represent the situation and activity of an object-oriented system. The systems itself is then constituted by an open set of objects communicating with each other, i.e. sending messages to each other.

So far we can answer the question "what is object-orientation": a system is object-oriented if it is build upon objects communicating with each other, where every object represents an individual entity of the described system.

2.2. What defines object-orientation?

The question "what defines of object-orientation?" may seem confusing as we just answered the question "what is object-orientation?" However the following section is not supposed to repeat the concepts mentioned in the preceding section, but should identify features and characteristics that evolve from the over-all concept of object-orientation as described in section 2.1..
Although the previous section gives a clear answer as to which concept forms the basis of any object-oriented system, it did not state what features make an object-oriented system.

Example
A "motorised passenger car" strictly speaking is not more than the combination of a motor and the possibility to transport a person, but nevertheless every car has tyres, doors, seats and much more "features". Today you expect a car to have a (clear) dashboard, mirrors and safety systems. Eventually the common perception has developed, concerning what the properties of a car.

In the same manner the perception changes, concerning object-oriented systems. This and the following section present concrete features of major object-oriented systems of the present. Thereby object-orientation should be determined, i.e. defined, on the basis of the characteristics of object-oriented systems.
There are two major reasons for this approach:

1. There is no (widely recognized) definition of object-orientation in genera
2. There is no inter-country commission or group summarising and publishing the actual features of object-orientation.

As a result of this approach we are going to identify more characteristics than only those that were originally indented by the idea of object-orientation.

2.2.1. Structuring methods / type-system

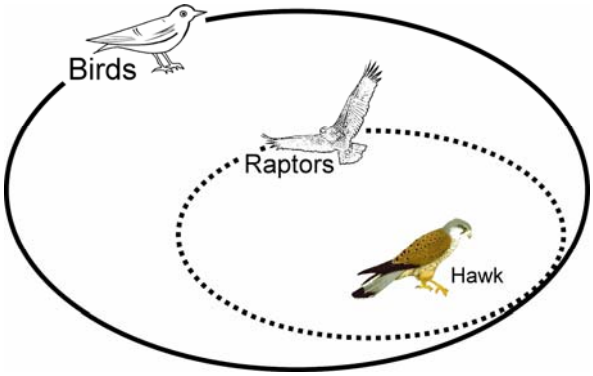
Besides data encapsulation and abstraction the original idea of object-orientation also promised some instant advantages for the user like structuring methods, lucidity and code reuse (with programming languages).
An object-oriented system can be structured by a "type-system", also known as "class-concept", and the concepts of generalisation and aggregation.

A type-system introduces types, grouping objects of the same kind, i.e. same behaviour, into a "class of objects of the same type". The class is then called the type of the grouped objects. Since objects of the same type only differ in the values of their attributes, only the system's classes and their relations are regarded when considering the system's structure.

These relations are mainly the mentioned concepts of generalisation and aggregation. Today the former is often known as the concept of inheritance. If an object o1 is a generalisation of an object o2, o2 is called a refinement of o1, inheriting from o1, i.e. o2 has the same properties like o1, but may have additional (specialised) attributes. Let us assume t1 is the type of o1 and t2 the type of o2. We call the t1 a supertype of t2 and vice versa t2 a subtype of t1.

Example

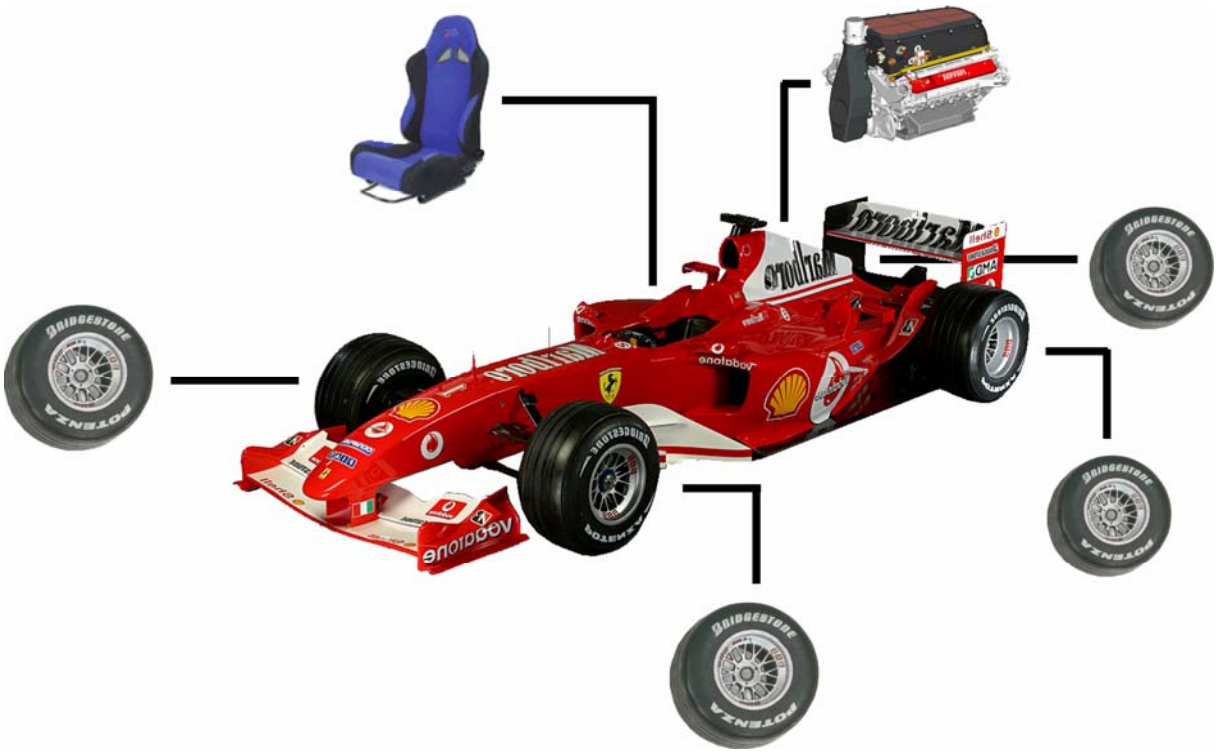
Bird is a generalisation (umbrella word) of hawk. Hawk inherits the properties of bird, e.g. laying eggs, but is more distinct as it also has the property of being a raptor, i.e. a carnivorous bird;
(see upcoming paragraph 2.3.2).



Aggregation marks the building of new objects based on existing objects, where an object becomes part of another:

Example

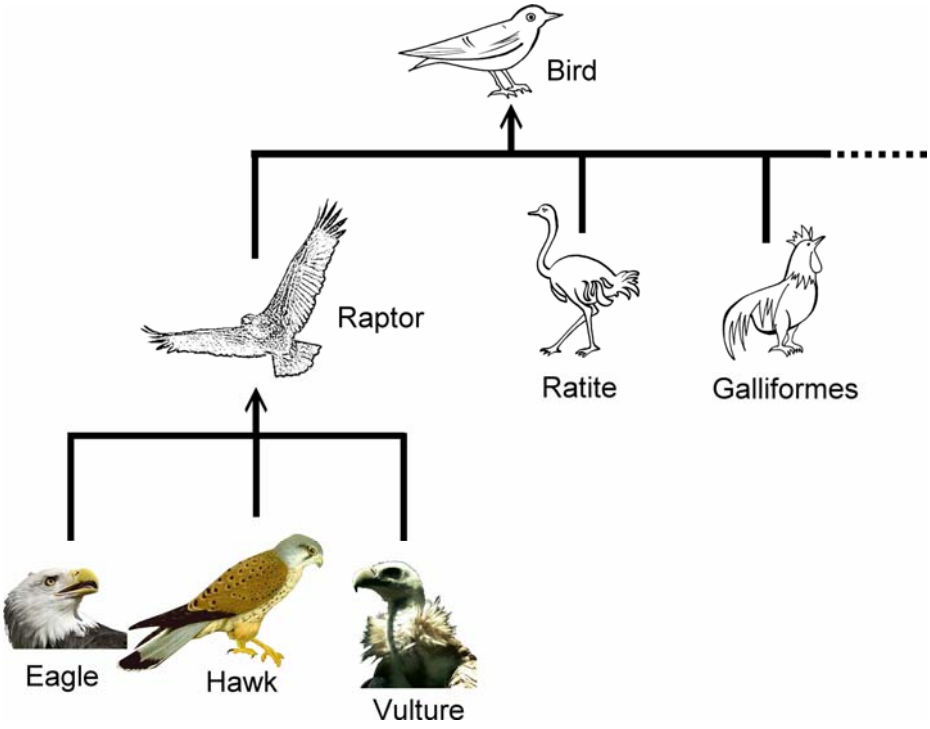
A (formula one) car aggregates a motor, a seat, tyres (and more):



Generalisation and aggregation are important structuring methods and relationships between classes (and thus between objects). By expressing relations between objects, the two concepts structure the usually immense space of objects of an object-oriented system, i.e. relations between classes structure the type-system.

Example

All carnivorous birds have the same type: raptor; being part of the class of raptors. Raptors, Ratites, Galliformes and others on the other hand are classified as Birds.



Technically the type of an object is described by a class and it is said that objects are instances of classes. The class of a certain type defines the behaviour of the type's objects and the attributes used to describe their state. All objects of the same type resemble in behaviour and have the same attributes, but differ from each other, having unique identities and possibly different attribute values.

Example

Though all carnivorous birds are raptors, every bird alone has its unique identity; and even if the attributes used to describe one raptor are the same, their values differ from raptor to raptor, since the birds differ in size, age, feather-colour and so on.

Most object-oriented systems feature a type-system and the structuring methods generalisation and aggregation. Hence these concepts belong to the most important principles of object-orientation itself.

2.2.2. Concrete object-oriented systems

Certainly object-oriented systems are systems that are object-oriented in the sense of section 2.1., but this paragraph aims more at the different existing kinds of object-oriented systems.
The term object-orientation originates in the context of programming languages, where Simula '67 is said to be the first programming language constructed upon object-oriented concepts (see [BDMN73]). While trying to solve a combinatorial explosion within ship simulations, the designers of Simula '67 had the idea to group different types of ships into different classes, each class being responsible for defining its own data and behaviour. Later this idea was refined when the programming language Smalltalk was designed aiming for a fully dynamical system, in which objects could be created and modified at runtime (see [KAY80]). After more and more new object-oriented programming languages like Eiffel emerged, the dominance of object-orientation ultimately peaked with the rising popularity of graphical user interfaces, for which object-oriented programming is allegedly well-suited (see [Mey01],[RBP01],[Jac01]).

Today the object-oriented programming technique is the most widespread with prominent representatives like C++, Java, Perl and many of today's script languages (see section 1.2.).
All object-oriented programming languages have more or less different implementation of concepts like inheritance (generalisation) and aggregation. Different languages want to emphasise different concept and the so-called "clean implementation" of one concept may affect the implementation of another.

Example
While the programming language Eiffel is offering an extensive system of inheritance, Java is setting a higher value on "security" and tries to eliminate foreseeable problems. This way Java prohibits the concept "multiple inheritance" as it implicates the largest problem of generalisation: for a type, being a subtype to two different types that embody the same method m, it may be ambiguous from which of the two types to inherit m from (see following paragraph 2.3.2). Eiffel leaves it largely to the user/programmer to avoid, or rather clarify emerging problems and ambiguities.

However mostly all these languages agree on the implementation of lots of concepts, including also many features that are not originally associated to the idea of object-orientation. While working with object-oriented programming languages, developers and programmers got used to these features and thus today they may confidently be called concepts of object-orientation, including catch phrases like "access levels", "concurrency", "genericity".

Returning to the question of this section: object-oriented systems do not only cover the area of object-oriented programming (OOP). Today object-oriented systems are mainly found over the field of object-oriented software development and engineering (OOSD and OOSE, respectively), including programming and specification techniques, but even spreading beyond these, e.g. object-oriented system analysis, object-orientation in graphical systems.
This paper mainly focuses on object-oriented specification techniques like class diagrams, state charts, sequence diagrams etc. and languages, embracing these techniques like EXPRESS, ROOM (Real-Time Object-Oriented Modeling) and particularly the UML (Unified Modeling Language)., probably the most established of the mentioned.

Specification languages constitute a very important part of today's software engineering projects as it is the casual way of developers to coordinate their work in a team and on the same project. Specifications made by the use of specification techniques describe the targeted software system and are the link between the participating engineering groups. For this purpose object-oriented specification languages offer ways to describe software systems as extensive and detailed as possible, comprising all reasonable possibilities to design and structure object-oriented systems. Although they specify software systems that are realised by programming languages they are not linked, i.e. constrained to any certain programming language, presuming that all of today's object-oriented programming languages are capable of expressing the used features. The latter fact makes specification languages to some kind of indicators which concepts can be expected of object-oriented systems (programming languages) today, i.e. a collection of object-oriented concepts.

Example
Specification languages like the UML support the use of several access levels (like private, protected, package), exception handling and overloading of methods, to name a few concepts that were not originally intended by object-orientation. Furthermore specifications, created with these languages, mostly have to be implemented by programming languages and thus expect all programming languages to embody these concepts.

Supporting these concepts and expecting other object-oriented systems to be familiar with these, eventually makes all these features to concepts of object-orientation itself. It should be the intention of this paper to comprehend all object-oriented concepts of today. On the basis of today's most common programming languages like Java and C++ and predominantly used specification languages like the UML we will see that the identified concepts are commonly used and supported in the majority of these object-oriented systems even there neither is a shared definition of object-orientation nor do all of these features originate in the basic paradigms of object-orientation

2.3. Characteristics of object-orientation

The following section is going to identify all features that can be called object-oriented concepts, i.e. all qualities that major object-oriented systems of the present have in common. Thus, today, an object-oriented system is expected to provide these features.
Therefore object-oriented concepts of the same domain are grouped to so-called characteristics of object-orientation. Every characteristic is introduced by one of the following paragraphs. Eventually the identified object-oriented characteristics should later serve as definition of object-orientation, in order to properly define Object-Oriented Transformation Systems.

2.3.1. The object as smallest unit

This first characteristic sums up all the concepts that were originally intended by the idea of object-orientation.

As stated in section 2.1 in every object-oriented system objects constitute the smallest possible component of the system's structure. Since objects are grouped to classes, the structure usually defines a "type-system", also known as "class concept". Classes define the attributes, the methods, and the interface for all objects of the described type. The individual objects still differ due to their unique identities and different attribute-values.

Having the object as indivisible element, the terms “data encapsulation” and information hiding come into play: data and methods that belong together are packed into objects representing an entity of the system. Since objects encapsulate all information of the system they act as black boxes to their environment, hiding their inside-information. Following the concept of “information hiding”, the data of an object is only accessible through its interface: only those methods are available that directly serve the object’s purpose to the outside.
“Information hiding” serves the overall data security: an object’s attributes can’t be manipulated directly from other objects and thus no contradictory or inconsistent state can be generated. An object can provide methods to (indirectly) manipulate its attributes: it is that method’s task to prevent inconsistent data states. The latter is called “data abstraction”, since the data can only be accessed through the respective methods on an abstract level.

Those parts of an object that are hidden from the environment are called “private”. Visible/accessible parts are called “public”. Normally attributes, constituting the state of an object, are not “public” and can only be modified via “public” methods. Public methods are also called messages that an object understands where a message has to be understood as a method call. All public methods together form the interface of an object.
“Private” and “public” are called access levels. In between new access levels have evolved, e.g. “protected” qualifies methods of an object o1 that are available to objects of the same type or a subtype (of o1’s type); “package” restricts the access to an object’s method to objects of the same package.
These and other access levels have been established in the course of introducing new concepts like packaging and inheritance.

Together the mentioned concepts from the characteristic “object as smallest unit”:

- object as atomic component
- type-system / class-concept
- data encapsulation
- information hiding
- access levels
- data abstraction

This characteristic constitutes the most fundamental principle of object-orientation.

2.3.2. Generalisation

The characteristic “generalisation” comprises the concept of generalisation, as described in paragraph 2.2.1, along with all concepts and ideas that arose from it.

The existence of a type-system, being part of the preceding characteristic, implies the existence of structuring methods like generalisation and aggregation (see paragraph 2.3.3.). The former means inheritance among types:

Example
Let us assume type T_0 is a generalisation of T_1, making T_0 a supertype/parent of T_1. Thus T_1 is a subtype of T_0 and inherits T_0’s behaviour, including the private parts of T_0. In addition the subtype T_0 can extend and/or refine the inherited behaviour.

An object obj is called an instance of a class C, if it contains the behaviour, defined by the class ($BehSig_C$). Therefore C has to be the type of obj, i.e. obj is a C-object, or an ancestor of t hat type. Hence instances have (all) the behaviour that is defined by the class and may have more.
Within a type-system an object has a well defined type, but can be the instance of several classes:

Example
The type of a hawk is Raptor, but a hawk is also an instance of Carnivore, Bird, Vertebrate, and so on, i.e. Raptor can be classified as Carnivore, Bird, Vertebrate, etc.[15]

The concept of inheritance always has to face up to the question of multiple inheritance and the problems connected with it: repeated or even recursive inheritance. The specific concept of "multiple inheritance" permits a type to have two or more supertypes.

Example
Let us assume a class C_1, being the ancestor to another class over two different inheritance-paths, known as "repeated inheritance". There will be a class C_2, where it becomes cloudy from which supertype to inherit the behaviour from that has indirectly been inherited from the base class C_1: the possibilities include inheriting the behaviour twice (disjoint), inheriting it from only one of the superclasses, merging the overlapping parts (if compatible). It gets even worse, when a class becomes the ancestor of itself, known as "self- inheritance"…

Actually the mentioned problems denote only the tip of the iceberg as solving one problem of multiple inheritance often makes new problems emerge. Many programming languages, e.g. Java, restrict the use of "multiple inheritance" to avoid the mentioned problems. On the over hand Java introduces the concept of interface-classes, in order to profit from the advantages of multiple inheritance.

The concepts around "generalisation" realise ideas directly taken out of reality like inheritance and abstraction. However its use in software development often mostly serves the purpose of code-reuse, not having to program repeatedly the same methods[16]. Code-reuse is a reasonable programming paradigm, but it existed long before the idea of object-orientation.

Together the mentioned concepts form the characteristic "generalisation":

- generalisation
- (single) inheritance
- multiple inheritance

[15] Hence the type of an instance of a class it the class itself or one of its subclasses (descendents).
[16] Even more important: not having to change the same code (methods) repeatedly at different positions, if corrections / adaptations are necessary.

2.3.3. Aggregation

As stated in paragraph 2.2.1., besides generalisation there is a second major structuring method: aggregation.

Apart from primitive data types[17], being used to construct objects, also objects can be used, to build bigger structures, i.e. higher order structures. New types of objects can be constructed out of existing (types of) objects, i.e.: existing classes serve as components of new classes. Thereby one or more classes become part of another class and thus objects become part of new objects. Nevertheless, apart from being used to build new structures, objects keep their independent status.

Example

A new type "Car" can be build upon existing types "Tyre" and "Driver", whereby a set of Tyre-objects and a Driver-object become part of a ForumlaCar-object, while still being individual entities of the system: a Driver-object can be a part of a Car-object and may at the same time be aggregated by a Family-object.

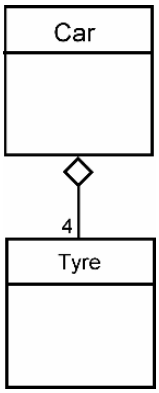
A Driver-object d_1 can be part of two different objects without the two objects having to deal with each other. On a software level the two aggregating objects hold a reference to d_1: as stated in section 2.1 every object can be unambiguously identified and referred to by its unique object-identity, i.e.: in fact the Driver-object d_1 is not a part of the Car-object, but the Car-object (as well as the Family-object) contains a pointer to the Driver-object.

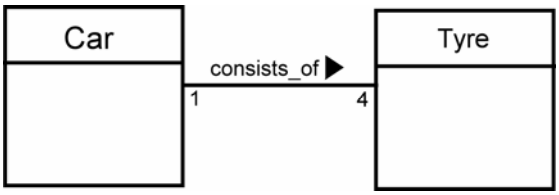
The described concept is called "aliasing", derived from the references being aliases for the referenced objects. Containing these aliases instead of whole objects implicates the following advantage: any change that is made to the state of a referenced object instantly affects all objects containing a reference to this object, known as "reference semantics". This might sound natural, but if the aggregating objects were to contain the whole object or a copy of it the state would have to be changed in every copy/aggregating object separately. The concept of copying objects and storing its data separately is called "value semantics"[18]: some aggregating object might have the purpose to save a state of an aggregated object before modifications are made and its values are therefore not meant to be changed if the aggregated object's state changes.

Picking up the example of paragraph 2.2.1.: 4 Tyre-objects are part of 1 Car-object, meaning that the Car-type aggregates the Tyre-type. The numbers become apparent as so-called "multiplicities": in our example "4:1". In programming languages these multiplicities are expressed by arrays or similar containers, comprising a fixed number of instances of a certain class; on the other hand multiplicities can directly be defined as attributes of type-relations in specification languages.

[17] Primitive data types are for instance: numbers, signs and strings, Boolean expressions true and false

[18] In contrast to "reference semantics" where every object only exists once unless explicitly cloned, no matter how often it is referred to or copied.






```
class Car {
  private Driver d;
  private Tyre[] tyres = new Tyre[4];
} //class
class Tyre {
} //class
```

Together the mentioned concepts form the characteristic “aggregation”:

- aggregation
- aliasing (reference semantic)
- multiplicities / arrays

2.3.4. Delegation & polymorphy

This characteristic addresses tasks and concepts that emerged from the concepts of generalisation and inheritance.

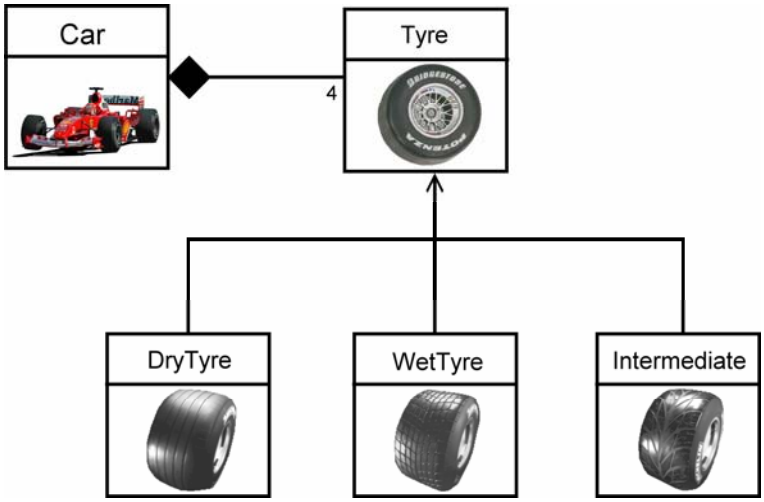
As stated in section 2.1 the type of an object defines the object’s entire behaviour and its interface, i.e. messages the object understands. Some of the messages that an object understands have been inherited from ancestor-classes. Conforming to the concept of code-reuse the respective methods are only included in that ancestor-class and not in the class that inherited the behaviour.
Thus, when getting a message[19] at run time an object passes the message on to its type, i.e. the class containing the object’s entire behaviour. If the respective method is not located in the class itself, it must have been inherited from an ancestor-class. The message is passed on to the superclass that the method was inherited from. In this manner the propagation of the message goes on until it eventually reaches the class, containing the method‘s definition (implementation). This process is called “delegation of responsibility” or simply (message-) dispatching. The more complex the relations between types can be (inheritance, multiple inheritance etc.) the more complicated the dispatching-process can get.

[19] That is listed in the object’s interface, i.e. the object understands the message.

Example

Resuming the example of the Car-object, in this case a formula-1 car, and its tyres, let us assume the Tyre-type has the subtypes DryTyre and WetTyre, both extending the behaviour of Tyre by some specific attributes. A car can by provided with either dry or wet tyres, in short, with four tyres.

On a software level a Car-object aggregates four instances of Tyre[20] (see paragraph 2.3.2.), i.e. any Car-object has four attributes with the type Tyre. As specified before these attributes are references (pointers) to instances of the class Tyre.



Since DryTyre is a subtype of Tyre, every DryTyre-object is (also) an instance of Tyre, having all the attributes and the entire behaviour of any Tyre-object, and as well: understanding all messages that Tyre-objects understand (see paragraph 2.3.2). In short: since a DryTyre-object is an instance of the class Tyre, it can replace and act as a Tyre-object. In this respect a DryTyre-object is called polymorphic.

The concept "polymorphy" is closely connected to the concept of "inheritance", since objects of subclasses inherit all attributes and methods of the superclass and can thus be polymorphic, i.e. replace and act as objects of the superclass.

However, in order to provide polymorphy an object-oriented system has to implement dynamic binding (late binding), where the definition/implementation of a method is not bound to the method's name before running time. That is, the name of a method may be bound to different implementations at running time, e.g. the same method meth() is called at two different times, i.e. the same message "meth" is sent, and is bound to different implementation in both cases, i.e. different methods are invoked.

Example

Let us assume that the type Tyre provides a "lifetime" method, returning the tyre's durability for a given average speed. The method "lifetime" may be refined/adjusted by the subtypes DryTyre and WetTyre, being more and less long-living than an average tyre, respectively.

Now a Car-object, provided with four DryTyre-objects, calls the "lifetime" method on its aggregated Tyre-objects, i.e. sends the message "lifetime". To return the correct durability-information the refined method of the DryTyre-class has to be called.

Later the tyres of the car have been changed and the Car-object is then aggregating four WetTye-objects. Calling the same "lifetime" method will now result in the execution of the respective method of the WetTyre-class.

[20] I.e. objects of the type Tyre, DryTyre, WetTyre or any other subtype of Tyre

```
class Tyre {
  private int compoundHardness;

  public int lifetime(int avgSpeed) {
     return compoundHardness;
  } //lifetime()
} //class
```

```
class WetTyre extends Tyre {
  private int numberOfRills;

  public int lifetime(int avgSpeed) {
     return compoundHardness * numberOfRills;
  } //lifetime()
} //class
```

The preceding example described the delegation of incoming messages to the type of the aggregated objects rather than to the type of respective attribute. Since a message is not bound to a method before at runtime, this is called “dynamic dispatching”.
The decision where to delegate the message “lifetime” to can not be made at compile-time, but at run time, when the type of a car’s “tyre”-attributes is known.

The subject dispatching imposes many tasks and problems on a programming language that are differently solved by different programming languages.
However dispatching is neither an object-oriented concept nor an advantage of object-oriented systems, but merely a task that has to be solved to realise polymorphy.

Together the mentioned concepts form the characteristic “delegation & polymorphy”:

- polymorphy
- delegation (of responsibility)
- (dynamic) dispatching

2.3.5. Reutilisation

This characteristic sums up those concepts of object-orientation that serve the purpose of code-reuse, i.e. the avoidance of repetition.

The last chapter addressed the delegation of methods. However the dispatching process may be ambiguous if the name of the message (method) does not clearly indicate its target.
Messages are identified by their name, implicating the problem of possible name clashes. While equally named methods in different classes do not produce any problems, equally named methods in the same class do: when a class receives a message, corresponding to these methods, it has to be decided, which method to initiate.

The multiple use of a name for different methods is called “overloading”. Some programming languages prohibit overloading in the first place to avoid problems and deceleration. However it is often reasonable and intuitive to name methods identical that are equally intended or similar. Overloading can be resolved if the methods differ in number or type of input or return parameters. However, since a message usually does not contain information about the number and type of the expected return values, the input parameters of equally named methods have to differ.

In contrast to equally named methods, the concept of genericity aims at methods, having the same purpose. The concept of genericity allows the construction of whole new types, using abstract parameters.
So-called parameterised types are built upon place holder types (α), called “formal (generic) parameters” that are actualised with an existing, so-called “actual parameter”, when instantiated. While replacing the place holder types of parameterised types with various concrete types, new types and methods are created without writing new code.

Example
The method "abs" (short form of "absolute value") operates on various types of numbers and if implemented generically any new number-type could use its implementation. The latter is realised once using a generic parameter α, resulting in a template method abs(number:α). By replacing the generic parameter by a concrete number type the method's implementation can be used for any kind of numbers, without rewriting the respective code. The resulting methods look like abs(number:int), abs(number:float), etc..

A generic class, whose formal parameter has not yet been replaced does not constitute a usable type, but a so-called "type template". Genericity is also known as "parameterised programming" and the concept of templates, originally introduced by the programming language C++.

Together the mentioned concepts form the characteristic "reutilisation":

- genericity
- overloading

In the field of programming languages this characteristic probably is experiencing the most innovations in the near future, as Java, the most popular object-oriented programming language of today, is just releasing a new version (Java v1.5) supporting genericity.

2.3.6. Other concepts / extensions

This sixth characteristic comprises all concepts of object-orientation that have not been considered before, since they do not fit in one of the other characteristics

.

These mainly include concepts that have been developed, or rather established in the recent past. Since the evolution of concepts goes on, more and more ideas and concepts arise that have been and will be added to object-oriented systems.
These concepts include class attributes and methods, concurrency, exceptions, assertions, and aspect orientation. However this list is not exhaustive.

The concept distinguishes from the other concepts that have been mentioned. On the on hand it could have been stated in the first characteristic, since it deals with attributes and methods, but on the other hand did not fit, since it is not a concept that was originally intended by the idea of object-orientation and does not refer to objects.
The idea of class attributes and methods is to have distinguished so-called "static" instance of a class that holds data and information of the class itself. Previously a class was not more than a generic clause for its objects, but this concept allows a class to have a state and a common behaviour as well. The term "static instance" refers to the fact that the class state is the same for all objects.

The other concepts are rather enlargements of a given system than modifications/extensions of its basic structure:
Concurrency adds parallel threads, i.e. simultaneous execution to a system. A system without concurrency has a straight execution progression. Concurrency calls this kind of execution a thread and allows the simultaneous execution of multiple thread, then so-called "parallel" threads. This concept is also known as "multitasking", e.g. an operating system is a concurrent system, since it offers the possibility of executing several programs at the same time (simultaneously).

The concept of “exceptions”, also known as “error handling” introduces a way processing unexpected situations. The latter, usually caused by unforeseen errors posed a major problem to running software systems, since these errors resulted in termination of execution, so-called dead locks, system crashes, or similar circumstances. Exceptions allow the so-called “catching” of errors that otherwise would have stopped the system’s execution, and a adequate reaction.

The other concepts, mentioned above (“assertions”, “aspect orientation”) have not been established as concepts of object-orientation. Both denote programming paradigms, but on the one hand they are not supported by major object-oriented systems of the present and on the other there is not yet a common approach, in order to specify their significance.

Together the mentioned concepts form the characteristic “extensions”:

- class attributes and methods
- concurrency
- exceptions

Since the other concepts have not been established yet, they are not listed above. It remains to be seen what kind of impact these concepts will have on object-orientation of tomorrow.

Definition 0 (Object-orientation)

The characteristics, identified in this section, constitute the features and qualities of today’s object-oriented systems and thus of object-orientation today.
Eventually these characteristics specify the requirements on object-oriented systems and thus will be used as a working definition of object-orientation within this paper:

- The term object-orientation can be described by the characteristics, identified in section 2.3.
- Object-oriented systems either embody these characteristics or are able to represent them.

Object-Oriented Transformation Systems have to be able to represent the characteristics, identified in section 2.3.. Since these characteristics serve as definition of object-orientation, they are listed in appendix A.

Chapter 3

Existing approaches

Chapter 2 determined the term "object-orientation" and thus the meaning of the term "object-oriented system". Hence the paper's goals G1 and G3 (see section 1.2) are now clarified in detail: Object-Oriented Transformation Systems have to be able to represent characteristics[21] of object-oriented systems and have to have an object-oriented structure themselves, terms that have been vague before.
Against the background of the characteristics of object-orientation existing approaches can now entirely be checked against the goals of this paper, identifying missing features, but also those qualities that are preferable for Object-Oriented Transformation Systems as well. As stated before existing and related approaches of this paper are the ones of J. Tenzer [Ten01] and D. Parnitzke [Par01] that have been combined by M. Große-Rhode (see [Gro01], chapter 6) and the one of B. Braatz, M. Klein and G. Schroeter. Both approaches developed within the DFG[22]-project IOSIP[23], where also Transformation Systems have been introduced as a starting-point of the project.
First of all this common basis will be introduced: Transformation Systems by M. Große-Rhode, followed by a short presentation of the existing approaches. The third section (3.3) then identifies which features have already been integrated/realised in these efforts and which are missing. Section 3.3 should yield a list of preferable ideas that are to be adopted by Object-Oriented Transformation Systems.

3.1. Transformation Systems

In 1998 M. Große-Rhode developed a reference model with the purpose of integrating the semantics of heterogeneous specification languages in a common semantical framework: Transformation Systems. Transformation Systems constitute the first step of the IOSIP-project that is concerned just with the integration of object-oriented software specification techniques.
Since it is the framework's purpose to be open to all kinds of specifications, it is kept as general as possible. Hence its definitions are easy to understand on the one hand, but are not specialised and have to be adjusted in particular cases.
Transformation System are two-layered structures, consisting of a static and a dynamic level, a data space and a control graph, respectively, both represented by directed graphs, so-called transition graphs[24]. The idea is to have a static data level, the data space, providing all possible system states, its nodes, and transformations leading from one state to another, its edges. In addition there is a dynamic level, the control graph, controlling dynamic aspects of the system and modelling a temporal ordering of the system's steps.

---

[21] The characteristics, identified in section 2.3.
[22] DFG = „**D**eutsche **F**orschungs**g**emeinschaft" (German Research Council)
[23] IOSIP = **I**ntegration of **o**bject-oriented software **s**pecification techniques and their application-specific extension for **i**ndustrial **p**roduction systems
[24] In contrast to "transition systems" a transition graph's states are also labelled.

The two transition graphs are connected over a transition graph morphism that labels the nodes of the control graph with system states, and its edges with transformation of the data space.

In short a system state of the data space is determined by concrete values for all existing data-attributes. Transformations between data stated are labelled by actions that change the system's state from state A to state B,

Example
If system state B differs from A in the value of attribute x, the action leading from A to B manipulates attribute x, i.e. an assignment-action (see definition 7).

The data space contains all possible data states of the system[25], i.e. system states, irrespective of whether they are ever reached or executed. The selection of reasonable data states and transformations from the data space and their temporal ordering are represented by the control graph. All so-called control state[26] and transitions between them are labelled with system states and transformation, respectively. However there may be system states (and transformations) that do not label any control state (transition), denoting unreachable data states that do not have to be considered any longer.
Eventually a transformation system is a labelled transition system, where both the states and the transitions are labelled. Via the distinction of the two levels the control flow information can be separated from the data state information. Correspondingly, first the data space of a system may be defined, and in a second step its control graph, defining how the atomic data transformations are composed to processes. The paths in the control graph then model the sequential execution of steps; branching indicates non-deterministic choice.

There is a so-called data space signature $D\Sigma = (\Sigma,A)$, consisting of an algebraic signature Σ and an action-signature A. It determines a Transformation System's data space, since every system state is given by an Σ-algebra and thus is said to induce the data space.
There may be (induced) data states that are not reasonable or not reachable. A data state X is called "reachable", if there is a sequence of transformations leading from an initialisation state to X, where initialisation states are especially marked starting-points of a Transformation System.
Transformations are given by actions and a tracking relation, where actions are induced by the data space signature. The tracking relation maps carrier sets of the first state onto carrier sets of the second state and operations to operations.

In every individual case the data space signature has to be adjusted to the kind of specifications that are integrated. Designing a suitable data space signature and defining relevant actions sets and tracking relations are tasks that have to be performed when applying Transformation Systems to a concrete software specification. Transformation Systems mainly differ in data space signature and action signature[27]. Various Transformation Systems integrating the same specification languages or systems though may look alike and use the same data space signature and action sets. Thus a once defined data space signature can be used again for integrating specifications of the same kind or may be even refined, based on the experience that has been made.

---

[25] A system state can be seen as a collection of data values, a value for each attribute.
[26] Nodes of the control graph are called control states.
[27] The action signature A actually is part of the data space signature (Σ,A). When both signature are denoted within one sentence the term "data space signature" rather refers to the algebraic signature Σ.

Beyond having to describe data states by algebras there is a modified version of Transformation Systems that allows defining data space signatures by other than algebraic signatures called "generic Transformation Systems". Generic Transformation Systems can be instantiated by defining the structure of data space signatures, the structure of corresponding system states, a set of actions, and a function Mod that maps a data space signature onto its respective data space (including the respective system states and transformations). The defined data space signatures and the function Mod have to form a concrete institution, in the sense of [GB84b] and [GB92] (for detailed description of these requirements, please refer to section 4.5., in which it will be proven that Object-Oriented Transformation Systems constitute an institution of generic Transformation Systems). A main reason for instantiating generic Transformation Systems is the existence of theoretical results like composition and development of generic Transformation Systems, proven for all instances (see section 4.6.).

3.2. Presentations

After this short introduction of Transformation Systems the related approaches of this paper will be presented. Like Object-Oriented Transformation Systems they are defined on the basis of Transformation Systems by M. Große-Rhode.

3.2.1. Integration of the UML by M. Große-Rhode

This paragraph presents the approach(es) of J. Tenzer [Ten01] and D. Parnitzke [Par01]. Since their approach addresses the integration of UML[28] Class diagrams into Transformation Systems or rather generic Transformation Systems. The two approaches have been combined by M. Große Rhode in an example of use (see chapter 6 of [Gro01]). Furthermore the instantiation is extended by the integration of UML state machines and UML sequence diagrams, being an example for integrating several specifications into a Transformation System.
The example applies generic Transformation Systems to the semantics of UML Software Specifications. More precisely it is the example's goal to integrate specifications of UML class diagrams, UML state machines and UML sequence diagrams. The result is a "UML-Transformation System", being a united specification of the software system that the diagrams describe from individual points of view.

The basic idea states that a class diagram describes the static structure of the system; an individual system state is than derived by putting together concrete values for all variables and system's attributes that appear in the class diagram[29]. Therefore the data space signature, being the origin of all system states of a Transformation System, is directly derived from the given class diagram. State Machine and Sequence Diagram on the other hand serve the construction of the control graph as both specify dynamic aspects of the system.
M. Große-Rhode introduces so-called class signatures as algebraic interpretation of UML classes[30]. The names of all data sorts and classes that occur in the UML class define the class signature's sorts, while its operations are given by the function names of the class diagram, i.e. all attribute- and method-names together with their functionality.
A whole UML class diagram is interpreted by a so-called Signature Diagram SD, becoming the data space signature of the UML-Transformation System. A Signature Diagram is a graph, containing a class signature for each class of the UML class diagram and a datatype signature

[28] UML stands for Unified Modeling Language, a formal specification language for software development and -engineering with the object-oriented approach. The current features of "UML v1.5" can be found in [UML03].
[29] I.e. a model of the data space signature.
[30] UML classes, in the sense of UML class diagrams.

for each appearing primitive datatype[31], the diagram's nodes. Besides there are three sets of edges, between the class signatures, representing associations, generalisation and inclusion, respectively. Inclusion edges only may appear from datatypes to classes (or other datatypes), but never between classes. Thus inclusion edges do not represent the object-oriented concept of aggregation. The latter is expressed through ordinary association edges named "belongs_to". Generalisation edges though do represent the concept of generalisation. Inheritance though is secured by corresponding constraints, but by interpreting the so-called "full-descriptors" of UML classes[32]. The consequential inheritance constraints however there are no inheritance constraints[33].
Eventually a system state of "UML Transformation Systems" is given by a static set of partial "datatype algebras", one per primitive datatype, and partial algebras, representing the system's objects and their actual state, a set of partial ΣC-Algebras per class C, where ΣC is the class signature to C. A so-called classification-function assigns each ΣC-Algebra to its class. Finally edges between objects, denoting associations among each other, complete a system state.
Between objects, denoted by partial algebras, there are so-called "subclass inclusion edges", corresponding to "generalisation edges" between classes, i.e. if there is a generalisation edge from class Cl1 to class Cl2, there will be a subclass inclusion edge from every Cl1-object to an object of Cl2. These subclass inclusion edges realise a polymorphic property of UML Transformation Systems: any object of class Cl1 can be casted to an object of superclass Cl2, denoted by the respective subclass inclusion edge.

3.2.2. Object-oriented Transformation Systems by Braatz and Klein

The other related approach of B. Braatz, M. Klein and G. Schroeter targets a more general, i.e. less UML-specific, but object-oriented way of integrating various specification techniques than "UML Transformation Systems". Matching the purpose of this paper, their adaptation of Transformation Systems aims at object-oriented specification languages, more precisely at object-oriented viewpoint specifications.

> Viewpoint specification techniques are means to handle heterogeneous aspects of complex (software) systems. Views are functions abstracting from the properties not described by the specification technique. These views are parameterized by the viewpoint specifications, because a viewpoint specification should in general be able to designate some substructure of models, which should be projected from concrete models into abstract models, while the rest of the models is forgotten (taken from [BKS04])

Though this paragraph refers to the paper [BKS04], some ideas that will be presented, in particular call stacks, can not be found in [BKS04] as they were never published. Nevertheless they will be introduced, since they directly affect the definition of Object-Oriented Transformation Systems.

Braatz, Klein and Schroeter introduce so-called object-oriented Transformation Systems as concrete mathematical models of object-oriented systems. In the following paragraph I am referring to this approach as "Object-oriented Transformation Systems by Braatz, Klein, & Schroeter".

---

[31] Primitive datatypes are datatypes like numbers, chars or Boolean values.
[32] The full-descriptor of a UML-class is somewhat the union of the class itself and all its superclasses. Thus the full-descriptor also includes methods, i.e. method-interfaces, that have been declared within superclasses and constitutes inheritance among UML-classes.
[33] I.e. subclasses do not have to inherit behaviour.

The latter and this paper, both target a framework for integrating all kinds of object-oriented specifications. Braatz, Klein & Schroeter though approach this goal questioning how object-orientation is represented internally, targeting a machine-oriented description of running systems that control object-oriented software, in short "running object-oriented software". To approach object-oriented systems from a machine oriented view they break down the universe of objects into sets of functions, attributes and methods accompanied by carrier sets for primitive data types[34]. Hence objects, the entities of the system, are flattened down to their components.
The so-called "object-oriented" data space signature DSig = (S, Fun, Attr, Meth, Constr) consists of merged information of all classes of the system[35], containing the names of appearing datatypes and classes ($DS \cup OS \subseteq S$) and the functionalities of functions (Fun), Attributes (Attr), Methods (Meth) and Constructors (Constr). While Fun is a set-family of static functions names, whose functionality only comprises primitive data types, Meth is the family of methods that were originally bound to certain classes. In object-oriented Transformation Systems by Braatz, Klein & Schroeter methods are no longer encapsulated in an object, but their first input-parameter denotes the object, the method belongs to. The family of constructors Constr contains special methods that create new objects, i.e. instances to certain classes. Finally the elements of Attr contain information about their type and the class, they belong to.

A system state of an object-oriented Transformation Systems by Braatz, Klein & Schroeter interprets every sort-symbol of S by either a static carrier set for data sorts out of DS, or by a static set of object names, for object sorts of OS, i.e. classes. Furthermore functions and attributes are interpreted by static partial data functions and partial attributation functions[36], respectively. In addition every system state features a so-called "call stack", a stack of methods and their environment. It tracks all invoked (and running) methods: when a method is invoked it is pushed onto the stack, taking the top position, displaying that it is in control of the running process. A method is popped off the stack when it returns the control to the method it was invoked from. In fact the active method knows where to return the control to by looking up the second top position of the stack. At the call stack every method is joined by its environment, containing the method's local variables and their assignments. For example every environment contains a variable "self", pointing at the object the active method belongs to, i.e. the active object.

The system's actions and thus the transformations between system states are mostly given by attribute-assignment actions, method calls, and corresponding return-actions. Attribute assignments can only affect the values of the active object[37], respecting the concept of data encapsulation where only the object's own methods can affect its state. As methods (and constructors) are not explicitly interpreted by system states, their semantics are implicitly given by the sequence of transformations that lies between the method's call- and return-action.

Braatz, Klein & Schroeter place the interaction between objects in the centre of their model: approaching from a more machine-oriented standpoint their adaptation consists mainly of message exchange between objects and modification of attributes. They mostly disregard the abstract concepts of object-orientation that point out the object itself, not its methods and attributes.

---

[34] Primitive datatypes are datatypes like numbers, chars or Boolean values.
[35] I am assuming the system to have a type-system and thus classes.
[36] Partial attributation functions can either be undefined or have a value.
[37] Denoted by the local variable „self".

3.3. Needs for changes

The two mentioned approaches "UML Transformation Systems" by M. Große Rhode and "object-oriented Transformation Systems by Braatz, Klein & Schroeter" are both based on Transformation Systems. The following section is going to work out which of the object-oriented concepts and characteristics, identified in section 2.3, have been realised in these approaches, which are missing or not adequately considered. Finally on the one hand it should get clear what properties of the related approaches hinder the achievement of this paper's goals. On the other hand though the qualities of the approaches should be identified that have be adopted within the upcoming definition of Object-Oriented Transformation Systems.
While considering the existing approaches, they are checked against the object-oriented characteristics that have been identified in section 2.3. In order to refer to these characteristics abbreviations like "C3" will be used, e.g. C3 refers to characteristic 3 "aggregation". The respective characteristics and concepts can be gathered from Appendix A.

3.3.1. Integration of the UML by M. Große-Rhode

The UML is one of "today's major object-oriented languages" that helped identifying the characteristics of object-orientation in section 2.3. UML Transformation Systems describe techniques of the UML, mainly UML class diagrams, regarding the (static) structure of the modelled system. The following paragraph is going to check if UML Transformation Systems can express the characteristics, identified in section 2.3.

"C1"
UML Transformation Systems feature a clear type-system, represented by the so-called class graph (classes & type-system). Furthermore the system states do not contain smaller units than objects that are represented by partial algebras object as atomic component). The latter involves that an object's attributes (and methods) are expressed, i.e. encapsulated, by the respective partial algebra. Though encapsulation is implemented, the object-oriented concepts "information hiding" and "access levels" are not realised: all data of all classes/objects is available to all other classes and objects. There are no visibility modifiers.

"C3"
The latter offers a possibility of representing the aggregation of classes by an association-edge "belongs_to".

"C2"
Generalisation is expressed by so-called "generalisation edges" between classes in the class-graph. Inheritance is assumed from the classes UML full-descriptors. Multiple-inheritance though is not supported: M. Große-Rhode assumes syntactical correctness of the addressed UML class diagrams and multiple inheritance is prohibited by the UML.

"C4"
Each generalisation edge between classes results in a unique edge between objects of the class and objects of the superclass, thus it is possible to cast objects of the subclass to objects of the superclass.

The latter forces the existence of a superclass' object even the described system state is not supposed to possess any objects of that superclass. However the idea of the object-oriented concept of polymorphy is not to yield new superclass-objects, but for objects to be able to act as superclass-objects, without the need for any superclass-object to exist.

UML Transformation Systems offer a way of constituting the semantics of UML class diagrams, state charts and sequence diagrams. In a first step all possible semantic interpretations of a class diagram are represented by system states of a UML Transformation System. Certainly that meets the intention of defining UML Transformation Systems as an example of use for Transformation Systems, but it limits this approach to the abilities of the UML.
While some concepts of object-orientation are not implemented simply because they are not supported by the UML, e.g. multiple inheritance, others can not be expressed by UML Transformation Systems, including access levels, data abstraction, static methods, overloading. The identified Characteristics C5 and C6 (see section 2.3) can not be expressed by UML Transformation Systems at all.
Eventually it also has to be annotated that the amount of new definitions introduced for UML Transformation Systems complicate a comprehension of the system, since they are not likely to the ones of existing object oriented systems.

3.3.2. Object-oriented Transformation Systems by Braatz, Klein & Schroeter

Both the approach of Braatz, Klein & Schroeter and this paper target a framework for integrating various kinds of object-oriented specifications. The main difference is that Braatz, Klein & Schroeter approach the goal from a machine oriented view, questioning how object-orientation is represented internally.

"C1"
A system state of object-oriented Transformation Systems by Braatz, Klein & Schroeter consist mainly of sets of attributes, methods and constructors[38]. Their first parameter indicates the object they are linked to. Although attributes of an object can only be modified of that object's methods, realising the concept "information hiding" the concept "data encapsulation" is not fully implemented: attributes and methods are not encapsulated since objects only exist as symbols within the carrier sets of so-called "object-sorts". Eventually it does not get clear how methods and attributes are connected to become associated units of data and behaviour (object as atomic component).
Hence objects are not the building blocks of the system as they reside at the same abstraction level as methods. Beyond even there are so-called object-sort symbols, there is no type-system, resulting in an object-oriented Transformation Systems by Braatz, Klein & Schroeter having no hierarchy to structure its data.

"C2"
Thus generalisation (and inheritance) can not be expressed by object-oriented Transformation Systems by Braatz, Klein & Schroeter.

"C3"
There is an (intuitive) implementation of aggregation as attributes can be references, i.e. aliases, to other objects (object names).

"C4"
The environment of methods in the call stack always includes a variable "self", denoting the active object of the system. Tracking the actual active object corresponds with the concept "delegation of responsibility", where a message is delegated until the receiver can understand and answer its request.

[38] Besides there are carrier sets for primitive data types and object sorts as well as a so-called call-stack of running methods and their environments (see section 3.2.2).

The modifications of the system's state are placed in the centre of consideration. The focus is set on attributes and methods that manipulate attributes, blurring the concept of objects as atomic units.
The result is a system with a transparent structure that is constituted mostly by the changing of the active object, i.e. method calls and –returns, and the manipulation of attributes.
This differs from this paper's goal of defining a framework that is object-oriented itself in order to facilitate the integration process of object-oriented systems.

However object-oriented Transformation Systems by Braatz, Klein & Schroeter offer two major advantages. First: a transparent structure of the system where every step of the described system can be assigned to a transformation. This leads to an comprehensible, but not object-oriented structure and thus is not of interested within this paper. Moreover a comprehensible structure can also be achieved by an object-oriented structure, as stated in section 1.2.
The second advantage is the way of expressing the delegation of responsibility from object to object. The latter does not cover the whole universe of object-oriented dispatching, but still is a quality that should be adopted by the framework of this paper.

Chapter 4

Object-Oriented Transformation Systems

After considering the two existing related approaches in the preceding chapter it became evident that both can not meet the goals of this paper (see section 1.2).
While UML Transformation Systems were primarily intended as example of use for Transformation Systems and are, on the one hand, not able of expressing several object-oriented characteristics (see section 2.3.), and on the other, specialised on one concrete object-oriented system (the UML), the goals of Braatz, Klein & Schroeter conform better with the ones of this paper, but their approach is not object-oriented itself and thus is not able of representing some of the fundamental principles of object-orientation, e.g. data encapsulation.

4.1. Revised object-oriented Transformation Systems

This section's purpose is the introduction of "Object-Oriented Transformation Systems", being able to express the characteristics, identified in section 2.3 (see appendix A).
As stated in the preceding section there are several goals that this Object-Oriented Transformation Systems and the equally named approach by Braatz, Klein & Schroeter have in common. Thus some parts in particular call stacks, will be incorporated, but have to be adjusted to fit in the intended object-oriented structure.
Other concepts have to be newly included

Example
The concept "access levels" has not been implemented either by UML Transformation Systems or by object-oriented Transformation Systems by Braatz, Klein & Schroeter. Since the variety of visibility modifiers (access levels) differs between different object-oriented systems, the set of access levels has to be generic and their significance has to be adjustable: some object-oriented system do not have any access levels, other only "public" and "private", today there are usually lots more like "protected", "friendly".

There are other concepts that also have to be newly defined for Object-Oriented Transformation Systems, for example the concept of "multiplicities", i.e. the possibility of creating arrays (lists) of objects, or class-attributes and -methods, known as so-called static parts of classes.

Finally section 5.2. will check the definition of Object-Oriented Transformation Systems against the concepts and characteristics that have been identified in section 2.3..

4.2. Signatures & Terms

In order to construct a Transformation System it is the first and principle task to set up a way to describe concrete system states. All possible system states together build a Transformation System's data space. In the first place we need to define the structure of a data space signature DSig, being the syntactical (and abstract) description of a system state. The data space signature must hold all information about the system's types, including its classes, their definitions and relations among each other. Thus the type-system has to be realised by the data space signature. The latter is accompanied by terms, denoting a syntactical way to represent data values. Terms can later be evaluated to concrete values.

Definition 1 (Object-oriented Data Space Signature)

DSig = (**DS, DFun, AL, Classes**), with

- Data sorts **DS**:

  a set of (static) sort symbols of primitive datatypes.

  As a minimum requirement we will require a data sort for Boolean expressions ("bool") and a data sort for natural numbers ("nat"), to define an existence predicate for objects and the multiplicities of aggregations, respectively (see definition 4).

  OTCsorts: bool, nat $\in$ DS

- Data functions **DFun** = $(DFun_{w \to s})_{w \in DS^*, s \in DS}$:

  a family of sets of function symbols, where a function symbol $f \in DFun_{w \to s}$ can also be written as $f: w \to s \in DFun$.

  In order to obtain terms (constants) for Boolean values and natural numbers, the existence of the following data functions is required:

  OTCterms: true, false $\in DFun_{\lambda \to bool}$[39]
  zero $\in DFun_{\lambda \to nat}$
  succ $\in DFun_{nat \to nat}$

- Access levels **AL**:

  a set of access level symbols, i.e. constants. Known examples are: "public", "protected", "package", "private", but also: "friendly", "private_protected".

- Classes **Classes**:

  a set of class symbols. Every class symbol C $\in$ Classes induces a Class Signature CSig (see definition 2).
  There has to be at least one class "Object", designated to be the root-class of all other classes (see Remark 1):

  OTCobject: Object $\in$ Classes

[39] λ denotes the empty word.

The sets DS and Classes, representing primitive data types and complex data structures, respectively, together constitute the types of the system. They are joined by so-called arrays to represent multiple relations between certain types (known as multiplicities).
The set of all system-types $\mathbf{Types_{DSig}}$ is then defined by

$$\mathbf{Types_{Dsig}} := (DS \cup Classes) \bullet \{ [] \}^* = (DS \cup Classes) \cup \{ t[] \mid t \in Types_{Dsig} \}$$ [40]

◇

Remark 1 (Object-oriented Data Space Signature)

The set of primitive data types **DS** corresponds to the common approach of predefining certain data types, including numbers, characters, strings, Boolean values and other types

Example

The programming language Java predisposes integers (int, short and long), floating point numbers (float, double and long double), characters (char), Boolean values (bool), representations of data-bytes (byte), the empty- and pseudo-data type void, and as a mixture between primitive data types and classes: strings (String). The various possibilities of a type, e.g. int, short and long, denote different saving alternatives, i.e. the data type int uses 4 bytes (= 32 bits) to store an integer-number, making it possible to denote 2^{32} different numbers. That is much[41] *more (less) than the type short (long) is able to represent using 2 (8) bytes of storage space.*
Special data-base-oriented programming- and specification languages even predispose concrete classes like dates, timestamps, sets, enumerations and so-called blobs, representing large amounts of data as "large binary objects".
The Unified Modeling Language (UML) does not strictly circumscribe the set of predefined data types: some primitive data type for numbers, characters, Boolean values and similar are assumed as well as structures like Enumerations, but also complex types like DateTime and similar can be taken for granted by the user if necessary for the system. On the other hand there is no list of predefined (primitive) data types.

Predefined primitive data types can be accessed and referred to from everywhere within the system and from any object, disregarding the concepts of "access levels" and "information hiding". The elements of their carrier sets (see definition 4) are not to be understood as (individual) entities of the system, but as simple values that are used to define the state of objects.

Example

"In contrast to objects of Classes, where different instances can have the same state, but still distinguish from each other as individual entities, there is only one integer number 5. Thus two attributes of two different objects assigned to the integer value 5 are equal to each other, Unlike two Tyre-objects of a Car(-object) that may have the exact same state, but still represent individual (and unequal) entities, namely tyres."

---

[40] "•" concatenates two strings/symbols: t • [] = t[]. Hence the set $Types_{Dsig}$ contains all sorts of (DS ∪ Classes) concatenated with any number of []-symbols, where t[] is called an array of type t. If t is an array itself t[] is called multi-dimensional array. More precisely, if t is a n-dimensional array of type t_0, t[] is a (n+1)-dimensional array of type t_0.
"M*" denotes the set of all words over M, where M is a set of symbols.
λ denotes the empty word.

[41] 2^{16} (= 65536) times

Nevertheless primitive data types (**DS**) do not contradict the general ideas of object-orientation as identified in section 2.3, in particular the concept of "data encapsulation", since these data types represent static data values and no interactive entities of the system. **DS** and **DFun** remain invariant throughout the system (see definitions of actions).

DS and **DFun** together can be understood as algebraic signature **DΣ**:

DΣ := (**DS**,**DFun**), where

DS defines the signature's sort symbols and DFun the signature's operation symbols (including its constant symbols). Vice versa an algebraic signature DΣ can hold all information that is needed to express DS and DFun. Thus the declaration of DΣ is equivalent to the declaration of DS and DFun individually.

Example

Since "bool" and "nat" are obligatory data sorts (see constraint OTCsorts) and "true", "false", "zero", and "succ" are obligatory data functions (see constraint OTCterms) the following signature is a sub-signature to DΣ of any Data Space Signature:

$DΣ_{min}$

sorts: *bool, nat*

opns: *true:* $\rightarrow$ *bool*

false: $\rightarrow$ *false*

zero: $\rightarrow$ *nat*

succ: nat $\rightarrow$ *nat*

As seen in the preceding example: the statement (succ $\in$ $DFun_{nat\rightarrow nat}$) can be equivalently written as (succ: nat $\rightarrow$ nat $\in$ DFun), whereas the statement (succ $\in$ DFun) refers to the existence of any function symbol "succ"[42]:

$$f \in DFun \;:\Leftrightarrow\; (\exists w \in DS^*, \exists s \in DS: f \in DFun_{w\rightarrow s}) \;\Leftrightarrow\; f \in \bigcup_{w\in DS^*, s\in DS} DFun_{w\rightarrow s}$$

The set of access levels **AL** can be scaled down or extended to fit the individual case. All definitions of object-oriented Transformation Systems that depend on these access levels are generic and adapt to the size and elements of **AL** (see definition 4).

To avoid any possible confusion I require the sets DS, DFun, AL, and Classes to be disjoint:

PCONdisjoint: $x \cap y = \varnothing \quad \forall x,y \in \{DS, DFun, AL, Classes\}$

Although not being formally mandatory this requirement eases the specification-work of this paper and is a usual procedure in object-oriented systems:

Example

The sets DS, DFun and AL are predefined sets in the programming language Java and thus disjoint from the outset. All their elements are called "keywords" and are not available to the user. Hence it is not possible to define any keyword as a class name, making DS, DFun, AL and Classes disjoint within all Java programs.

[42] with unknown typing

It is a consequence of convention PCONdisjoint that $Types_{DSig}$ is disjoint from DFun and AL:

$Types_{DSig} \cap DFun = \emptyset \quad \wedge$
$Types_{DSig} \cap AL = \emptyset$

If the possibility of confusion can be excluded $Types_{DSig}$ will be referred to simply as **Types**. The definition of DSig is completed by class signatures being **introduced** in the following paragraph.

Definition 2 (Class Signature)

Every class symbol $C \in Classes$ induces a class signature CSig, given by

CSig = (**Super**$_C$, **AC**$_C$, **StSig**$_C$, **BehSig**$_C$, **ClStSig**$_C$, **ClBehSig**$_C$), with

- Parents **Super**$_C \subseteq$ **Classes**:
  the set of Parent- / Superclasses of **C**. From the set **Super**$_C$ we directly derive the set **Ancestors**$_C$, recursively defined by

  Ancestors$_C$:= **Super**$_C \cup \{ c \mid c \in Super_P \wedge P \in$ **Ancestors**$_C \}$

  Obtaining a DAG-structure[43] is ensured by the following constraint:
  OTCdag: **Super**$_C \neq \emptyset \quad \forall C \in Classes \setminus \{Object\}$

  To prevent self-inheritance, it is required that
  OTCselfinh: **C** $\notin$ **Ancestors**$_C$.

- AccessClasses **AC**$_C$ = $(AC_{C,al})_{al \in (AL \cup \{Class\})}$:
  a family (of sets) of classes with $AC_{C,al} \subseteq Classes$. A class $Cl \in AC_{C,al}$ (more precisely the objects of class Cl) may access attributes and methods that are tagged with access level "al". The access classes **AC**$_C$ denote the interpretation of access level symbols AL by class C.

  The special case al = Class denotes the set $AC_{C,Class}$ of classes that can access the (whole) class **C** itself. "Class" can be understood as the access level of class C and thus $AC_{C,Class}$ is the set of classes that may access class C.

- State Signature **StSig**$_C$ = $(StSig_{C,al,type})_{al \in AL, type \in Types}$:
  a family (of sets) of (instance-) attribute symbols[44]. An attribute symbol $att \in StSig_{C,al,type}$, can also be written as al_type_att $\in$ **StSig**$_C$, being an instance attribute to sort type of class C.

  Inheritance is secured by the following constraint:
  OTCinher1: $\forall P \in$ **Super**$_C$: al_type_att $\in StSig_P \Rightarrow$
  $\exists al'$: al'_type_att $\in StSig_C \wedge AC_{C,al} \subseteq AC_{C,al'}$

  Attributes have to differ in name, i.e. there must not be two attributes with the same name, but different typing[45]:
  OTCtyping1: $\forall a1,a2 \in AL, \forall t1,t2 \in Types$: $(a1 \neq a2) \vee (t1 \neq t2) \Rightarrow$
  $StSig_{C,a1,t1} \cap StSig_{C,a2,t2} = \emptyset$[46]

---

[43] DAG = **D**irected **A**nticyclic **G**raph

[44] Attribute symbols of $StSig_C$ are also called "instance"-attribute symbols as they will be used to describe the state of the class's objects (instances). On the other hand attribute symbols of $ClStSig_C$, used to describe the state of class C itself, will be called "class"- or "static" attribute-symbols. The phrase "static" originates from the keyword static that is used by programming languages like Java and C++ to distinguish class-attributes from other attributes.

[45] The typing of an attribute is its type; the typing of a method are the types of the method's input parameters and return values.

[46] The secured property will be called "Unique Typing".

• Behaviour Signature $\mathbf{BehSig_C} = (BehSig_{C,al,args,type})_{al \in AL,\ args,type \in Types^*}$:
a family of method symbols, where a method symbol $meth \in BehSig_{C,al,args,type}$ can also be written as al_type_meth(args) $\in \mathbf{BehSig_C}$, being a method to the sort type of class **C**.

Inheritance is secured by the following constraint:
OTCinher2: $\forall P \in \mathbf{Super_C}$: al_ty_mt(args) $\in BehSig_P \Rightarrow$
$\exists al'$: al'_ty_mt(args) $\in BehSig_C \wedge AC_{C,al} \subseteq AC_{C,al'}$

Methods have to differ in name or in their arguments, i.e. there must not be two methods with same name and same argument-types but different typing:
OTCtyping2: $\forall a1,a2 \in AL,\ \forall ag1,ag2, t1,t2 \in Types^*$:
$BehSig_{C,a1,ag1,t1} \cap BehSig_{C,a2,ag2,t2} \neq \emptyset \Rightarrow$
$(ag1 \neq ag2) \vee ((a1 = a2) \wedge (t1 = t2))$

• Class State Signature $\mathbf{ClStSig_C} = (ClStSig_{C,al,type})_{al \in AL,type \in Types}$:
a family of class-attribute symbols, where an attribute symbol $catt \in ClStSig_{C,al,type}$ can also be written as al_type_catt $\in \mathbf{ClStSig_C}$.

Inheritance is secured by the following constraint:
OTCinher3: $\forall P \in \mathbf{Super_C}$: al_type_catt $\in ClStSig_P \Rightarrow$
$\exists al'$: al'_type_catt $\in ClStSig_C \wedge AC_{C,al} \subseteq AC_{C,al'}$

Unique Typing is secured by the following constraint:
OTCtyping3: $\forall a1,a2 \in AL,\ \forall t1,t2 \in Types$: $(a1 \neq a2) \vee (t1 \neq t2) \Rightarrow$
$ClStSig_{C,a1,t1} \cap ClStSig_{C,a2,t2} = \emptyset$

• Class Behaviour Signature $\mathbf{ClBehSig_C} = (ClBeh_{C,al,args,type})_{al \in AL,\ args,type \in Types^*}$
a family of class-method symbols. A method symbol $cm \in ClBehSig_{C,al,args,type}$, can also be written as al_type_cm (args) $\in \mathbf{ClBehSig_C}$.

There is a special static method "constructor" that is required to be part of every class C, i.e. being an element of $\mathbf{ClBehSig_C}$. It constructs and returns a new C-object. Hence the method's type is C:

OTCconstructor: constructor $\in \mathbf{ClBehSig_{C,C}}$

Inheritance is secured by the following constraint:
OTCinher4: $\forall P \in \mathbf{Super_C}$: al_ty_cmt(ag) $\in ClBehSig_P \Rightarrow$
$\exists al'$: al'_ty_cmt(ag) $\in ClBehSig_C \wedge AC_{C,al} \subseteq AC_{C,al'}$

Unique Typing is secured by the following constraint:
OTCtyping4: $\forall a1,a2 \in AL,\ \forall ag1,ag2, t1,t2 \in Types^*$:
$ClBehSig_{C,a1,ag1,t1} \cap ClBehSig_{C,a2,ag2,t2} \neq \emptyset \Rightarrow$
$(ag1 \neq ag2) \vee ((a1 = a2) \wedge (t1 = t2))$

◇

Remark 2 (Class Signature)

A class signature describes a class's structure and content and its location within the system. A class signature distinguishes between external and internal definitions, its parents $Super_C$ and access classification AC_C from the outside and the internal description of the class's state and behaviour, respectively.

$Super_C$ and AC_C classify/group all other classes from C's point of view:
The set of Superclasses $\mathbf{Super_C}$ lists all generalisations of C, i.e. all classes that C inherits from (see constraints OTCinher1-4). When inheriting from its Superclasses C may change the access levels of inherited attributes and methods. In order to realise polymorphy, however, C must not reduce visibility[47] of an inherited method.

Example
C inherits method oldal_type_meth(args) from Superclass P, changing the method's access level from oldal to newal. By virtue of the fact that subclass C should promise to do whatever P does, it must not reduce accessibility of meth in order to realise polymorphy. Thus C's access class of the former access level oldal has to be a subset of the new access level newal's access class:

$$AC_{C,oldal} \subseteq AC_{C,newal}$$

At this point I want to remark that Object-Oriented Transformation Systems do not allow covariant (return-)types within inherited methods, i.e. subtypes of the original (return-)types, a feature offered by the programming language Eiffel, to give an example. However covariant (return-)types are prohibited in programming languages like Java and C++ and not supported in specification languages like the UML and would have to be realised by explicit casting of return-values.[48]

As C inherits the behaviour of all its Superclasses, objects of C may replace objects of C's Superclasses[49]: this property will be described by Instance Sets (see definition 4).

Constraints OTCdag and OTCselfinh prevent self-inheritance. (see paragraph 2.3.2).[50]
As a consequence all classes and their Superclass-relation(s) describe a (hierarchical) DAG-structure, featuring the obligatory class "Object" as root.[51] Hence "Object" is Ancestor to all other classes. Vice versa, the objects of all classes, i.e. all objects of the system, are instances of "Object" (see definition 4).
Multiple inheritance implicates the possibility of repeated inheritance, i.e. a class may inherit the same behaviour from an ancestor-class over different inheritance-paths. However, since class signatures and their respective behaviour signatures only inherit method-symbols and not method-bodies this does not pose a problem to Object-Oriented Transformation Systems: these method-symbols are merged, i.e. methods that have been inherited twice will only appear once. Hence, when translating a concrete object-oriented system into Object-Oriented Transformation Systems, the behaviour of one method may be adopted, the other overwritten.

---

47 i.e. accessibility

48 Needless to mention that Object-Oriented Transformation Systems do neither support contravariant argument-types nor covariant.

49 Complying with the object-oriented concept "polymorphy" (C4).

50 Conforming with major object-oriented systems of the present: even programming languages like Eiffel that do not restrict generalisation and inheritance prevent self-inheritance.

51 As the class signature of an Object-Oriented Transformation System defines its type-system, the type-system also describes a DAG-structure with the root-class "Object".

The set $\mathbf{Ancestors_C}$ lists all ancestor-(super-)classes of C. As there is no repeated inheritance $\mathbf{Ancestors_C}$ implicitly denotes an inheritance hierarchy for C and constitutes its position in the system's class graph, given by the type-system. The set is later used to realise polymorphy as C-objects are instances of any ancestor-class of C.

$\text{Object} \in \text{Ancestor}_C \qquad \forall C \in \text{Classes}$

Access classes $\mathbf{AC_C}$ denote C's interpretation of the system's access levels AL. All class's attributes and methods are tagged with access levels to limit their access to a certain group of classes.

Example

The method $meth_1 \in BehSig_{C,protected,args,type}$ *is tagged with the access level "protected", written protected_type_*meh_1*(args).* $meth_1$ *may only be accessed by classes Cl that are elements of C's "protected"-access class, i.e.* $Cl \in AC_{C,protected}$.
On the other hand, every class $Cl \in AC_{C,protected}$*, or rather its objects, may access any of C's methods (and attributes) that are tagged with the access level "protected".*

To adjust the common understanding of certain access levels I postulate the following convention for within this paper: the access level "private" restricts access to the class itself, while the access level "public" approves access to all classes of the system. Furthermore class C itself should be part of all access classes:

PCONlevels: $\text{private}, \text{public} \in AL$
PCONprivate: $AC_{C,\text{private}} = \{C\} \qquad \forall C \in \text{Classes}$
PCONpublic: $AC_{C,\text{public}} = \text{Classes} \qquad \forall C \in \text{Classes}$
PCONalself: $C \in AC_{C,al} \qquad \forall al \in (AL \cup \{\text{Class}\})$

This convention is not mandatory and concrete Object-Oriented Transformation Systems may violate it. Actually there is no need to define the access levels "private" and "public" at all.

An Object-Oriented Transformation System can go without any access levels, i.e. AL = {_}, where "_" is the "boundless" access level, treated like the access level "public" (see above), that scribally can be omitted: $AC_{C,\_} = AC_C = \text{Classes} \quad \forall C \in \text{Classes}$.

Attributes of $\mathbf{StSig_C}$ also define access levels. When an attribute is tagged with an access level other than "private" this contradicts the original idea of object-orientation, where the attributes of objects are encapsulated and hidden from the outside:
To conform to the object-oriented concept of "information hiding" (C1) the access levels of all attributes have to be set to "private". Hence attributes that have to be externally manipulated should be made accessible through so-called "getter"- and "setter"-methods. Public "getter"- and "setter"-methods of a private attribute implicitly make the attribute public itself without violating the concept of "information hiding" and actually following the concept "data abstraction" (C1)[52].

Nevertheless it is possible to define attributes with access levels different to "private" to facilitate the comprehension of Object-Oriented Transformation Systems, following the manner customary: all major object-oriented systems of today, including programming languages like Java and C++ or specification languages like the UML, allow the definition of attributes with access levels other than private, i.e. the external manipulation of attributes.

[52] Making all attributes "private" and enabling access through "getter"- and "setter"-methods is colloquially known as part of a "good programming style".

The access levels of attributes and methods have to be distinct. Beyond that attributes are required to differ in name while methods have to be distinguishable by name and argument-list, i.e. there must not be two equally named methods with identical argument-lists (that only differ in their return-types), secured by constraints OTCtyping1-4.
Hence methods can be distinguished by name and argument-list and thus their access level and return-value-list can be omitted. For the same reason an attribute's access level and type can be omitted:

$att \in StSig_{C,al,type} \Rightarrow att := type\_att := al\_type\_att$
$meth \in BehSig_{C,al,args,type} \Rightarrow meth(args) := type\_meth(args) := al\_type\_meth(args)$
$catt \in ClStSig_{C,al,type} \Rightarrow catt := type\_catt := al\_type\_catt$
$cmeth \in ClBehSig_{C,al,args,type} \Rightarrow cmeth(args) := type\_cmeth(args) := al\_type\_cmeth(args)$

For the scope of this paper I want to go even further when requiring that a class's state signature and its class state signature are disjoint, i.e. there are no attributes that are instance attributes and class attributes at the same time. Analogously I require instance methods and class methods to differ in name or argument-list[53]:

PCONatt: $\forall a1,a2 \in AL, \forall t1,t2 \in Types: StSig_{C,a1,t1} \cap ClStSig_{C,a2,t2} = \emptyset$
PCONmeth: $\forall a1,a2 \in AL, \forall args1,args2, t1,t2 \in Types^*:$
$BehSig_{C,a1,args1,t1} \cap ClBehSig_{C,a2,args2,t2} \neq \emptyset \Rightarrow (args1 \neq args2)$

Since a class's state signature may not have two equally named attributes the following notation can be defined unambiguously:

$att \in StSig_{C,acle,type} \Rightarrow \mathbf{al(att)} := acle$

al can be extended to functions $\mathbf{al_{C,St}}$ and $\mathbf{al_{C,ClSt}}$ that map attributes onto their access levels:

$al_{C,St}: StSig_C \rightarrow AL$
$al_{C,St}(att) = acle :\Leftrightarrow att \in StSig_{C,acle}$

$al_{C,ClSt}: ClStSig_C \rightarrow AL$
$al_{C,ClSt}(att) = acle :\Leftrightarrow att \in ClStSig_{C,acle}$

$StSig_{C,acle}$, $ClStSig_{C,acle}$ are defined as follows: the following declarations are inducted to ease/shorten the specification-work at some points of this paper:

$att \in StSig_{C,type} :\Leftrightarrow (\exists al \in AL: att \in StSig_{C,al,type}) \Leftrightarrow att \in \bigcup_{al \in AL} StSig_{C,al,type}$
$att \in StSig_{C,al} :\Leftrightarrow (\exists type \in Types: att \in StSig_{C,al,type}) \Leftrightarrow att \in \bigcup_{type \in Types} StSig_{C,al,type}$

The preceding declarations do not lead to any ambiguity as what is meant with $\mathbf{StSig_{C,x}}$ since the sets AL and Types are disjoint (see convention PCONdisjoint).

[53] Conventions PCONatt and PCONmeth are satisfied by programming languages like Java and C++ and specification languages like the UML. All mentioned object-oriented systems merge the definition of instance- and class-attributes (methods) and thus implicitly prohibit equally named instance- and class-attributes (methods that match in name and argument-list).

Analogous declarations apply to $\mathbf{ClStSig_C}$:

$catt \in ClStSig_{C,type} :\Leftrightarrow (\exists al \in AL: catt \in ClStSig_{C,al,type}) \Leftrightarrow att \in \bigcup_{al \in AL} ClStSig_{C,al,type}$

$catt \in ClStSig_{C,al} :\Leftrightarrow (\exists type \in Types: att \in ClStSig_{C,al,type}) \Leftrightarrow att \in \bigcup_{type \in Types} ClStSig_{C,al,type}$

As before $\mathbf{ClStSig_{C,x}}$ is an unambiguous notation because the sets AL and Types are disjoint (see convention PCONdisjoint).

There are similar declarations for a class's behaviour signature $\mathbf{BehSig_C}$ and its methods:

$meth \in BehSig_{C,type} :\Leftrightarrow (\exists al \in AL, \exists args \in Types^*: meth \in BehSig_{C,al,args,type}) \Leftrightarrow meth \in \bigcup_{al \in AL, args \in Types^*} BehSig_{C,al,args,type}$

Analogous declarations apply to $\mathbf{ClBehSig_C}$:

$cmeth \in ClBehSig_{C,type} :\Leftrightarrow (\exists al \in AL, \exists args \in Types^*: cmeth \in ClBehSig_{C,al,args,type}) \Leftrightarrow cmeth \in \bigcup_{al \in AL, args \in Types^*} ClBehSig_{C,al,args,type}$

To denote the empty type-list $\lambda \in Types^*$ it is usual to write "**void**", the so-called "empty type". Within this paper I am going to use "void" as a synonym for the empty type-list λ:

PCONvoid1: $void \notin Types$
PCONvoid2: $void := \lambda \in Types^*$

The former convention PCONvoid1 is made to protect convention PCONdisjoint and its consequence $Types^* \cap AL = \varnothing$.

As "void" is a keyword of today's major object-oriented programming languages like Java and C++ and specification languages like the UML and therefore intuitively understood by persons familiar with these systems, it can facilitate the declaration of a method's typing, i.e. the types of a method's input parameters and return values.

Example
A method $meth_1 \in BehSig_{C,al,type,void}$, getting no arguments (input parameters), may be written as $al\_type\_meth_1()$ or $al\_type\_meth_1(void)$, while a method $meth_2 \in BehSig_{C,al,void,args}$ with no return values may be written as $al\_void\_meth_2(args)$.

The empty argument list usually is not replaced by void, while the use of void as special return type, i.e. replacement of the empty return value list, is the normal case[54].

Though there are no major object-oriented systems that do support multiple (more than one) return values, Object-Oriented Transformation Systems do offer this possibility. The representation of object-oriented systems like Java and the UML would get by with the following definition of a class's behaviour signature:

$BehSig_C = (BehSig_{C,al,args,type})_{al \in AL, args \in Types^*, type \in (Types \cup \{void\})}$

[54] In programming languages like Java and C++.

Class attributes catt $\in$ **$ClStSig_C$** are used to describe the state of class C itself within a system state (see definition 4). To persons familiar with object-oriented programming languages "class"-attributes may be better known as "static" attributes, as referred to by Java, C++ and other languages. Though class-attributes are not literally static, the keyword "static" refers to these attributes not having different values for each instance of C: class-attributes are directly bound to class C itself and there is only one value for a class-attribute catt, no matter how many instances of C there are.
Within a system state all class-attributes are embraced by a so-called "static instance of C" (see definition 4).
Analogically, class methods cmeth $\in$ **$ClBehSig_C$** may be better known as a class's "static methods". Class methods do not describe the behaviour of object of C, but the behaviour of class C itself. Directly, class methods can only access class attributes.[55]
The obligatory class method "constructor" is not bound to a certain argument-list, but only to a fixed return type, i.e. the class C itself. There can be more than one "constructor"-method, having various argument-lists. However every class has to define at least one "constructor"-method, having the following default-typing:
public_C_constructor() $\in$ $ClBehSig_C$

An object-oriented data space signature is accompanied by a term family $T_{DSig}(X)$, where X is a set of variables: terms can be used as syntactical representation of data values and later be evaluated to concrete (semantic) values.
To be more precisely: the sorts (DS $\cup$ Classes) of an object-oriented data space signature will be interpreted by carrier sets. Terms represent elements of these semantic sets using syntactical symbols and can later be evaluated to concrete values.

Definition 3 (Data Space Signature Terms)

The term family $T_{DSig}(X)$ w.r.t. an object-oriented data space signature DSig and a set of variables X is defined by

$T_{DSig}(X) = (T_{DSig}(X)_s)_{s \in Types}$, where
$T_{DSig}(X)_s = X_s \cup$
$\{ f(t_1, \ldots, t_n) \mid (f: s_1 \ldots s_n \rightarrow s) \in DFun \wedge \forall i \in [1..n]: t_i \in T_{DSig}(X)_{si} \} \cup$
$\{ t_0.att \mid att \in St_{C,s} \wedge t_0 \in T_{DSig}(X)_C \wedge C \in Classes \} \cup$
$\{ C.att \mid att \in ClSt_{C,s} \wedge C \in Classes \} \cup$
$\{ t_0[t_n] \mid t_0 \in T_{DSig}(X)_{s[]} \wedge t_n \in T_{DSig}(X)_{nat} \}$
$\{ \#(t_0) \mid t_0 \in T_{DSig}(X)_{type[]} \wedge s = nat \wedge type \in Types \} \cup$

A function "vtype" attaches a sort to each variable, implying a variable-family $(X_s)_{s \in Types}$:

vtype: $X \rightarrow Types$
$\Rightarrow$ $X_s := \{ x \mid x \in X \wedge vtype(x) = s \}$ $\forall s \in Types$

The set of variables X describes local variables: active methods declare local variables to save temporary data (values) that are used to accomplish the method's tasks. These include the delivered input parameters and the return values of methods that have been invoked and yet finished their execution. In addition the variable "self" is always included in X, marking the actual active object:

OTCself: self $\in$ X

[55] In contrast to instance attributes of $StSig_C$. Vice versa instance methods can access the attributes of their instance as well as all class attributes.

◇

Remark 3 (Data Space Signature Terms)

Apart from the variable "self" X denotes a dynamic set of variables: system states or more precisely their active method(s) interpret X as a set of variables that is bound to the method's environment (see Call Stacks of Definition 4). Thus X differs between system states and self is the only variable, always contained in X.

The induced algebraic signature $D\Sigma = (DS, DFun)$ (see Remark 1) can now be supplemented by a family of signature-terms $\mathbf{T_{D\Sigma}}$:

$\mathbf{T_{D\Sigma}} = (T_{D\Sigma,s})_{s \in DS}$, where

$T_{D\Sigma,s} = \{ f(t_1, \ldots, t_n) \mid (f: s_1 \ldots s_n \rightarrow s) \in DFun \ \wedge \ \forall i \in [1..n]: t_i \in T_{D\Sigma}(X)_{s_i} \}$

It is quite evident that

$T_{D\Sigma,s} \subseteq T_{DSig}(X)_s \quad \forall s \in DS$

$\mathbf{T_{D\Sigma}}$ denotes the family of (ground-)terms of $D\Sigma$ that later will be used for accessing the values (elements) of primitive data sorts.

Example

Picking up the example of remark 1, the following family of Terms $T_{D\Sigma_{min}}$ denotes terms that are available in all Object-Oriented Transformation Systems:

$T_{D\Sigma_{min}} = (T_{D\Sigma_{min},s})_{s \in DS}$, *where*

$T_{D\Sigma_{min},bool} = \{ true, false \}$

$T_{D\Sigma_{min},nat} = \{ zero \} \cup \{ succ(t) \mid t \in T_{D\Sigma_{min},nat} \}$

4.3. Data Space & System states

Data space signature DSig and terms T_{DSig} constitute an abstract description of the system[56]. A concrete system state is then given by a concrete model of the data space signature. In order to instantiate DSig a carrier set has to be assigned to each sort: primitive data sorts $s \in DS$ are interpreted by simple mathematical carrier sets. The carrier sets to class symbols $C \in$ Classes are called object sets, whose elements (objects) interpret the respective class signature CSig by assigning concrete values to the attributes of $StSig_C$.

Class attributes (also known as "static" attributes of a class), describing a class's state, are encapsulated within a so-called class-instance (also known as static instance): one class-instance for each class $C \in$ Classes.

A system state is completed by a method-stack: methods are pushed onto the stack, when invoked, and popped off the stack, when finishing execution. The call stack tracks the active and all waiting methods of the system, together with their actual state. A method's state is given by its environment, i.e. its local variables and their values.

[56] The Object-Oriented Transformation System or rather the system, it represents.

Definition 4 (Data State / System State)

$\mathbf{A_{DSig}} = ((\mathbf{A_s})_{s \in (DS \cup Classes)}, (\mathbf{f_A})_{f \in DFun}, (\mathbf{static_{A,C}})_{C \in Classes}, \mathbf{stack_A})$

- Carrier sets $(\mathbf{A_s})_{s \in (DS \cup Classes)}$:

  a carrier set $\mathbf{A_s}$ for each sort $s \in (DS \cup Classes)$, where we distinguish 2 cases:

  1. case ($s \in DS$)

  $\mathbf{A_s}$ is the carrier set to the (primitive) data sort symbol s.

  In order to define an existence-predicate for objects, the obligatory data type bool $\in$ DS (see OTCsorts) has to be interpreted by the carrier set {T,F}[57], i.e. the set of Boolean values $\mathbb{B}$.
  In order to describe ordered Arrays over data types, the obligatory data sort nat $\in$ DS has to be interpreted by natural numbers $\mathbb{N}$ or a finite subset of $\mathbb{N}$[58]:

  OTCabool: $A_{bool} = \{T, F\} \ \wedge \ A_{nat} = \mathbb{N}$

  2. case ($s \in Classes$)

  A_s is the object set to the class symbol $s \in Classes$.
  We directly derive the family of **Instance Sets $(\mathbf{AI_s})_{s \in Classes}$**, defined by

  $\mathbf{AI_s} = A_s \cup \{ obj \mid obj \in A_P \wedge s \in Ancestors_P \} \cup \{ null \}$

  The latter make possible a declaration of semantical (value) domains for all system-types in Types:

  $\mathbf{Domains_{A_{DSig}}} := \{A_s \mid s \in DS\} \cup \{AI_C \mid C \in Classes\} \cup \{D^* \mid D \in \mathbf{Domains_{A_{DSig}}}\}$

  A function **VDom** assigns a value-domain to every system's type $\in$ Types:

  VDom: Types $\rightarrow$ $Domains_{A_{DSig}}$

  $$VDom(type) = \begin{cases} A_{type} & \text{, for type } \in \text{ DS} \\ AI_{type} & \text{, for type } \in \text{ Classes} \\ VDom(t)^* \cup \{null\} & \text{, for type = t[]} \end{cases}$$

  Every object $\mathbf{obj \in A_s}$ induces a family of partial attribute values that will be referred to as the object's (concrete/actual) state, given by:

  $(\mathbf{obj}.att: \rightarrow VDom(type))_{type\_att \in (StSig_C \cup \{bool\_exists\})}$,

  an partial attribute value obj.att for each attribute symbol $att \in St_C$.

  There is an obligatory attribute "exists", denoted by the partial attribute value obj.exists: $\rightarrow$ bool, where bool $\in$ DS. obj.exists is set to true in the beginning of an object's life cycle and set to false when the object dies.

  OTCexists: $obj.exists \in A_{bool} \quad \forall obj \in A_C, \forall C \in Classes$

---

[57] bool may also be interpreted by an isomorphic set like {1,0}. However within this paper the values of A_{bool} will be referred to as T and F.

[58] Computers are finite ☺.

In order to realise object identities, making each object individual, it is required that all object-sets are disjoint[59]:

OTCidentities: $A_{s1} \cap A_{s2} = \varnothing \quad \forall s1,s2 \in Classes$

- Functions $(\mathbf{f_{\underline{A}}})_{f \in DFun}$:
  a partial data function $f_A: A_w \to A_s$ for each function symbol $(f: w \to s) \in DFun$.
- Class-states $(\mathbf{static_{A,C}})_{C \in Classes}$:
  $\mathbf{static_{A,C}} = (\ \mathbf{static_{A,C}}.att: \to VDom(type)\ )_{type\_att \in ClStSig_C}$,
  a so-called class-instance for each class $C \in Classes$, being a family of partial attribute values;
  an partial attribute value $static_{A,C}.att$ for each attribute symbol $att \in ClStSig_C$.
- Call Stack $\mathbf{stack_A} \in (\mathbf{Meth} \times \mathbf{VType_A} \times \mathbf{VValue_A})^*$:
  a call stack of methods that have been invoked, but not yet finished execution. Meth is the set of all methods of all classes, i.e. all methods of the system:

$$\mathbf{Meth} = \bigcup_{C \in Classes} (BehSig_C \uplus ClBehSig_C)$$
$$= \{\ meth \mid meth \in (BehSig_C \uplus ClBehSig_C) \wedge C \in Classes\ \}$$
$$\mathbf{VType_A} = \{\ vtype \mid vtype: X \to Types\ \}$$
$$\mathbf{VValue_A} = \{\ vvalue \mid vvalue: X \to \mathbf{A_{DSig}}\ \}$$

An element of the call stack represents a method $meth \in \mathbf{Meth}$. The latter is joint by its actual environment (vtype, vvalue), where

$vtype: X \to Types$ and
$vvalue: X \to A_{DSig}$[60].

vtype implies the sort-specific functions $(vvalue_s)_{s \in Types}$:

$vvalue_s: X_s \to VDom(s)$
$vvalue_s(x) = vvalue(x) \quad \forall\ x \in X_s$

The environment of a method (vtype, vvalue) contains the method's local variables X, their types and values, denoted by vtype and vvalue, respectively. It implies a partial term evaluation function $eval = eval(vtype,vvalue)$[61]:

$\mathbf{eval}: T_{DSig} \to A_{DSig}$
$eval = (eval_s)_{s \in Types}$, where

$eval_s$ is recursively defined w.r.t a given function $vvalue: X \to A_{DSig}$ and w.r.t partial functions $f_A: A_{s1} \times \ldots \times A_{sn} \to A_s$ for $(f: s1 \ldots sn \to s) \in DFun$,
partial attribute values $obj.att: \to VDom(s)$ for $s\_att \in StSig_C$, $obj \in A_C$, and
partial class-attribute values $static_{A,C}.att: \to VDom(s)$ for $s\_att \in ClStSig_C$:

---

[59] Disjoint in pairs

[60] An element of A_{DSig}, comprising the carrier sets $(A_s)_{s \in (DS \cup Classes)}$ and lists over these carrier sets, i.e. all elements of $Domains_A$ (see Remark 4).

[61] The definition of eval(vtype,vvalue) depends on the environment (vtype,vvalue) and is linked to the definition of the two functions vtype and vvalue. If the meant environment is unambiguously clear, the evaluation function will simply be referred to as eval. Below the denotations $eval^2$ and eval' will implicitly mean signify the evaluation function $eval(vtype^2, vvalue^2)$ and eval(vtype',vvalue'), respectively.

$\forall s \in$ Types:

$eval_s$: $T_{DSig}(X)_s \rightarrow VDom(s)$

$$eval_s(t) = \begin{cases} vvalue_s(t) & , \text{for } t \in X_s \\ f_A\left(eval_{s1}(t_1), \ ..., \ eval_{sn}(t_n)\right) & , \text{for } t = f_A(t1, \ ..., \ tn) \\ \left(eval_C(t_0)\right).att & , \text{for } t = t_0.att \ \wedge \ eval_C(t_0) \neq null \\ null & , \text{for } t = t_0.att \ \wedge \ eval_C(t_0) = null \\ static_{A,C}.att & , \text{for } t = C.att \\ eva[]_s\left(eval_{s*}(t_0), \ eval_{nat}(t_n)\right) & , \text{for } t = t_0[t_n] \wedge eval_C(t_0) \neq null \\ null & , \text{for } t = t_0[t_n] \wedge eval_C(t_0) = null \\ eva\#_{s'}\left(eval_{s'}(t_0)\right) & , \text{for } t = \#(t_0) \wedge t_0 \in T_{DSig}(X)_{s'} \end{cases}$$

The last case remains irrelevant unless s = nat.
The Array-functions eva[] and eva# are given by

$eva[]_{type}$: $VDom(type)^* \times A_{nat} \rightarrow VDom(type)$

$$eva[]_{type}(a,n) = \begin{cases} e & , \text{for } a = eL \ \wedge \ n = 0 \\ eva[]_{type}(L,n-1) & , \text{for } a = eL \ \wedge \ n > 0 \\ null & , \text{for } a = \lambda \end{cases}$$

$eva\#_{type}$: $VDom(type)^* \rightarrow A_{nat}$

$$eva\#_{type}(a) = \begin{cases} 0 & , \text{for } a = \lambda \\ eva\#_{type}(L) + 1 & , \text{for } a = eL \end{cases}$$

where $e \in VDom(type)$, $L \in VDom(type)^*$, $n \in A_{nat}$.

Finally, there is a constraint OTCterms that specifies how the obligatory terms (see OTCterms) have to be evaluated:

OTCeval: $eval_{bool}(\text{ true }) = T$
$eval_{bool}(\text{ false }) = F$
$eval_{nat}(\text{ zero }) = 0$
$eval_{nat}(\text{ succ(t) }) = eval_{nat}(\text{ t }) + 1 \qquad \forall t \in T_{DSig}(X)_{nat}$

◇

Remark 4 (Data State / System State)

An Instance set $\mathbf{AI_C}$ to class C ∈ Classes contains all C-objects, i.e. carrier set $A_C \subseteq AI_C$, and all objects of subtypes of C, realising the concept of polymorphy (C2). The original idea of polymorphy can be expressed by the following statement: an object obj is an instance of its type. Since the object's type C contains all methods that any supertype contains (constraint OTC5), the object obj understand all messages, i.e. method calls, that an object of a supertype understands and thus can take its place. Hence obj is an instance of any ancestor-type of C. This approach manifests in the original definition of Instance sets $(AI_C)_{C \in Classes}$ that is equivalent to the one given in definition 2:

All Instance Sets $\mathbf{AI_s}$ are also defined by

$$(A_s \subseteq AI_s) \wedge (A_s \subseteq AI_{sc} \quad \forall sc \in Ancestors_s) \wedge (null \in AI_s) \qquad \forall s \in Classes$$

"**null**" denotes the special "undefined instance" of a class C. null denotes an undefined attribute value. The latter are then simply called "null"-values. Analogously attributes are said to be "nullpointers" or having the value "null" since they realise pointers to other carrier sets. Hence two attributes may be equal to each other though not even having the same type. Consequentially, the term-evaluation function $eval_C$ maps terms of undefined C-attributes to null ∈ AI_C.

AI_C is a value domain for all attributes, variables and terms of type C. The function VDom[62] assigns a value domain to any type of the system, including Arrays that can be constructed over primitive data or classes of the system. Hence VDom points from the set of all system types to the set of all value domains of the system[63]:

VDom: Types $\rightarrow$ $Domains_A$[64]

$Domains_A$ includes all (semantical) values of the system within system state A_{DSig}, grouped by type into so-called value-domains of the respective type.

The concept of interface-classes, known from the programming language Java, can be constituted by Object-Oriented Transformation Systems: an interface-class is represented by a class ICl whose carrier set is empty, i.e. $A_{ICl} = \varnothing$. Nevertheless there may be instances of ICl, since all objects of subclasses will be elements of the instance set AI_{ICl}. The latter corresponds with the concept of interface-classes in Java: interface-classes must not be "instantiated", i.e. must not have any objects, but since other classes may implement these interfaces, there may be instances as well.

An object obj's partial attribute value **obj.att**, interpreting an attribute symbol type_att, maps to the value domain VDom(type). In the beginning of an object's life cycle, when all its attributes are undefined, those, whose type is a class, are set null (see definition 5).

The partial attribute values of class instances $(\mathbf{static_{A,C}})_{C \in Classes}$ are defined analogously. Thus, although their name might imply, these special instances are not static at all. Their attribute values can be manipulated like the attributes of objects. The notion "static" was adopted from programming languages like Java[65].

---

62 Short for "Value-Domain"

63 rather the system' s state A_{DSig} (than "system")

64 $Domains_A$ is an abbreviation of $Domains_{ADsig}$ (see below).

65 Always trying to ease the introduction to Object-Oriented Transformation Systems for users, familiar with object-oriented systems.

Data functions $(\mathbf{f_{\underline{A}}})_{f \in DFun}$ however remain static throughout the system, e.g. mathematical functions like sinus, absolute, floor. Furthermore the carrier sets of primitive data types $(A_s)_{s \in DS}$ remain unaltered throughout the system. The following constraints state the invariability of data functions, data sorts and their carrier sets:

For all system states A_{DSig}, B_{DSig}:

OTCconstds: $B_s = A_s \quad \forall s \in DS$
OTCconstdfun: $f_B = f_A \quad \forall f \in DFun$

The data functions and carrier sets of primitive data types remain part of every single system state to satisfy the proposition that a system state A_{DSig} completely describes a possible state of the system without exception and without the need of gathering additional information of other system-wide definitions.
The carrier sets $(A_s)_{s \in DS}$ and the data functions $(f_A)_{f \in DFun}$, together, denote a partial DΣ-Algebra $\mathbf{A_{DΣ}}$ (see Remark 1) that remains static throughout the system, i.e. within all possible system states:

$\mathbf{A_{DΣ}} := ((A_s)_{s \in DS}, (f_A)_{f \in DFun})$

Example
Picking up the example of remark 1 and remark 3, we can now define a partial algebra $A_{DΣmin}$, a so called $DΣ_{min}$-reduct (algebra) of $A_{DΣ}$[66].
$A_{DΣmin}$
$A_{DΣmin,bool} = \{T,F\}$
$A_{DΣmin,nat} = \mathbb{N}$
$true_{A_{DΣmin}} = T$
$false_{A_{DΣmin}} = F$
$zero_{A_{DΣmin}} = 0$
$succ_{A_{DΣmin}}: \mathbb{N} \rightarrow \mathbb{N}$
$succ_{A_{DΣmin}}(n) = n+1$

All mentioned parts describe the structure of the system state at one point in time, neither having any knowledge about past states nor containing any information about (possible) future states. It is not before a state's call stack comes into consideration that a system state finds some place in a kind of temporal order of the data space. It is not a task of the data space to care about a temporal order or any order of the system's (possible) states.
Nevertheless since a state's call stack $\mathbf{stack_A}$ = st.(meth, vtype, vvalue) lists all methods that have been invoked, but not yet finished execution, it contains information about what methods were running and are to reawakened when others finish execution.

Example
*The method in top of the call stack ($meth_1$) is the **active method**, controlling the actual course of actions. When it calls (invokes) another method $meth_2$, $meth_2$ is pushed onto the stack taking over control of the system. By the time $meth_2$ finished execution, it is popped off the call stack, returning responsibility (and possible return values) to $meth_1$. The time between pushed onto and popped off the call stack is called the "**running time**" of a method.*

[66] If $DΣ_{min} \subseteq DΣ$ ($DΣ_{min} = (S,OP)$ is subsignature of DΣ) , $A_{DΣ_{min}} \in Alg(DΣ_{min})$, $A_{DΣ} \in Alg(DΣ)$ and $A_{DΣ_{min},s} = A_{DΣ,s} \ \forall s \in S$, $op_{A_{DΣmin}} = op_{A_{DΣ}} \ \forall op \in OP$, $A_{DΣ_{min}}$ is called $DΣ_{min}$-reduct of $A_{DΣ}$.

When a method is invoked its environment (**vtype, vvalue**) solely contains the variable "self" and the delivered arguments. Actually self (and the supplied arguments) are part of the method's set of (local) variables **X**[67]. While running, a method adds the return values of other methods to its environment[68] or declares new "local" variables[69]. Since the environments of different methods differ their derived (partial) evaluation functions eval also do.

The number of cases, describing how **eval** evaluates a given term t, depends on the different possible kinds of terms (see definition 3). Eventually eval maps terms to primitive data values or attribute values of objects. Since the latter may be undefined eval is a partial function.
The 4. and 7. case of eval's definition (see definition 4) constitute the unwanted cases of accessing an attribute of an undefined instance and accessing an element of an undefined array, respectively.[70] These incidents are known as "NullPointer-Exceptions", a special case of so-called exceptions. Exceptions and error handling denote an extension of Object-Oriented Transformation Systems that is not yet realised in this paper.

Terms allow a syntactical, state-independent representation of semantical values. The associated data values in a concrete system state A_{DSig} depend on the active method's environment. The **active object** can be identified by the value of the variable "self" that is part of every (running) method's environment. If "self" has a type, i.e. vtype(self) = C, but its value $vvalue_C(self)$ is undefined, a static method of class C is running[71].

Every system state[72] has (at least) one active method and thus an active object (or instance), the active method belongs to.

The initialisation states $\mathbf{Init_{DSig}}$ are given by

$$\mathbf{Init_{DSig}} := \{ I \mid I = ((I_s)_{s \in (DS \cup Classes)}, (f_I)_{f \in DFun}, (static_{I,C})_{C \in Classes}, (\tau,\{(self,type)\},\varnothing)) \}$$
$$I_s = \varnothing \quad \forall s \in Classes$$
$$static_{I,C}.catt = null \quad \forall C \in Classes, \forall catt \in ClStSig_C$$
$$type \in Classes$$

τ denotes the "start-up method" of the system, known as main()-method in programming languages like Java and C++. The method's environment is empty except for the type of self that has to be set to one of the classes, i.e. the class the main()-method was invoked on. The latter implicitly turns type into the so-called **main class of the system**.
Comparable to the creation of a new object by a constructor-method, in the beginning of the system all class attributes are set to null, i.e. they are undefined.

If the possibility of confusion can be excluded A_{DSig} will simply be referred to as **A**.

The set of all system states w.r.t a data space signature DSig is named $\mathbf{States_{DSig}}$. The latter can be referred to as **States**, if the meant data space signature is unambiguously clear.

The transitions between system states will then be given by Actions (see following section).

---

[67] Please note that a set of (local) variables X belongs to a certain running method and may change between system states.
[68] More precisely to X.
[69] Known as „auxiliary variables"
[70] "Undefined instance" and "undefined array" are rather improper terms for nullpointers, i.e. attributes or other pointers, like variables, whose type is a class and that have not been assigned yet.
[71] Turning the class instance of C $static_{A,C}$ into the "active instance"
[72] Except for the finalisation stated of the system

4.4. Actions & Transformations

System states A_{DSig} and B_{DSig} w.r.t. a data space signature DSig, short A and B, represent the nodes of the system's data space, i.e. the transformation graph "DS" of a constructed Object-Oriented Transformation System. An edge between these nodes denotes a transformation from system state A to B. The latter is given by an action that "transforms" the system's state from A to B. Hence the same action may be found as transformation between various (combinations of) data states.
Act_{DSig} denotes the family of all actions of a data space signature DSig or rather its data states:

$Act_{DSig} = Act_{DSig}(A,B) \qquad A,B \in States_{DSig}$

If the set $Act_{DSig}(A,B)$ of actions between system states A and B is not empty there is a transformation leading from A to the (then) subsequent state B.[73]

Within actions of $Act_{DSig}(A,B)$ all references to concrete elements of A_s or B_s in the form of signature Terms $t \in T_{DSig}(X)$, i.e. using syntactical symbols. The latter allows the abstraction from concrete values (semantical elements) and thus the use of the same action between several system states.

To check whether a term is accessible to a the active class (and its objects) of a system state A there is a function "acc" (for "accessible"), defined by:

$acc_A: T_{DSig}(X) \rightarrow \{T, F\}$

$$acc(t) = \begin{cases} T & , \text{ for } t \in X \\ acc(t_1) \wedge ... \wedge acc(t_n) & , \text{ for } t = f(t_1,...,t_n) \\ acc(t_0) \wedge vtype(self) \in AC_{C',al_{C',St}(att)} & , \text{ for } t = t_0.att,\ t_0 \in T_{DSig}(X)_{C'} \\ vtype(self) \in AC_{C,Class} \wedge vtype(self) \in AC_{C,al_{C,Cl}(att)} & , \text{ for } t = C.att \\ acc(t_0) & , \text{ for } t = t_0[t_n] \\ acc(t_0) & , \text{ for } t = \#(t_0) \end{cases}$$

The value $T \in \{T,F\} = \mathbb{B}$ will be used throughout this section as short form of $eval_{bool}(\text{ true })$. As a result of constraint OTCeval, it is specified that $eval_{bool}(\text{ true }) = T$ for all system states of all Object-Oriented Transformation Systems.

The first kind of actions between two data states contains Call- and Return-actions of methods: a Call-action represents the invoking of a method, whereas a Return-action denotes the end of a method's execution. There can be Call- (and Return-) actions for both instance methods $meth \in BehSig_C$ of a class C and class methods $cmeth \in ClBehSig_C$:

[73] At this point of development the set of action between two system states A and B does not happen to have more than one element, thus we distinguish two cases: there is a transformation between A and B or not.

Definition 5 (Call-Actions)

(**a**) Call-actions for (instance) methods al_type_meth($s_1 \ldots sn$) $\in$ $BehSig_C$:

$\underline{call(\ t_0.meth(t1,\ldots,tn)\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_A = st.(m, vtype, vvalue)$[74]
   $stack_B = st.(m, vtype, vvalue).(meth, vtype^2, vvalue^2)$

2) $vtype(self) \in AC_{C,al}$

3) $eval(t_0) \in AI_C$
   $t_0.exists = true$[75]

4) $eval(t_i) \in VDom(s_i)$ $\quad \forall i \in [1,n]$

5) $X^2 = \{self\} \cup \{arg_i \mid i \leq n \ \wedge\ i \in \mathbb{N}^+\}$

6) $vtype^2(self) = C$
   $vvalue^2(self) = vvalue(t_0)$

7) $vtype^2(arg_i) = s_i$ $\quad \forall i \in [1,n]$
   $vvalue^2(arg_i) = eval(t_i)$ $\quad \forall i \in [1,n]$

0) $B_{Cl} = A_{Cl}$ $\quad \forall Cl \in Classes$
   $static_{B,Cl} = static_{A,Cl}$ $\quad \forall Cl \in Classes$

constraint1: $acc_A(t_0) = T$
constraint2: $acc_A(t_i) = T$ $\quad \forall i \in [1,n]$

◇

(b) Call-actions for class methods al_type_cmeth($s_1 \ldots s_n$) $\in$ $ClBehSig_C$:

$\underline{call(\ C.cmeth(t_1,\ldots,t_n)\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_A = st.(m, vtype, vvalue)$
   $stack_B = st.(m, vtype, vvalue).(cmeth, vtype^2, vvalue^2)$

2) $vtype(self) \in AC_{C,Class}$
   $vtype(self) \in AC_{C,al}$

3) $eval(t_i) \in VDom(s_i)$ $\quad \forall i \in [1,n]$

4) $X^2 = \{self\} \cup \{arg_i \mid i \leq n \ \wedge\ i \in \mathbb{N}^+\}$

5) $vtype^2(self) = C$
   $vvalue^2(self) = \begin{cases} obj & \text{, for cmeth = constructor} \\ undefined & \text{, otherwise} \end{cases}$

6) $obj.exists = eval_{bool}(true)$
   $obj.att$ undefined $\quad \forall att \in StSig_{C,s}, \forall s \in DS$
   $obj.att = null$ $\quad \forall att \in StSig_{C,Cl}, \forall Cl \in (Types \setminus DS)$

---

[74] $st \in (Meth \times VType_A \times VValue_A)^*$

[75] The statement of this equation is equivalent to the condition $eval_{bool}(t_0.exists) = eval_{bool}(true)$. Below: whenever equations exclusively contain terms, the evaluation function eval is implicitly meant.

$^{7)}$ $vtype^2(arg_i) = s_i$ $\quad \forall i \in [1,n]$

$vvalue^2(arg_i) = eval(t_i)$ $\quad \forall i \in [1,n]$

$^{0)}$ $B_C = \begin{cases} A_C \uplus \{obj\} & \text{, for cmeth = constructor} \\ A_C & \text{, otherwise} \end{cases}$

$B_{Cl} = A_{Cl}$ $\quad \forall Cl \in (Classes \setminus \{C\})$

$static_{B,Cl} = static_{A,Cl}$ $\quad \forall Cl \in Classes$

constraint1: $acc_A(t_i) = T$ $\quad \forall i \in [1,n]$

◇

Remark 5 (Call-Actions)

(**a**) A Call-action **call(t_0.meth(t_1,…,t_n))** changes the system's state from state A to B: the active object, denoted by the variable self[76], invokes method meth on object $eval_C(t_0)$. meth is being pushed onto the call stack, joint by its environment ($vtype^2$,$vvalue^2$). Thus meth becomes the active method and $eval_C(t_0)$ the active object. When meth is invoked, its environment only contains the variable self and the delivered arguments arg_1 through arg_n.

Several conditions specify the transformation(s) made by **call(t_0.meth(t_1,…,t_n))**:

1) The invoked method meth is pushed onto the stack, together with its environment ($vtype^2$,$vvalue^2$).
2) The active object self can access the method meth, i.e. self is part of the respective access class to meth's access level al. If t_0 denotes an instance of C that is not a C-object[77], meth's access level within class C is deciding.
3) The object, meth is invoked on, has to be alive and not destructed before.
4) The supplied arguments have to be elements of proper value-domains.
5) In the moment of invocation meth does not contain local variables other than self and the supplied arguments.
6) The object, denoted by t_0, becomes the active object (in state B)
7) t_i becomes the i. argument of meth, joining meth's environment ($vtype^2$,$vvalue^2$); $i \in [1,n]$.
0) All object sets and all static objects remain unaltered.

constraint1 states that t_0, the object meth is invoked on, has to be accessible for self, the formerly active object.
constraint2 states that the input parameters t_i have to be accessible for self.

(**b**) A Call-action **call(C.cmeth(t_1,…,t_n))** changes the system's state from state A to B: the active object, denoted by the variable self, invokes class-method cmeth on class C, turning cmeth into the active method. Since cmeth is a class-method there is no active object in state B, leaving the variable self undefined. In case cmeth is the special constructor-method a new C-object obj is created and turned into the active object at the same time. cmeth is being pushed onto the call stack, joint by its environment ($vtype^2$,$vvalue^2$). When meth is invoked, its environment only contains the variable self and the supplied arguments arg_1 through arg_n.

[76] Or rather eval(self)
[77] Thus it is an object of a subclass of C.

Several conditions specify the transformation(s) made by **call (C.cmeth(t_1,…,t_n))**:

1) The invoked method meth is pushed onto the stack, together with its environment (vtype², vvalue²).
2) The active object self can access class C and method cmeth.
3) The supplied arguments have to be elements of proper value-domains.
4) In the moment of invocation cmeth does not contain local variables other than self and the supplied arguments.
5) The variable self remains undefined. If the called method is the special constructor method, the object set of class C is extended by a new object obj, becoming the active object of state B.
6) This condition only applies if the invoked method is the special "constructor" method and thus a new object obj is created. The state of obj consists of undefined attribute values, which may be set to initial values by the constructor afterwards. Therefore, those attributes, whose type is a class, are set to null.
7) t_i becomes the i. argument of meth, joining meth's environment (vtype², vvalue²); $i \in [1,n]$.
0) All object sets except the one of class C, and all static objects remain unaltered. The object set to class C is extended by the newly created instance of C, if cmeth is the special constructor method.

constraint1 states that the input parameters t_i have to be accessible for self.

Definition 6 (Return-Actions)

(**a**) Return-actions for methods $al\_s_1 \ldots sn\_meth(args) \in BehSig_C$:

$\underline{return(\ meth.return(t_1,\ldots,t_n)\ )} \in \underline{Act_{DSig}(A,B)}$, if

1) $stack_A = st.(m, vtype, vvalue).(meth, vtype², vvalue²)$
   $stack_B = st.(m, vtype', vvalue')$

2) $eval²(t_i) \in VDom(s_i) \quad \forall i \in [1,n]$[78]

3) $X' = X \cup \{ret_{(q+i)} \mid \forall i \in [1,n]\}$[79]
   $\forall i \in [1,n]$:
   $vtype' = vtype \cup \{(ret_{(q+i)}, s_i)\}$
   $vvalue' = vvalue \cup \{(ret_{(q+i)}, eval²(t_i))\}$[80], where
   q denotes the number of collected return values[81]

0) $B_{Cl} = A_{Cl} \quad \forall Cl \in Classes$
   $static_{B,Cl} = static_{A,Cl} \quad \forall Cl \in Classes$

constraint1: $acc_A(t_i) = T \quad \forall i \in [1,n]$

◇

---

[78] $eval² := eval(vtype², vvalue²)$, see definition 4.

[79] vtype: $X \rightarrow$ Types, vtype': $X' \rightarrow$ Types.
Whensoever the term X' is used, it denotes the domain of the functions vtype' and vvalue'.

[80] $eval²(s_i)$ undefined $\Rightarrow$ $ret_{(q+i)}$ undefined

[81] $q = | X \setminus (\{self\} \cup \{ arg_i \mid i \in \mathbb{N}^+ \}) | = | \{ret_i \mid ret_i \in X \wedge i \in \mathbb{N}^+\} |$, where
|M| denotes the cardinality of set M

(**b**) Return-actions for class methods al_s_1…sn_cmeth(args) ∈ $ClBehSig_C$:

$\underline{return(\ cmeth.return(t_1,\ldots,t_n)\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_A$ = st.(m, vtype, vvalue) .(cmeth, vtype², vvalue²)
   $stack_B$ = st.(m, vtype', vvalue')

2) $eval^2(t_i) \in VDom(s_i)$ $\forall i \in [1,n]$

3) $X' = X \cup \{ret_{(q+i)} \mid i \in [1,n]\}$
   $\forall i \in [1,n]$:
   $vtype' = vtype \cup \{(ret_{(q+i)}, s_i)\}$
   $vvalue' = vvalue \cup \{(ret_{(q+i)}, eval^2(t_i))\}$, where
   q denotes the number of collected return values

0) $B_{Cl} = A_{Cl}$ $\forall Cl \in Classes$
   $static_{B,Cl} = static_{A,Cl}$ $\forall Cl \in Classes$

constraint1: $acc_A(t_i) = T$ $\forall i \in [1,n]$
constraint2: (cmeth = constructor) $\Rightarrow$ (t_1 = self)

◇

Remark 6 (Return-Actions)

(**a**) A Return-action **return(meth.return(t_1,…,t_n))** changes the system's state from state A to B: the active method (meth) on top of the call stack ($stack_A$) finishes execution and returns responsibility to the method (m) that invoked meth[82]. The latter is popped off the call stack and its return values are added to the environment of m. Mostly there is not more than one return value[83].

Several conditions specify the transformation(s) made by **return(meth.return(t_1,…,t_n))**:

1) meth lies on top of the call stack in state A, whereas meth was popped off the stack in B. The then active method m retrieves responsibility.
3) The returned values are added to method m's environment: if the environment already holds q return values (as local variables in X), the first return value becomes the (q+1). local variable of m, implicitly adding $ret_{(q+1)}$ to the set X'. In this manner all return values will be added to the environment.
0) All object sets and all static objects remain unaltered.

constraint1 states that the returned values t_i have to be accessible for self.

(**b**) A Return-action **return(cmeth.return(t1,…,tn))** changes the system's state from state A to B, i.e. the active class method (cmeth) on top of the call stack ($stack_A$) finishes execution and returns responsibility to the method (m) that invoked cmeth. Therefore cmeth is popped off the call stack and its return values are added to the environment of m, being the method that invoked cmeth. Naturally there is at most one return value (see remark 6a). If cmeth was the special "constructor" method it created a new object, denoted by $eval_C$(self), that will be returned as single return-value.

[82] Method m that invoked meth takes second place in the call stack of state A and thus lies on top of $stack_B$.
[83] Major programming languages like Java and C++ and specification languages like the UML restrict methods to have at most one return value.

Several conditions specify the transformation(s) made by **return(cmeth.return(t1,…,tn))**:

1) cmeth lies on top of the call stack in state A and was popped off the stack in B. The then active method m retrieves responsibility.

3) The returned values are added to method m's environment: if the environment already holds q return values (as local variables in X), the first return value becomes the (q+1)., implicitly adding $ret_{(q+1)}$ to the set of local variables X'. In this manner all return values will be added to the environment.

0) All object sets and all static objects remain unaltered. Even if cmeth was the special constructor method there will be no change, since the newly created object has been added before (see definition 5b).

constraint1 states that the returned values t_i have to be accessible for self.

constraint2 states: if cmeth was the special constructor method, it returns the newly created C-object, denoted by $eval_C(self)$. There will be no more return values, since the special constructor method only has one return value (see OTCconstructor).

Definition 7 (Assignment-Actions)

(**a**) Assignment-actions for (instance-) attributes al_type_att $\in StSig_C$:

$\underline{assign(\ t_0.att, t_1\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_B = stack_A$

2) $acc_A(t_0.att) = T$

3) $eval(t_0) \in AI_C$
$t_0.exists = true$[84]
$eval_C(t_0) =: obj$
$eval'_C(t_0) =: obj'$[85]

4) $eval(t_1) \in VDom(type)$

5) $obj'.att = eval_{type}(t_1)$[86]
$obj'.a = obj.a \quad \forall a \in StSig_C \setminus \{att\}$[87]
$obj'.exists = obj.exists$

0) $B_{Cl} = A_{Cl} \quad \forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl} \quad \forall Cl \in Classes$

constraint1: $acc_A(t_0) = T$
constraint2: $acc_A(t_1) = T$

◇

---

[84] The statement of this equation is equivalent to the condition $eval_{bool}(t_0.exists) = eval_{bool}(true)$. Whensoever equations exclusively contain terms, the evaluation function eval is implicitly meant.

[85] eval: $T_{DSig} \rightarrow A$, eval': $T_{DSig} \rightarrow B$

[86] eval(t1) undefined $\Rightarrow$ obj'.att undefined

[87] obj'.a undefined $\Leftrightarrow$ obj.a undefined

(**b**) Assignment-actions for class attributes al_type_catt ∈ $ClStSig_C$:

$\underline{assign(\ C.catt,\ t_1\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_B = stack_A$

2) $acc_A(C.catt) = T$

3) $eval(t_1) \in VDom(type)$

4) $static_{B,C}.catt = eval_{type}(t_1)$
$static_{B,C}.a = static_{A,C}.a \quad \forall a \in ClStSig_C \setminus \{catt\}$[88]

0) $B_{Cl} = A_{Cl} \quad \forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl} \quad \forall Cl \in Classes \setminus \{C\}$

constraint1: $acc_A(t_1) = T$

◇

(**c**) Creation-actions for local variables $ret_i \in X$:

$\underline{assign(\ val\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_A = st.(m, vtype, vvalue)$
$stack_B = st.(m, vtype', vvalue')$

2) $s \in DS$
$eval_s(val) \in A_s$

3) $X' = X \cup \{ret_{(q+1)}\}$
$vtype' = vtype \cup \{(ret_{(q+1)}, s)\}$
$vvalue' = vvalue \cup \{(ret_{(q+1)}, val)\}$, where
q denotes the number of collected return values

0) $B_{Cl} = A_{Cl} \quad \forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl} \quad \forall Cl \in Classes$

◇

(**d**) Assignment-actions for local variables $ret_i \in X$:

$\underline{assign(\ ret_i,\ t_1\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_A = st.(m, vtype, vvalue)$
$stack_B = st.(m, vtype', vvalue')$
$vtype' = vtype$

2) $ret_i \in X \setminus (\{self\} \cup \{arg_i \mid i \in \mathbb{N}^+\})$

3) $eval(t_1) \in VDom(vtype(ret_i))$

4) $X' = X$
$vvalue'(ret_i) = eval_{type}(t_1)$
$vvalue'(var) = vvalue(var) \quad \forall var \in X \setminus \{ret_i\}$

0) $B_{Cl} = A_{Cl} \quad \forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl} \quad \forall Cl \in Classes$

constraint1: $acc_A(t_1) = T$

---

[88] obj'.a undefined ⇔ obj.a undefined

◇

Remark 7 (Assignment-Actions)

(**a**) An Assignment-action **assign(t_0.att, t_1)** changes the system's state from state A to B, i.e. the object obj's attribute att is set to a new value, denoted by the term eval(t_1). If the evaluation of t_1 is undefined (or null), the new value of obj'.att also is undefined (or is null).

Several conditions specify the transformation(s) made by **assign(t_0.att, t_1)**:

1) The call stack remains unaltered, since all local variables maintain their old values.
2) The attribute, to be changed, is accessible for self.
3) obj denotes the state of the object, whose attribute is changed, in system state A, obj' denotes its state in B: obj $\in A_C$, obj' $\in B_C$. On the one hand obj and obj' denote the same object symbol (obj = obj'), but obj induces a family of partial attribute values in A and obj' induces a family of partial attribute values in B (see definition 4).
5) obj' of system state B is equal to obj of system state A, except for the value of attribute att.
0) All object sets and all static objects remain unaltered, even the family of induced attribute values by obj changes between system states A and B, since the value of obj.att changes.

constraint1 states that t_0, the object meth is invoked on, has to be accessible for self. This constraint is already secured by condition 2) (see definition of acc), but for consistency reasons restated by constraint1.

constraint2 states that the new value of att, denoted by $eval_{type}(t_1)$ has to be accessible for self.

This approach differs from the original object-oriented concept of "information hiding" / "data abstraction". The latter specifies that the data (values) of an object must not be manipulated from the outside (see section 2.1.). However major object-oriented systems of the present permit the direct manipulation of other object's attributes and thus Object-Oriented Transformation Systems do. This does not inevitably mean that Object-Oriented Transformation Systems violate this object-oriented concept, since the access levels of all attributes can be set to "private".

(**b**) An Assignment-action **assign(C.att, t_1)** changes the system's state from state A to B, i.e. the class attribute catt is set to a new value, denoted by the term eval(t_1). If the evaluation of t_1 is undefined (or null), the new value of C.catt also is undefined (or is null).

Several conditions specify the transformation(s) made by **assign(C.att, t_1)**:

1) The call stack remains unaltered, since all local variables maintain their old values.
2) The class attribute, to be changed, is accessible for self.
4) the "static instance" of C in system state B is equal to the static instance of C in system state A, except for the value of attribute catt.
0) All object sets remain unaltered, all static objects of classes other than, C remain unaltered.

constraint1 states that the new value of catt, denoted by $eval_{type}(t_1)$ has to be accessible for self

(**c**) A Creation-action **assign(val)** changes the system's state from state A to B, i.e. the local variable ret_i of type s is created and assigned to the primitive data value eval(val). This action may slightly differ from the general concept of information hiding as any value of primitive data types can be assigned. However it does not violate the object-oriented concepts of "data encapsulation" or "data abstractions" as primitive data values are static throughout the system, do not have identities, and eventually do not represent any entities of the system (see remark 2).

Several conditions specify the transformation(s) made by **assign(val)**:

1) The environment of the running method m changes as a new local variable is added.
2) The value of the new local variable is denoted by the term val.
3) $ret_{(q+1)}$ is added to the environment (vtype',vvalue') of the active method, where q is the number of local variables, the environment already holds.

(**d**) An Assignment-action **assign(ret_i, t_1)** changes the system's state from state A to B, i.e. the local variable ret_i is assigned to a new value, denoted by eval(t_1). If the evaluation of t_1 is undefined or null, the new value of ret_i is also undefined or null.

Several conditions specify the transformation(s) made by **assign(ret_i, t_1):**

1) The environment of the running method m changes, since the value of the local variable ret_i changes. However the types of all local variables remain unaltered.
2) The local variable, to be changed, exists.
4) The set of local variables does not change, only the value of ret_i is set to a new value.

constraint1 states that the new value of ret_i, denoted by $eval_{type}(t_1)$ has to be accessible for self.

Definition 8 (Other Actions)

(**a**) Destruction-action of objects obj $\in A_C$, with C $\in$ Classes:

$\underline{destruct(\ t_0\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_B = stack_A$

2) $eval(t_0) \in AI_C$
   $t_0.exists = true$
   $eval_C(t_0) =: obj$
   $eval'_C(t_0) =: obj'$

3) $obj'.exists = eval_{bool}(false)$
   $obj'.a = obj.a \qquad \forall\ a \in StSig_C$

0) $B_{Cl} = A_{Cl} \qquad \forall Cl \in Classes$
   $static_{B,Cl} = static_{A,Cl} \qquad \forall Cl \in Classes$

constraint1: $acc_A(t_0) = T$

◇

(**b**) Finalisation-actions:

$\underline{finalise()\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_A = (\tau, vtype^2, vvalue^2)$
   $stack_B = \lambda$

0) $B_{Cl} = A_{Cl} \qquad \forall Cl \in Classes$
   $static_{B,Cl} = static_{A,Cl} \qquad \forall Cl \in Classes$

◇

Remark 8 (Other Actions)

(**a**) A Destruction-action **destruct(t_0)** changes the system's state from state A to B, i.e. the object, denoted by eval(t0), "dies" and its life-cycle ends. The object is not deleted from the carrier set of C, because variables and attributes may still point on eval(t_0), but afterwards their value is equal to the one on an undefined variable.

Several conditions specify the transformation(s) made by **destruct(t0)**:

2) obj denotes the state of the object, to be destructed, in system state A, obj' denotes its state in B: obj $\in A_C$, obj' $\in B_C$. The object was "alive", i.e. existed, in system state A.

3) obj' of system state B is equal to obj of system state A, except for the value of attribute att.

constraint1 states that object, to be destructed, is accessible for self.

(**b**) A Finalisation-action **finalise()** changes the system's state from state A to B, i.e. the system's start-up function (main()-function) finishes execution. The system changes into its finalisation-state. Thus the system completes its execution and terminates.

The following conditions specifies the transformation(s) made by **finalise()**:

1) The call stack of system state A contains solely the system's start-up method τ, in many of today's object-oriented systems known as "main()"-method.

4.5. Object-Oriented Transformation Systems as Institution

The following section proves that Object-Oriented Transformation Systems are an instantiation of generic Transformation Systems by M. Große-Rhode.

M. Große Rhode defines Transformation System on the basis of partial algebras as data states. Nevertheless he offers a possibility to consider other data state models without losing the theoretical results he presents in [Gro01]. The precondition to replace partial algebras as data states by other models – the system states of Object-Oriented Transformation Systems for example – is that these form a concrete institution, in the sense of [GB84b] and [GB92].
The institution-independent definition of Transformation Systems, within this paper referred to as "generic Transformation Systems", is given in section 2.4 of [Gro01]: an arbitrary, but concrete data institution is extended to a specification framework that represents the basis of generic Transformation Systems. Later in that paper the composition as well as development relations of Transformation Systems are proven to apply for Transformation Systems w.r.t other Specification Frameworks, i.e. generic Transformation Systems.

In order to prove that Object-Oriented Transformation Systems are an instantiation of generic Transformation Systems, it has to be proven that the data space signatures of Object-Oriented Transformation Systems (see definition 1 and definition 2) and their respective system states (see definition 4) can be extended to form a concrete institution in the sense of [GB84b] and [GB92].
Since the data space signatures of generic Transformation Systems were not intended to contain behavioural information data **state** signatures will be introduced as a trimmed version of data **space** signatures, abstracting from the signature's method symbols, i.e. omitting $BehSig_C$ and $ClBehSig_C$ for all classes $C \in Classes$.

This section is structured in the following way:
First of all the definitions of a concrete institution and a specification framework will be recalled, mostly taken from [Gro01].

Afterwards it will be shown that Object-Oriented Transformation Systems form a concrete institution. Therefore the category of data state signatures has to be defined as well as a category of system states for each data state signature.
Every concrete institution can be extended to form a specification framework, in order to instantiate generic Transformation Systems. This extension is described in section 2.3. of [Gro01].

Finally a short outlook will be given, what is possible with the theorems that can be applied.

4.5.1. Concrete Institutions

A concrete institution is given by

INST = (**SIGN, Sen, Mod, |=, sorts,** |_|), with

- a category **SIGN**, where
  the objects are signatures and the morphisms are signature morphisms.
- a functor **Sen**: **SIGN** $\rightarrow$ SET,
  giving a set Sen(Σ) of Σ-sentences for each signature $\Sigma \in$ **SIGN** and a translation of sentences Sen(σ): Sen(Σ) $\rightarrow$ Sen(Σ') for each signature morphism σ: Σ $\rightarrow$ Σ' in **SIGN**.
- a functor **Mod**: **SIGN** $\rightarrow$ CAT^{op},
  giving a category Mod(Σ) of Σ-models for each signature $\Sigma \in$ |**SIGN**| and a forgetful functor Mod(σ): Mod(Σ') $\rightarrow$ Mod(Σ) for each signature morphism σ: Σ $\rightarrow$ Σ' in **SIGN**.
- a satisfaction relation $|=_\Sigma \subset |Mod(\Sigma)| \times Sen(\Sigma)$
  for each signature $\Sigma \in$ |**SIGN**|.
- a functor sorts: **SIGN** $\rightarrow$ SET[89],
  mapping every signature $\Sigma \in$ SIGN to a set of sort-symbols
- a natural transformations |_|: Mod $\Rightarrow$ SSet ° sorts,
  i.e., a family of functors $|\_|_\Sigma$: Mod(Σ) $\rightarrow$ $Set^{sorts(\Sigma)}$ such that
  $|\_|_\Sigma \circ V_\sigma = V_{sorts(\sigma)} \circ |\_|_{\Sigma'}$,
  where $V_\sigma := Mod(\sigma)$[90]

such that the following satisfaction condition holds
for every signature morphism σ: Σ1 $\rightarrow$ Σ2, every Σ2-model A_2 and every Σ1-sentence e1:

$$A_2 \models_{\Sigma 2} Sen(\sigma)(e1) \Leftrightarrow Mod(\sigma)(A_2) \models_{\Sigma 1} e1$$

◇

4.5.2. Specifification Framework

Let INST = (SIGN, Sen, Mod, |=, sorts, |_|) be a concrete institution.
The corresponding specification framework is then given by:

- A data space signature $D\Sigma = (\Sigma, N)$[91] w.r.t. INST is given by a signature $\Sigma \in$ |SIGN| and an S* × S*-indexed set $N = (N_{w,w'})_{w,w' \in S^*}$, where S := sorts(Σ).
- An action structure AS w.r.t. INST is given by a family of functors
  $AS_{D\Sigma}$: $SET^S \times SET^S \rightarrow SET$
  for each data space signature DΣ = (Σ,N), where S := sorts(Σ).

◇

[89] SET denotes the category of sets and functions between sets.
[90] V_σ is called the forgetful functor w.r.t. the data space signature morphism σ.
[91] The term "data space signature" is also used to define ordinary Transformation Systems.

4.5.3. Category of data state signatures

In this section the category of data space signatures CAT_{DSig} will be defined. Data state signature morphisms enable the translation of signatures.
Data space signatures are given by definition 1: DSig = (DS, DFun, AL, Classes), where every class $C \in Classes$ induces a class signature CSig, given by definition 2:
$CSig = (Super_C, AC_C, StSig_C, BehSig_C, ClStSig_C, ClBehSig_C)$.

Definition 10 (Data state signature)

Given a data space signature and induced class signatures as introduced by definition 1 and definition 2.
In order to specify data state signatures the behavioural parts of the original class signatures have to be omitted, resulting in the following definition:

DSig' = (DS, DFun, AL, Classes),
where every class $C \in Classes$ induces a class signature CSig', given by:

$CSig' = (Super_C, AC_C, StSig_C, ClStSig_C)$.

◇

Definition 11 (Data state signature morphism)

$(\sigma: DSig_1 \rightarrow DSig_2) := (\sigma_{DS}, \sigma_{DFun}, \sigma_{AL}, \sigma_C)$, where

- $\sigma_{DS}: DS_1 \rightarrow DS_2 = (\sigma_{DS,s})_{s \in DS_1}$
  a mapping of (primitive) data symbols.
- $\sigma_{DFun}: DFun_1 \rightarrow DFun_2$
  a mapping of function symbols (see constraint DMC1 below)[92].
- $\sigma_{AL}: AL_1 \rightarrow AL_2$
  a mapping of access levels (see constraint DMC3 below).
- $\sigma_C: Classes_1 \rightarrow Classes_2 = (\sigma_{C,Cl})_{Cl \in Classes_1}$
  a family of class signature morphisms, where
  the class signature morphism $\sigma_{C,Cl}$ for class Cl is given by
  $\sigma_{C,Cl} = (\sigma_{C,Cl,St}, \sigma_{C,Cl,ClSt})$, with
  $\sigma_{C,Cl,St}: StSig_{Cl} \rightarrow StSig_{\sigma_C(Cl)}$
  $\sigma_{C,Cl,ClSt}: ClStSig_{Cl} \rightarrow ClStSig_{\sigma_C(Cl)}$

For every data state signature morphism it is required that the following constraints hold:

DMC1: $\forall f \in DFun_1$:
$f \in DFun_{1,w \rightarrow s} \Rightarrow \sigma_{DFun}(f) \in DFun_{2,\sigma_{DS}(w) \rightarrow \sigma_{DS}(s)}$

DMC2: $\forall Cl \in Classes_1$:
$CP \in Super_{Cl} \Rightarrow \sigma_C(CP) \in Super_{\sigma_C(Cl)}$

DMC3: $\forall Cl \in Classes_1, \forall al \in AL_1$:
$C2 \in AC_{Cl,al} \Rightarrow \sigma_C(C2) \in AC_{\sigma_C(Cl),\sigma_{AL}(al)}$

DMC4,5: $\forall Cl \in Classes_1, \forall al \in AL_1, \forall type \in Types_1^*$:
$att \in StSig_{Cl,al,type} \Rightarrow \sigma_{C,Cl,St}(att) \in StSig_{\sigma_C(Cl),\sigma_{AL}(al),\sigma_{Types}(type)}$
$catt \in ClStSig_{Cl,al,type} \Rightarrow \sigma_{C,Cl,ClSt}(catt) \in ClStSig_{\sigma_C(Cl),\sigma_{AL}(al),\sigma_{Types}(type)}$

---

[92] Originally DFun was introduced as a family of sets of function-symbols (see definition 1). Since σ_{DFun} maps all function-symbols of $DFun_1$ to $DFun_2$, correct typing has to be secured explicitly by constraint DMC2.

DMC6: $\forall Cl \in Classes_1$:
$\sigma_{C,Cl,St}(exists) = exists$

◇

Remark 11 (Data state signature morphism)

σ_{DS}, σ_{DFun}, and σ_{AL}, the mapping of sort symbols, function symbols, and access levels, respectively, denote total functions. While DS_1 and AL_1 evidently denote sets (of data sort symbols and access-level-constants, respectively), $DFun_1$ denotes the union over all sets $DFun_{1,w,s}$, where $w \in DS_1^*$ and $s \in DS_1$, (see Remark 1 for the definition of the statement "f $\in$ DFun").

σ_{DS}: $DS_1^* \rightarrow DS_2^*$ denotes the extension of σ_{DS}: $DS_1 \rightarrow DS_2$ on words, i.e.

- $\sigma_{DS}(\lambda) = \lambda$
- $\sigma_{DS}(sw) = \sigma_{DS}(s)\,\sigma_{DS}(w)$

σ_{Types}: $Types_1 \rightarrow Types_2$ denotes the extension of σ_{DS}: $DS_1 \rightarrow DS_2$ and σ_C: $Classes_1 \rightarrow Classes_2$ on all system's types, i.e.

$$\sigma_{Types}(type) = \begin{cases} \sigma_{DS}(type) & , \text{for } type \in DS_1 \\ \sigma_C(type) & , \text{for } type \in Classes_1 \\ \sigma_{Types}(t)[] & , \text{for } type = t[] \end{cases}$$

σ_{Types}: $Types_1^* \rightarrow Types_2^*$ denotes the extension of σ_{Types}: $Types_1 \rightarrow Types_2$, i.e.:

- $\sigma_{Types}(\lambda) = \lambda$
- $\sigma_{Types}(sw) = \sigma_{Types}(s)\,\sigma_{Types}(w)$

In order to adjust data space signature morphisms for concurrent Object-Oriented Transformation Systems, another constraint has to be added, specifying the mapping of the obligatory attribute "locked":

DMC7: $\sigma_{C,Cl,St}(locked) = locked \qquad \forall Cl \in Classes_1$:

Fact 12 (Category DSIG)

The category DSIG has

- data state signatures as objects,
- data state signature morphisms as morphisms,
- the composition $(\tau \circ \sigma)$: $DSig_1 \rightarrow DSig_3$
  of two morphisms σ: $DSig_1 \rightarrow DSig_2$ and τ: $DSig_2 \rightarrow DSig_3$ is given by:
  $(\tau \circ \sigma)$: $DSig_1 \rightarrow DSig_3 := (\tau_{DS} \circ \sigma_{DS},\ \tau_{DFun} \circ \sigma_{DFun},\ \tau_{AL} \circ \sigma_{AL},\ \tau_C \circ \sigma_C)$
  $(\tau_{DS} \circ \sigma_{DS})$: $DS_1 \rightarrow DS_3$
  denotes the usual composition of functions between sets,
  corresponding with the composition of morphisms in category SET[93].
  $(\tau_{DFun} \circ \sigma_{DFun})$: $DFun_1 \rightarrow DFun_3$
  denotes the usual composition of morphisms between sets,
  corresponding with the composition of functions in category SET.

[93] SET denotes the category of sets (as objects) and functions (as morphisms).

$(\tau_{AL} \circ \sigma_{AL})$: $AL_1 \to AL_3$

denotes the usual composition of functions between sets, corresponding with the composition of morphisms in category SET.

$(\tau_C \circ \sigma_C)$: $Classes_1 \to Classes_3 = ((\tau_C \circ \sigma_C)_{Cl})_{Cl \in Classes_1}$

denotes the family of composition of class signature morphisms, where the composition of class signature morphism $(\tau_C \circ \sigma_C)_{Cl}$ for class Cl is given by

$(\tau_C \circ \sigma_C)_{Cl} = ((\tau_C \circ \sigma_C)_{Cl,St}, (\tau_C \circ \sigma_C)_{Cl,Beh}, (\tau_C \circ \sigma_C)_{Cl,ClSt}, (\tau_C \circ \sigma_C)_{Cl,ClBeh})$,

$(\tau_C \circ \sigma_C)_{Cl,St}$: $StSig_{Cl} \to StSig_{(\tau_C \circ \sigma_C)(Cl)}$
$= \tau_{C,\sigma_C(Cl),St} \circ \sigma_{C,Cl,St}$

$(\tau_C \circ \sigma_C)_{Cl,ClSt}$: $ClStSig_{Cl} \to ClStSig_{(\tau_C \circ \sigma_C)(Cl)}$
$= \tau_{C,\sigma_C(Cl),ClSt} \circ \sigma_{C,Cl,ClSt}$

Since $StSig_{Cl}$ and $ClStSig_{Cl}$ denote sets[94] of attribute symbols, the two compositions $(\tau_C \circ \sigma_C)_{Cl,St}$ and $(\tau_C \circ \sigma_C)_{Cl,ClSt}$ denote the usual composition of functions between sets.

- the identity data space signature morphism id_{DSig_1}: $DSig_1 \to DSig_1$

$id_{DSig_1} = (id_{DSig_1,DS}, id_{DSig_1,DFun}, id_{DSig_1,AL}, id_{DSig_1,C})$, where

$id_{DSig_1,DS}$: $DS_1 \to DS_1$ is the identity map of data sort symbols.

$id_{DSig_1,DFun}$: $DFun_1 \to DFun_1$ is the identity map of function symbols.

$id_{DSig_1,AL}$: $AL_1 \to AL_1$ is the identity map of access level constants.

$id_{DSig_1,C}$: $Classes_1 \to Classes_1 = (id_{DSig_1,C,Cl})_{Cl \in Classes_1}$

denotes the family of identity class signature morphisms:

$id_{DSig_1,C,Cl} = (id_{DSig_1,C,Cl,St}, id_{DSig_1,C,Cl,ClSt})$,

$id_{DSig_1,C,Cl,St}$: $StSig_{Cl} \to StSig_{Cl}$ and
$id_{DSig_1,C,Cl,ClSt}$: $ClStSig_{Cl} \to ClStSig_{Cl}$
denote the identity mappings of attribute symbols.

▽

4.5.4. Model-functor & category of system data states

The functor Mod_{OOTS} is the model functor in this institution. It interprets each data state signature DSig by the category of its respective system states. Data state signature morphisms are mapped to forgetful functors between the respective categories of system states.

The definition for system states has been given in definition 4. However, in order to omit behavioural information from the system states, system **data** states are introduced:

Definition 13 (System data state)

Given a system state A as introduced by definition 4:
$A = ((A_s)_{s \in (DS \cup Classes)}, (f_A)_{f \in DFun}, (static_{A,Cl})_{Cl \in Classes}, stack_A)$.

[94] See remark 2 for (alternative) definitions of the set $StSig_C$ and the statement $(att \in StSig_{C,type})$.

In order to specify the respective system data state the behavioural parts, i.e. the (primitive) data functions and the call stack(s), of the original system state have to be omitted, resulting in the following definition:

$A' = ((A'_s)_{s \in (DS \cup Classes)}, (static_{A',Cl})_{Cl \in Classes})$

◇

System data state morphisms are then introduced as follows:

Definition 14 (System data state morphism)

Given a data state signature DSig = (DS, DFun, AL, Classes) and two system data states
$A = ((A_s)_{s \in (DS \cup Classes)}, (static_{A,Cl})_{Cl \in Classes})$,
$B = ((B_s)_{s \in (DS \cup Classes)}, (static_{B,Cl})_{Cl \in Classes})$.

A system data state morphism $h: A \rightarrow B$ is then given by:

$h := ((h_s)_{s \in (DS \cup Classes)})$, where

- $h_s: A_s \rightarrow B_s$

  1. case ($s \in DS$)
  a mapping for the carrier set of primitive data type.

  2. case ($s \in Classes$)
  a mapping of objects (symbols)[95].

For every system data state morphism it is required that the following constraint holds:

SMC1: $\forall s \in DS:$
$h_s(x) = x \quad \forall x \in A_s$

◇

Remark 14 (System data state morphism)

The mapping of primitive data sorts os the identity mapping, since the carrier sets of primate data types can not change within an Object-Oriented Transformation System[96], specified by constraint OTCconstdfun (see constraint SMC1).

There is no explicit mapping class instances $static_{A,C}$ to $static_{B,C}$, since every system state has to have exactly one class instance for every class $C \in Classes$.

The definition of system data state morphisms is ready for concurrent Object-Oriented Transformation Systems.

Fact 15 (Category $Mod_{OOTS}(DSig)$)

Given the appropriate data state signature DSig to a given Object-Oriented Transformation System OOTS: the category $Mod_{OOTS}(DSig)$ then consists of

- system data states of DSig as objects,
- system data state morphisms as morphisms,

---

[95] Every object symbol obj induces a family of partial attribute values $(obj.att)_{att \in St_C}$ (see definition 4).

[96] Data functions $f \in DFun$ are once determined by the initialisation state for all system states of an Object-Oriented Transformation System (of a data space signature).

- the composition $(j \circ h)$: $A \to D$
  of two system data state morphisms $h: A \to B$ and $j: B \to D$ is given by:

  $(j \circ h): A \to D = ((j_s \circ h_s)_{s \in (DS \cup Classes)})$

  $(j_s \circ h_s): A_s \to D_s$
  denotes the usual composition of functions between sets,
  corresponding with the composition of morphisms in category SET.

- the identity system data state morphism $id_A: A \to A$
  $id_A = ((id_{A,s})_{s \in (DS \cup Classes)}, id_{A,stack})$, where

  $id_{A,s}: A_s \to A_s$ is the identity map of data sort-carrier sets and object symbols of class s, respectively.

▽

4.5.5. Functors

This paragraph introduces the missing definitions that are needed to specify Object-Oriented Transformation Systems as a concrete institution (see paragraph 4.5.7.).

Fact 16 (Forgetful functor $Mod_{OOTS}(\sigma)$)

Given a data space signature morphism $\sigma: DSig_1 \to DSig_2$, where
$DSig_1 = (DS, DFun, AL, Classes)$.

The forgetful functor $Mod_{OOTS}(\sigma): Mod_{OOTS}(DSig_2) \to Mod_{OOTS}(DSig_1)$
on $DSig_2$-system data states and $DSig_2$-system data state morphisms is then defined by

- For every $DSig_2$-system data state A_2 there is a $DSig_1$-system state
  $A_1 := Mod_{OOTS}(\sigma)(A_2)$, with

  $A_1 = ((A_{1,s})_{s \in (DS \cup Classes)}, (static_{A_1,Cl})_{Cl \in Classes})$
  $A_{1,s} := V_\sigma(A_{2,s})_{s \in \sigma(DS \cup Classes)}$[97] $\quad \forall s \in (DS \cup Classes)$
  $static_{A_1,Cl} := V_\sigma(static_{A_2, \sigma(Cl)})$[98] $\quad \forall Cl \in Classes$

- For every $DSig_2$-system data state morpism $h_2: A_2 \to B_2$ there is a $DSig_1$-system data state morphism $h_1 := Mod_{OOTS}(\sigma)(h_2): Mod_{OOTS}(\sigma)(A_2) \to Mod_{OOTS}(\sigma)(B_2)$

  $h_1 := ((h_{1,s})_{s \in (DS \cup Classes)})$, with

  $h_{1,s} := V_\sigma(h_{2,\sigma(s)})$[99]

▽

---

[97] The carrier sets of A_1 are the forgetful carrier sets of A_2, based upon the forgetful functor V_σ known from algebras and sets (see [WKWC94]).

[98] Classes of $DSig_2$ that do not have a preimage w.r.t. σ in DSig will be forgotten.

[99] The homomorphismus h_1 is given by the forgetful functions of $h_{2,\sigma(s)}$, based on the forgetful functor from algebras and sets.

Fact 17 (The model functor Mod_{OOTS})

The model functor Mod_{OOTS} of this institution can now be defined as a functor
Mod_{OOTS}: $DSIG \rightarrow CAT^{op}$

- For every data state signature $DSig \in |DSIG|$, $Mod_{OOTS}(DSig)$ is the category of all DSig-system data states as defined in definition 15.
- For every data space signature morphism $\sigma: DSig_1 \rightarrow DSig_2 \in Mor(DSIG)$, $Mod_{OOTS}(\sigma)$: $Mod_{OOTS}(DSig_1) \rightarrow Mod_{OOTS}(DSig_2)$ is the forgetful functor as defined in definition 16.

▽

Proof

It has to be proven that Mod_{OOTS} preserves identity morphisms and the composition, which follows directly from definition 16.

□

Fact 18 (The sentence functor Sen_{OOTS})

Since Object-Oriented Transformation Systems do not define any equations for data space signatures the sentence functor maps data state signatures onto the empty sets of equations.

Eventually sentence functor Sen_{OOTS} of this institution is defined by
Sen: $DSIG \rightarrow SET$

- $Sen(DSig) = \varnothing \qquad \forall DSig \in |DSIG|$
- $Sen(\sigma) = \varnothing \qquad \forall \sigma \in Mor(DSIG)$

$\varnothing$ denotes the empty function $\varnothing: \varnothing \rightarrow \varnothing$.

▽

Proof

Sen obviously constitutes a functor.

□

Definition 19 (Satisfaction relation |=)

Since there are no sentences over data state signature, i.e. no equations, the satisfaction relation is the empty relation for all data state signatures.

$|=_{Dsig} \subset |Mod_{OOTS}(DSig)| \times Sen_{OOTS}(Dsig)$

- $|=_{DSig} = \varnothing \qquad \forall DSig \in |DSIG|$

◇

Fact 20 (The functor sorts)

The functor sorts maps a data state signature onto the set of its sorts, forgetting all structural information.

sorts: $DSIG \rightarrow SET$

- $sorts(DSig) = DS_{DSig} \cup Classes_{DSig} \qquad \forall DSig \in |DSIG|$
- $sorts(\sigma) = \sigma \qquad \forall \sigma \in Mor(DSIG)$

▽

Remark 20 (The functor sorts)

A data state signature is mapped onto the union of primitive data sorts and classes, differing from the data state signature's set Types. Since the latter can be derived from the sets DS and Classes (see definition 1), the functor sorts maps a data state signature onto its basic sorts.

A data state signature morphism σ is mapped identically. Since a data state signature morphism σ consists merely out of mappings between the carrier sets, it does not map any structural information. Though the typing of $\sigma \in Mor(SET)$ is different from $\sigma \in Mor(DSIG)$, it is the same family of functions.

Definition 21 (The natural transformation |_|)

The natural transformation |_| maps categories of data state signature-models (system data states) to categories of carrier sets; in short, system data states are mapped onto their carrier sets, system data state morphisms are mapped onto families of functions (between the carrier sets of two system data states).

$$|\_|: Mod_{OOTS} \Rightarrow SSet \circ sorts = (|\_|_{DSig}: Mod_{OOTS}(DSig) \rightarrow SET^{sorts(DSig)})_{DSig \in |DSIG|}$$

- $\forall DSig = (DS, DFun, AL, Classes) \in |DSIG|$:

  $|\_|_{DSig}: Mod_{OOTS}(DSig) \rightarrow SET^{sorts(DSig)}$, where

  $| A |_{DSig} = | ((A_s)_{s \in (DS \cup Classes)}, (static_{A,Cl})_{Cl \in Classes}) |_{DSig} = ((A_s)_{s \in (DS \cup Classes)})$

- $\forall \sigma: DSig \rightarrow DSig' \in Mor(DSIG)$:

  $|\_|_{DSig} \circ Mod_{OOTS}(\sigma) = Mod_{OOTS}(sorts(\sigma)) \circ |\_|_{DSig'}$

  $| h: A \rightarrow B |_{DSig} = (h'_s)_{s \in (DS \cup Classes)}$, where

  $h'_s: A_s \rightarrow B_s := h_s$

◇

Remark 21 (The natural transformation |_|)

The natural transformation |_| maps a system data state, i.e. a model to a given data state signature, onto its carrier sets, omitting structural information, in this case the static instances.

System data state morphisms between two system data states, i.e. two models of a given data state signature, are mapped onto a family of function between the carrier sets of the two system data states, assuming the functions of the system data state morphism. Since a system data state morphism merely constists of functions between carrier sets, only the typing is altered.

4.5.6. Institution of Object-Oriented Transformation Systems

This paragraph introduces the institution OOTS of Object-Oriented Transformation Systems. The preceding paragraphs introduced the necessary definitions (see paragraph 4.5.1.), based on Object-Oriented Transformation Systems, introduced by definition 1 through 8.

Fact 22 (Institution of Object-Oriented Transformation Systems OOTS)
The institution OOTS of Object-Oriented Transformation Systems is given by

OOTS = (DSIG, Mod_{OOTS}, Sen_{OOTS}, ∅, sorts, |_|), where

- DSIG is the category of data state signatures as specified in definition 12.
- Mod_{OOTS} is the model functor as specified in definition 17.
- Sen_{OOTS} is the sentence functor, mapping every data space signature onto the empty set of sentences as specified in definition 18.
- ∅ is the empty satisfaction relation (see definition 19)
- sorts is the functor sorts as specified in definition 20
- |_| is the natural transformation |_| as specified in definition 21

▽

Proof
It just has to be proven that the satisfaction condition holds. Since the satisfaction relation is the empty relation it always holds.

□

Corollary 23 (Specification framework of Object-Oriented Transformation Systems)
Given OOTS = (DSIG, Mod_{OOTS}, Sen_{OOTS}, ∅, sorts, |_|) , a concrete institution as defined in Fact 22. The corresponding specification framework is then given by

- A data space signature DSig' = (DSig, N)[100] w.r.t. OOTS is given by a signature DSig ∈ |DSIG| and an S* × S*-indexed set $N = (N_{w,w'})_{w,w' \in S^*}$, where S := sorts(DSig).
- An action structure AS w.r.t. OOTS is given by a family of functors

  $AS_{DSig}: SET^S \times SET^S \rightarrow SET$

  for each data space signature DSig', where S := sorts(DSig).

▽

Remark 23 (Specification framework of Object-Oriented Transformation Systems)
A "OOTS-specification framework for generic Transformation Systems" 𝕀F = (OOTS, AS, I, ε) is given by

- the concrete institution OOTS,
- an action structure AS w.r.t. OOTS,
- a family of designated DSig-models $I_{DSig} \in |Mod(DSig)|$ for each signature DSig, and
- a family of designated actions $\varepsilon_{DSig,A} \in AS_{DSig}(|A|,|A|)$ for each data space signature DSig' = (DSig, N) and each model $A \in |Mod(DSig)|$.

[100] The term "data space signature" is also used to define ordinary Transformation Systems.

4.5.7. Result

Section 4.5. has proven that data space signatures as defined in definition 1 and definition 2 and corresponding system states as defined in definition 4 can be extended to form a concrete institution.

Arbitrary concrete institutions can be extended to specification frameworks in order to instantiate generic Transformation Systems. The instantiation of generic Transformation Systems by arbitrary specification frameworks has been proven by M. Große-Rhode in section 2.3. of [Gro01].
Hence Object-Oriented Transformation Systems, building the institution OOTS of data space signatures and system states, are an instantiation of generic Transformation Systems.

This result enables Object-Oriented Transformation Systems and all its instances[101] to use all theorems that have been proven for generic Transformation Systems in general.
Particularly these theorems include the Composition of two Transformation Systems. Hence the composition of Object-Oriented Transformation Systems can applied as specified for generic Transformation Systems in section 5.10. of [Gro01].
Furthermore development relations, also introduced by M. Große Rhode in [Gro01], can be applied to Object-Oriented Transformation Systems. How the several development relations can be applied to instances of generic Transformation Systems, is specified in section 4.8. of [Gro01].

[101] I.e. every concrete Object-Oriented Transformation System.

4.6. Concurrent Object-Oriented Transformation Systems

The following section introduces "concurrent Object-Oriented Transformation Systems", an extension of Object-Oriented Transformation Systems adjusted to concurrent systems. Concurrent systems allow the simultaneous execution of multiple methods (threads), called concurrency.
Hence the main difference to "ordinary" Object-Oriented Transformation Systems is the need for a system states, having to track multiple threads instead of just one, i.e. there have to be more than one call stacks, corresponding to the number of simultaneous threads. In addition there have to be ways to invoke, terminate and co-ordinate the various threads. Co-ordinating threads is either done by synchronisation or message-exchange, i.e. communication between threads.
In order to realise the mentioned changes, the definition of classes will have to be altered and the set of actions has to be extended:

4.6.1. Changes to non-concurrent Object-Oriented Transformation Systems

Since a system state of concurrent Object-Oriented Transformation Systems describes a set of running methods instead of only one, the call-stack of a system state has to be replaced by a set of stacks. In order to initiate new threads a new method "run()" is added to the behaviour-signature $BehSig_C$ of classes $C \in$ Classes, becoming an obligatory method of every object. The run()-method starts a new parallel thread without making the method that invoked the run()-method loose responsibility of its own thread, i.e. the set of call stacks is extended by a new call stack. The latter necessitates the adjustment of Call-actions (definition 5(a)). On the other hand the Return-action of run()-methods denotes the end of a thread's execution and the according call stack has to be deleted.

Example
Let us assume method m1 of object o1 calls the run()-method of object o2. m1 does not loose responsibility and o1 remains the active object (of their respective thread) while a new thread is started with o2 as the active object. Both threads, i.e. both methods m1 and run, continue their execution, running simultaneously.

Parallel running methods can be synchronized: in order to co-ordinate threads, methods can lock objects that have not been locked before. Methods that want to lock an already locked object have to wait until the object is loosened again. Thus methods that depend on the locking of the same object can be synchronised. Eventually two new actions have to be introduced, locking and unlocking objects, respectively.
Finally a special attribute "locked" has to be added to the state signature $StSig_C$ of classes $C \in$ Classes, becoming an obligatory attribute of objects: the attribute value "obj.locked" then indicates if an object obj is locked (by any method). Its initial value is set to false.

4.6.2. Modified definitions

The following paragraphs present the new definitions, where all definitions of Object-Oriented Transformation Systems (definitions 1 – 8) are assumed as before and only their extensions are specified.

Example
*Definition 2^c introduces the extensions of definition 2 for concurrent Object-Oriented Transformation Systems, i.e. definition 2^c is equal to definition 2 except for the added method "run()" and the added attribute-symol "locked". The superscript-letter "c" is supposed to indicate the extension, concerning **c**oncurrent Object-Oriented Transformation Systems.*

All definitions that are not explicitly listed in this section, for instance definition $1^{(c)}$ and definition $3^{(c)}$, are taken over from ordinary Object-Oriented Transformation Systems without modification.
There is also a new definition 9, specifying Synchronisation-actions. Methods can lock and unlock accessible objects, in order to co-ordinate their execution with other methods that need to lock the same objects. Definition 1 through 9, along with the following modifications, introduce concurrent Object-Oriented Transformation Systems.
Since all modifications are denoted by extensions of the original definitions, "ordinary" Object-Oriented Transformation Systems constitute a special case of concurrent ones, having only one call stack instead of the set of stacks, no special "run()"-method, and no special "locked"-attribute.

Definition 2^c (Class Signature)

Every class symbol $C \in Classes$ induces a class signature CSig, given by

$CSig = (Super_C, AC_C, StSig_C, \mathbf{BehSig_C}, ClStSig_C, ClBehSig_C)$, with

- The definitions of $Super_C$, AC_C, $StSig_C$, $ClStSig_C$, and $ClBehSig_C$ remain unaltered
- Behaviour Signature $\mathbf{BehSig_C} = (BehSig_{C,al,args,type})_{al \in AL,\ args,type \in Types^*}$:
  the definition of a classes' behaviour signature remains like in the original definition 2.
  There is a new obligatory method run() that every class C has to contain. When invoked, run() does not only take responsibility of the thread, but starts a new parallel thread (see definition 5^c(a)):

  OTCrun: $run \in BehSig_{C,al,void,void}$, where $al \in AL$

◇

Remark 2 (Class Signature)

The only change to the original definition of class signatures is the new obligatory method "run". As a result every object has to contain a run() method, starting a new thread and taking responsibility of it.
However, a method, invoking the special run()-method, does no lose responsibility of its own thread and continues execution, while run() is executed simultaneously. In doing so it becomes the start-up method of the new thread. Eventually the new thread is terminated when the run() method finishes execution, just like an "ordinary" Object-Oriented Transformation Systems terminates, when its start-up method finishes.

Definition 4^c (Data State / System State)

$A_{DSig} = ((A_s)_{s \in (DS \cup Classes)}, (fun_A)_{fun \in DFun}, (static_{A,C})_{C \in Classes}, \mathbf{stackSet_A})$

- Carrier sets $(A_s)_{s \in (DS \cup Classes)}$

  The carrier sets $(A_s)_{s \in DS}$ remain unaltered.
  The definition of instance sets, Domains and the carrier sets $(\mathbf{A_s})_{s \in Classes}$ remain unaltered.

  The induced families of partial attribute values will be extended by a new obligatory attribute "**locked**" that is used to synchronise methods: the family of partial attribute values belonging to an object obj, is extended by obj.locked: $\rightarrow$ bool. obj.locked is set to false in the beginning of an object's life cycle and it can only be changed by synchronisation actions (see definition 9).

  OTClocked: obj.locked $\in A_{bool}$ $\forall$obj $\in A_C$, $\forall C \in$ Classes

- Data functions $(f_A)_{f \in DFun}$ and the "static" parts of Classes $(static_{A,C})_{C \in Classes}$ remain like in definition 4.

- Call Stacks $\mathbf{stackSet_A} = \{ stack_A \mid stack_A \in (Meth \times VType_A \times VValue_A)^* \}$:
  a set of call stacks, where every call stack belongs to one of several running threads, denoting which method is running (and active) and which methods are waiting for others to finish.
  The definitions of Meth, $VType_A$, $VValue_A$ and eval remain unchanged.

◇

Remark 4^c (Data State / System State)

There are not a lot of changes between definition 4^c and definition 4; eventually definition 4 is a special case of definition 4^c, where the set $stackSet_A$ has at most one element.
The initialisation states remain as defined in remark 4. Instead of just a single call stack there is a set (of call stacks), containing exactly one stack:

$\mathbf{Init_{DSig}} := \{ I \mid I = ((I_s)_{s \in (DS \cup Classes)}, (f_I)_{f \in DFun}, (static_{I,C})_{C \in Classes}, \{ (\tau,\{(self,type)\},\varnothing)) \} \}$
$I_s = \varnothing \quad \forall s \in Classes$
$static_{I,C}.catt = null \quad \forall C \in Classes, \forall catt \in ClStSig_C$
$type \in Classes$

The remaining structure remains unaltered. This means that definition 1 and definition 3 remain the same for concurrent Object-Oriented Transformation Systems as for "ordinary" Object-Oriented Transformation Systems. However there are several changes to the definition of actions. Since system states now contain sets of call stacks instead of single stacks and there is a new attribute value "locked", actions, the transformations between system states, have to be adjusted.
Since only synchronisation-actions can change the value of the new attribute "locked", all other actions do not change the attribute's value.

Definition 5^c (Call-Actions)

(**a**) Call-actions for (instance) methods al_type_meth($s_1 \ldots sn$) $\in$ $BehSig_C$:

$\underline{call(\ t_0.meth(t1,\ldots,tn)\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_A$ = st.(m, vtype, vvalue)

$$stack_B = \begin{cases} (meth,\ vtype^2,\ vvalue^2) & \text{, for meth = run} \\ st.(m,\ vtype, vvalue).(meth,\ vtype^2, vvalue^2) & \text{, otherwise} \end{cases}$$

$$stackSet_B = \begin{cases} stackSet_A \uplus stack_B & \text{, for meth = run} \\ (stackSet_A \setminus stack_A) \uplus stack_B & \text{, otherwise} \end{cases}$$

2) $vtype(self) \in AC_{C,al}$

3) $eval(t_0) \in AI_C$
$t_0.exists = true$

4) $eval(t_i) \in VDom(s_i)$ $\quad \forall i \in [1,n]$

5) $X^2 = \{self\} \cup \{arg_i \mid i \leq n \ \wedge\ i \in \mathbb{N}^+\}$

6) $vtype^2(self) = C$
$vvalue^2(self) = vvalue(t_0)$

7) $vtype^2(arg_i) = s_i$ $\quad \forall i \in [1,n]$
$vvalue^2(arg_i) = eval(t_i)$ $\quad \forall i \in [1,n]$

0) $B_{Cl} = A_{Cl}$ $\quad \forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl}$ $\quad \forall Cl \in Classes$

constraint1: $acc_A(t_0) = T$
constraint2: $acc_A(t_i) = T$ $\quad \forall i \in [1,n]$

◇

(b) Call-actions for class methods al_type_cmeth($s_1 \ldots s_n$) $\in$ $ClBehSig_C$:

$\underline{call(\ C.cmeth(t_1,\ldots,t_n)\ )\ \in\ Act_{DSig}(A,B)}$, if

1) $stack_A$ = st.(m, vtype, vvalue)
$stack_B$ = st.(m, vtype, vvalue).(cmeth, $vtype^2$, $vvalue^2$)
$stackSet_B = (stackSet_A \setminus stack_A) \uplus stack_B$

2) $vtype(self) \in AC_{C,Class}$
$vtype(self) \in AC_{C,al}$

3) $eval(t_i) \in VDom(s_i)$ $\quad \forall i \in [1,n]$

4) $X^2 = \{self\} \cup \{arg_i \mid i \leq n \ \wedge\ i \in \mathbb{N}^+\}$

5) $vtype^2(self) = C$

$$vvalue^2(self) = \begin{cases} obj & \text{, for cmeth = constructor} \\ undefined & \text{, otherwise} \end{cases}$$

6) $obj.exists = eval_{bool}(true)$
$obj.locked = eval_{bool}(false)$
$obj.att$ undefined $\forall att \in StSig_{C,s}, \forall s \in DS$
$obj.att = null$ $\forall att \in StSig_{C,Cl}, \forall Cl \in (Types \setminus DS)$

7) $vtype^2(arg_i) = s_i$ $\forall i \in [1,n]$
$vvalue^2(arg_i) = eval(t_i)$ $\forall i \in [1,n]$

0) $B_C = \begin{cases} A_C \uplus \{obj\} & \text{, for cmeth = constructor} \\ A_C & \text{, otherwise} \end{cases}$
$B_{Cl} = A_{Cl}$ $\forall Cl \in (Classes \setminus \{C\})$
$static_{B,Cl} = static_{A,Cl}$ $\forall Cl \in Classes$

constraint1: $acc_A(t_i) = T$ $\forall i \in [1,n]$

◇

Remark 5^c (Call-Actions)

Definition 5^c resembles with the original definition 5 except for the treatment of the new obligatory and special method run that adds a new call stack (thread) to the set of stacks.

The definition 5(b) has to be modified for the case that the invoked class method is the special constructor method (condition 6) only applies in this case): since objects now contain the obligatory attribute "locked", it has to be set to its initial value (false). In order to conform to the modified definition of system states, also condition 1) is slightly modified.

Definition 6^c (Return-Actions)

(**a**) Return-actions for methods $al\_s_1 \ldots sn\_meth(args) \in BehSig_C$:

$\underline{return(\ meth.return(t_1,\ldots,t_n)\ ) \in Act_{DSig}(A,B)}$, if

1) $stack_A = \begin{cases} (meth, vtype^2, vvalue^2) & \text{, for meth = run} \\ st.(m, vtype, vvalue).(meth, vtype^2, vvalue^2) & \text{, otherwise} \end{cases}$

$stack_B = \begin{cases} \lambda & \text{, for meth = run } ^{102} \\ st.(m, vtype', vvalue') & \text{, otherwise} \end{cases}$

$stackSet_B = \begin{cases} stackSet_A \setminus stack_B & \text{, for meth = run} \\ (stackSet_A \setminus stack_A) \uplus stack_B & \text{, otherwise} \end{cases}$

2) $eval^2(t_i) \in VDom(s_i)$ $\forall i \in [1,n]$

3) $X' = X \cup \{ret_{(q+i)} \mid \forall i \in [1,n]\}$
$\forall i \in [1,n]$:
$vtype' = vtype \cup \{(ret_{(q+i)}, s_i)\}$
$vvalue' = vvalue \cup \{(ret_{(q+i)}, eval^2(t_i))\}$, where
q denotes the number of collected return values

0) $B_{Cl} = A_{Cl}$ $\forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl}$ $\forall Cl \in Classes$

constraint1: $acc_A(t_i) = T$ $\forall i \in [1,n]$

[102] λ denotes the empty (call) stack.

◇

(**b**) Return-actions for class methods al_s_1…sn_cmeth(args) ∈ $ClBehSig_C$:

return(cmeth.return(t_1,…,t_n)) ∈ $Act_{DSig}(A,B)$, if

1) $stack_A$ = st.(m, vtype, vvalue) .(cmeth, vtype², vvalue²)
   $stack_B$ = st.(m, vtype', vvalue')
   $stackSet_B$ = ($stackSet_A$ \ $stack_A$) ⊎ $stack_B$

2) $eval²(t_i) \in VDom(s_i)$ $\forall i \in [1,n]$

3) X' = X ∪ {$ret_{(q+i)}$ | i ∈ [1,n]}
   ∀i ∈ [1,n]:
   vtype' = vtype ∪ {($ret_{(q+i)}$, s_i)}
   vvalue' = vvalue ∪ {($ret_{(q+i)}$, $eval²(t_i)$)}, where
   q denotes the number of collected return values

0) $B_{Cl} = A_{Cl}$ $\forall Cl \in Classes$
   $static_{B,Cl} = static_{A,Cl}$ $\forall Cl \in Classes$

constraint1: $acc_A(t_i)$ = T $\forall i \in [1,n]$
constraint2: (cmeth = constructor) ⇒ (t_1 = self)

◇

Remark 6^c (Return-Actions)

Definition 6^c resembles with the original definition 6 except for the treatment of method run, finishing execution: as the calling of the special run()-method always causes the creation of a new call stack (thread), the thread's call stack is empty when the special run()-method finishes execution.
The adjusted definition 6^c removes empty call stacks from the set of call stacks, denoting the termination of the corresponding thread. Nevertheless the set of call stacks will never be empty as the call stack including the system's start-up method τ will not be removed by this.

The definition 6(b) of Return-actions for class methods virtually remains unaltered. In order to conform to the modified definition of system states, condition 1) is modified.

Definition 7^c (Assignment-Actions)

(**a**) Assignment-actions for (instance-) attributes al_type_att ∈ $StSig_C$:

assign(t_0.att, t_1) ∈ $Act_{DSig}(A,B)$, if

1) $stack_B = stack_A$

2) $acc_A(t_0.att)$ = T

3) $eval(t_0) \in AI_C$
   t_0.exists = true
   $eval_C(t_0)$ =: obj
   $eval'_C(t_0)$ =: obj'

4) $eval(t_1) \in VDom(type)$

5) obj'.att = $eval_{type}(t_1)$
   obj'.a = obj.a $\forall a \in StSig_C \setminus \{att\}$
   obj'.exists = obj.exists

obj'.locked = obj.locked

0) $B_{Cl} = A_{Cl}$ $\forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl}$ $\forall Cl \in Classes$

constraint1: $acc_A(t_0) = T$
constraint2: $acc_A(t_1) = T$

◇

(**c**) Creation-actions for local variables $ret_i \in X$:

assign(val) $\in$ $Act_{DSig}(A,B)$, if

1) $stack_A = st.(m, vtype, vvalue)$
$stack_B = st.(m, vtype', vvalue')$
$stackSet_B = (stackSet_A \setminus stack_A) \uplus stack_B$

2) $s \in DS$
$eval_s(val) \in A_s$

3) $X' = X \cup \{ret_{(q+1)}\}$
$vtype' = vtype \cup \{(ret_{(q+1)}, s)\}$
$vvalue' = vvalue \cup \{(ret_{(q+1)}, val)\}$, where
q denotes the number of collected return values

0) $B_{Cl} = A_{Cl}$ $\forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl}$ $\forall Cl \in Classes$

◇

(**d**) Assignment-actions for local variables $ret_i \in X$:

assign(ret_i, t_1) $\in$ $Act_{DSig}(A,B)$, if

1) $stack_A = st.(m, vtype, vvalue)$
$stack_B = st.(m, vtype', vvalue')$
$vtype' = vtype$
$stackSet_B = (stackSet_A \setminus stack_A) \uplus stack_B$

2) $ret_i \in X \setminus (\{self\} \cup \{arg_i \mid i \in \mathbb{N}^+\})$

3) $eval(t_1) \in VDom(vtype(ret_i))$

4) $X' = X$
$vvalue'(ret_i) = eval_{type}(t_1)$
$vvalue'(var) = vvalue(var)$ $\forall var \in X \setminus \{ret_i\}$

0) $B_{Cl} = A_{Cl}$ $\forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl}$ $\forall Cl \in Classes$

constraint1: $acc_A(t_1) = T$

◇

Remark 7^c (Assignment-Actions)

The only difference between the original definition 7(a) and definition 7^c(a) is the extension of condition 5), concerning the new obligatory attribute value "locked".

The definitions 7(b) remains unaltered, since the respective Assignment action has no effect either on the call stack(s) of the system's state or on any object's locked-attribute; condition 1) may be extended by $StackSet_B = StackSet_A$.

The definition 7(c) and definition 7(d) virtually remain unaltered, since only condition 1) is slightly modified, in order to conform to the modified definition of system states.

Definition 8^c (Other Actions)

(**b**) Finalisation-actions:

$\underline{finalise()} \in Act_{DSig}(A,B)$, if

1) $stack_A = (\tau, vtype^2, vvalue^2)$
$stack_B = \lambda$
$stackSet_B = (stackSet_A \setminus stack_A) \uplus stack_B$

0) $B_{Cl} = A_{Cl} \quad \forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl} \quad \forall Cl \in Classes$

◇

Remark 8^c (Assignment-Actions)

The definitions 8(a) remains unaltered, since the respective Assignment action has no effect either on the call stack(s) of the system's state or on any object's locked-attribute; condition 1) may be extended by $StackSet_B = StackSet_A$.

The definition 8(b) virtually remain unaltered, since only condition 1) is slightly modified, in order to conform to the modified definition of system states.
However the meaning of this action clearly changes: in "ordinary" Object-Oriented Transformation Systems, where only thread is running, this action is equivalent to the termination of whole system.
In concurrent Object-Oriented Transformation Systems however, it may be that another thread is still running and thus the system does not terminate, before all other threads of $stackSet_B$ have finished execution, i.e. before their call stacks have been deleted from the set of call stacks.

Definition 9 (Synchronizing Actions)

(**a**) Locking-action of objects $obj \in A_C$, with $C \in Classes$:

$\underline{lock(\ t_0\ )} \in Act_{DSig}(A,B)$, if

1) $stackSet_B = stackSet_A$

2) $eval(t_0) \in AI_C$
$t_0.exists = true$
$eval_C(t_0) =: obj$
$eval'_C(t_0) =: obj'$

[3] $obj.locked = eval_{bool}(false)$
$obj'.locked = eval_{bool}(true)$
$obj'.a = obj.a \quad \forall a \in StSig_C$
$obj'.exists = obj.exists$

[0] $B_{Cl} = A_{Cl} \quad \forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl} \quad \forall Cl \in Classes$

constraint1: $acc_A(t_0) = T$

◇

(**b**) Unlocking-action of objects $obj \in A_C$, with $C \in Classes$:

unlock(t_0) $\in$ $Act_{DSig}(A,B)$, if

[1] $stackSet_B = stackSet_A$

[2] $eval(t_0) \in AI_C$
$eval_C(t_0) =: obj$
$eval'_C(t_0) =: obj'$

[3] $obj.locked = eval_{bool}(true)$
$obj'.locked = eval_{bool}(false)$
$obj'.a = obj.a \quad \forall a \in StSig_C$
$obj'.exists = obj.exists$

[0] $B_{Cl} = A_{Cl} \quad \forall Cl \in Classes$
$static_{B,Cl} = static_{A,Cl} \quad \forall Cl \in Classes$

constraint1: $acc_A(t_0) = T$

◇

Remark 9 (Synchronizing Actions)

(**a**) A Locking-action **lock(t_0)** changes the system's state from state A to B, i.e. the object, denoted by $eval(t_0)$, is locked (by the active method). As a result no other method can lock the object obj. Other threads that need to lock t_0 need to wait until t_0 is loosened (unlocked) again.

Example
The programming language Java uses the keyword 'synchronized' to lock objects and thus coordinate simultaneous methods, e.g. "synchronized(t0) { <statements> }".
Synchronised methods lock the instance they belong to, e.g. "public synchronized void sync_method() { <method body> }".

Normally only the method that locked an object can unlock it. The latter is not represented by the actions of definition 9 and thus has to be expressed, i.e. restricted, by the control graph of the Object-Oriented Transformation System, if wanted.

The following condition specifies the transformation(s) made by **lock(t_0)**:

[3] A method can only lock an object obj, if it was not locked before, i.e. if obj has already been locked, the method has to wait for obj to be loosened, before it can lock it again and continue execution.
obj denotes the state of the object that is locked in system state A, obj' denotes its state in B: $obj \in A_C$, $obj' \in B_C$. On the one hand obj and obj' denote the same object symbol (obj = obj'), but obj induces a family of partial attribute values in A and obj' induces a family of partial attribute values in B (see definition 4).

(**b**) A Unlocking-action **unlock(t_0)** changes the system's state from state A to B, i.e. the locked object obj, denoted by eval(t_0), is freed (loosened). Hence a method, waiting for t_0 to be unlocked, can now lock it and continue its execution.

The following condition specifies the transformation(s) made by **lock(t_0)**:

3) A method can only unlock an object obj, if it was locked before.

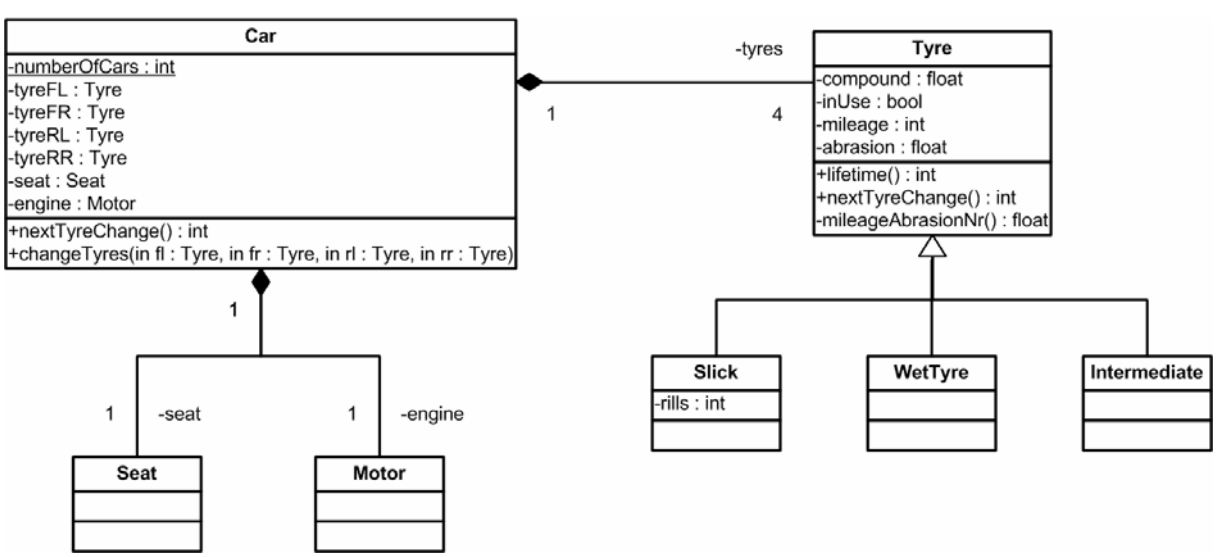
Chapter 5

Example

The purpose of this chapter is to construct a complete Object-Oriented Transformation System as defined in chapter 4 to support perception and comprehension of Object-Oriented Transformation Systems. The latter will be depicted by both textual representations as in the definitions of chapter 4 and figurative diagrams. The example specifications are given mainly by diagrams of the UML and intuition.
Finally the example Object-Oriented Transformation System should assist the validation if the presented definitions implement all object-oriented characteristics as identified in section 2.3.

5.1. Example

The following UML class diagram depicts the running example of this paper, containing a Car class that aggregates a seat, an engine and 4 tyres, whereas Tyre is the generalisation of the types Slick, WetTyre and Intermediate:



The following section constructs a concrete Object-Oriented Transformation System to validate the implementation of object-oriented characteristics as identified in section 2.3 on the one hand and to facilitate a more intuitive understanding of Object-Oriented Transformation Systems on the other.

5.1.1. Creating a Data Space Signature to our example

DSig = (DS, DFun, AL, Classes), with

DS := {bool, float, int}
DFun := (+: float float → float, +: int int → int, }
AL := {private, public}
Classes := {Object, Car, Tyre, Slick, WetTyre, Intermediate, Seat, Motor}, where

Car := ($Super_{Car}$, AC_{Car}, $StSig_{Car}$, $BehSig_{Car}$, $ClStSig_{Car}$, $ClBehSig_{Car}$), with

$Super_{Car}$:= {Object}
⇒ $Ancestors_{Car}$ = {Object}
$AC_{Car,private}$:= {Car}
$AC_{Car,public}$:= Classes
St_{Car} := {Tyre_tyreFL, Tyre_tyreFR, Tyre_tyreRL, Tyre_tyreRR, Motor_engine, Seat_seat}
Beh_{Car} := {public_void_changeTyres(Tyre Tyre Tyre Tyre),
public_float_nextTyreChange()}
$ClSt_{Car}$:= {int_numberOfCars}
$ClBeh_{Car}$:= {public_Car_constructor()}

Tyre := ($Super_{Tyre}$,AC_{Tyre}, $StSig_{Tyre}$, $BehSig_{Tyre}$, $ClStSig_{Tyre}$, $ClBehSig_{Tyre}$), with

$Super_{Tyre}$:= {Object}
⇒ $Ancestors_{Tyre}$ = {Object}
$AC_{Tyre,private}$:= {Tyre}
$AC_{Tyre,public}$:= Classes
St_{Tyre} := {float_compoundHardness, bool_inUse, int_mileage, float_abrasion}
Beh_{Tyre} := {public_int_lifetime(int), private_int_mileageAbrasionNr(),
public_float_nextTyreChange(), public_void_setInUse(bool)}
$ClSt_{Tyre}$:= {}
$ClBeh_{Tyre}$:= {public_Tyre_constructor()}

Slick := ($Super_{Slick}$,AC_{Slick}, $StSig_{Slick}$, $BehSig_{Slick}$, $ClStSig_{Slick}$, $ClBehSig_{Slick}$), with

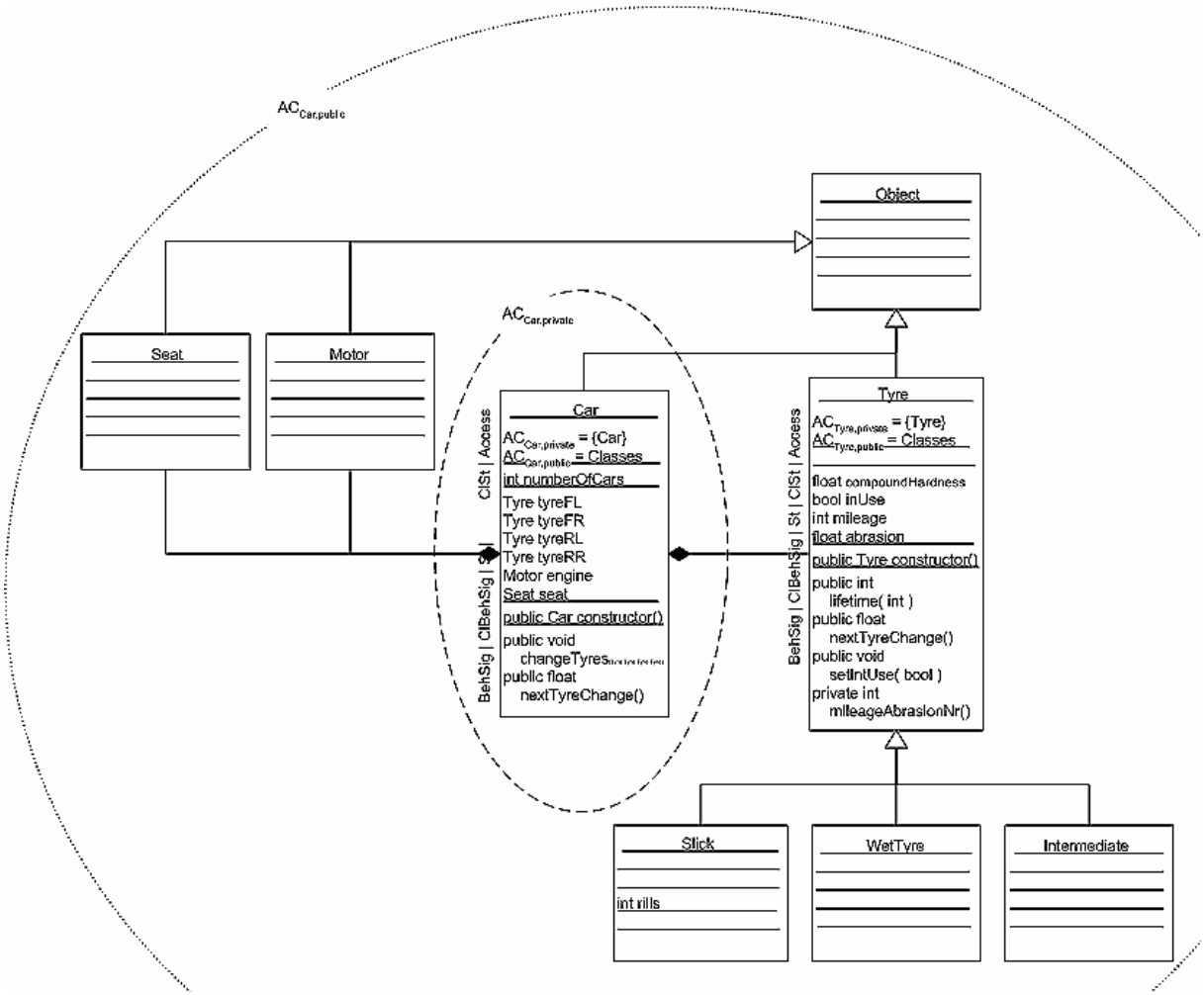
$Super_{Slick}$:= {Tyre}
⇒ $Ancestors_{Slick}$ = {Tyre, Object}
$AC_{Slick,private}$:= {Slick}
$AC_{Slick,public}$:= Classes
St_{Slick} := St_{Tyre} ∪ {int_rills}
Beh_{Slick} := Beh_{Tyre} ∪ {}
$ClSt_{Slick}$:= {}
$ClBeh_{Slick}$:= {public_Slick_constructor()}

$WetTyre := (Super_{WetTyre}, AC_{WetTyre}, StSig_{WetTyre}, BehSig_{WetTyre}, ClStSig_{WetTyre}, ClBehSig_{WetTyre})$, with
$Super_{WetTyre} := \{Tyre\}$
$\Rightarrow Ancestors_{WetTyre} = \{Tyre, Object\}$
$AC_{WetTyre,private} := \{WetTyre\}$
$AC_{WetTyre,public} := Classes$
$St_{WetTyre} := St_{Tyre} \cup \{\}$
$Beh_{WetTyre} := Beh_{Tyre} \cup \{\}$
$ClSt_{WetTyre} := \{\}$
$ClBeh_{WetTyre} := \{public\_WetTyre\_constructor()\}$

$Intermediate := (Super_{Intermediate}, AC_{Intermediate}, StSig_{Intermediate}, BehSig_{Intermediate}, ClStSig_{Intermediate}, ClBehSig_{Intermediate})$, with
$Super_{Intermediate} := \{Tyre\}$
$\Rightarrow Ancestors_{Intermediate} = \{Tyre, Object\}$
$AC_{Intermediate,private} := \{Intermediate\}$
$AC_{Intermediate,public} := Classes$
$St_{Intermediate} := St_{Tyre} \cup \{\}$
$Beh_{Intermediate} := Beh_{Tyre} \cup \{\}$
$ClSt_{Intermediate} := \{\}$
$ClBeh_{Intermediate} := \{public\_Intermediate\_constructor()\}$

The following diagram[103] presents a figurative depiction of the defined data space signature DSig. The diagram still contains the UML connectors for generalisation and aggregation: generalisation edges depict the set Parents of every class. The association edges are implicitly expressed by attributes whose type is a class itself.
In opposition to the UML class diagram of above the following data space signature diagram shows a new class Object that is the root class of every Object-Oriented Transformation System. Also depicted in the diagram are the access classes $AC_{Car,private}$ and $AC_{Car,public}$ of class Car that describe the meaning of access levels private and public w.r.t class Car.[104]
The major difference between the two diagrams is that UML classes consist of 3 parts (class name, state attribute symbols, behaviour) and classes of a data space signature diagram of 6 parts. The three supervening parts are implicitly expressed in UML class diagrams: access classes can not be defined in the UML as the set of possible access levels fixed for the entire UML and also the meant classes can not be adjusted individually. Data space signature diagrams need to express access classes, either textually or figuratively (see diagram below).
Data space signatures also differ between class methods (class behaviour ClBeh) and instance methods (behaviour Beh), which are mixed in a UML class behaviour. That also applies to class- and instance attribute symbols.

[103] Herein after referred to as "data space signature diagram".
[104] Access classes do not need to be subsets of each other (being the case in diagramX).



5.1.2. A system state may look like

The following system state, denoted by A_{DSig} or simply A, will be introduced extensively detailed. However the definition will not be complete, e.g. not all objects (and their states) will be specified in detail as their state resembles the one of described objects.

$A_{DSig} = ((A_s)_{s \in (DS \cup Classes)}, (fun_A)_{fun \in DFun}, (static_{A,C})_{C \in Classes}, stack_A)$, with

- Carrier Sets

  $A_{bool} := \{true, false\}$
  $A_{int} := Z$
  $A_{float} := Q$
  $A_{Object} := \varnothing$
  $A_{Car} := \{ferrari\}$
  $A_{Tyre} := \varnothing$
  $A_{Slick} := \{bs01, bs02, bs03, bs04\}$
  $A_{WetTyre} := \{wet01, wet02, wet03, wet04\}$
  $A_{Intermediate} := \{im01, im02, im03, im04\}$
  $A_{Motor} := \{f700ps, f800ps\}$
  $A_{Seat} := \{seat\_ms, seat\_rb\}$
  $\Rightarrow AI_{Tyre} = \{bs01,bs02,bs03,bs04,wet01,wet02,wet03,wet04,im01,im02,im03,im04\}$

where

ferrari := (ferrari.tyreFL = bs01,
ferrari.tyreFR = bs02,
ferrari.tyreRL = bs03,
ferrari.tyreRR = bs04,
ferrari.engine = f700ps,
ferrari.seat = seat_ms)

bs01 := (bs01.compoundHardness = 0.8,
bs01.inUse = true,
bs01.mileage = 400,
bs01.abrasion = 0.85
bs01.rills = 4)

wet01 := (wet01.compoundHardness = 0.4,
wet01.inUse = false,
wet01.mileage = 0,
wet01.abrasion = 1.0)

im01 := (im01.compoundHardness = 0.6,
im01.inUse = false,
im01.mileage = 10,
im01.abrasion = 0.95)

- Functions

  $+_A: A_{float} \rightarrow A_{float}$
  $+_A(a,b) = a+b$

  $+_A: A_{int} \rightarrow A_{int}$
  $+_A(a,b) = a+b$

- "Static" parts of Classes

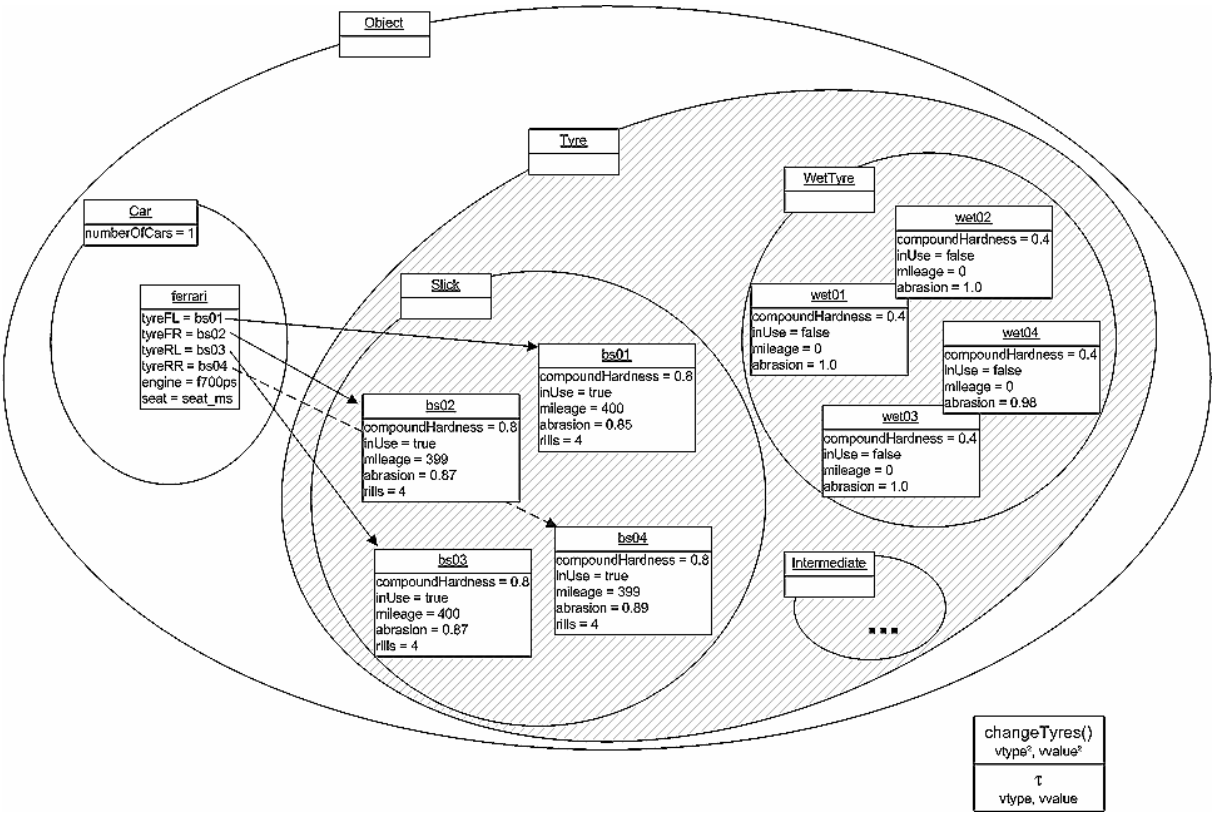
  $static_{A,Car} = (static_{A,Car}.numberOfCars = 1)$
  $static_{A,C} = \emptyset \qquad \forall C \in (Classes \setminus \{Car\})$

- Call Stacks

  $stack_A = (\tau, vtype, vvalue)$, with

  $X = \{self, ret1, ret2, \ldots, ret17\}$
  $vtype: X \rightarrow (DS \cup Classes)$
  vtype = {(self,Car), (ret1,Slick), (ret2,Slick), (ret3,Slick), (ret4,Slick), (ret5,WetTyre), (ret6,WetTyre), (ret7,WetTyre), (ret8,WetTyre), (ret9,Intermediate), (ret10,Intermediate), (ret11,Intermediate), (ret12,Intermediate), (ret13,Motor), (ret14,Motor), (ret15,Seat), (ret16,Seat), (ret17,Car)}
  $vvalue: X \rightarrow A_{DSig}$
  vvalue = {(ret1,bs01), (ret2,bs02), (ret3,bs03), (ret4,bs04), (ret5,wet01), (ret6,wet02), (ret7,wet03), (ret8,wet04), (ret9,im01), (ret10,im02), (ret11,im03), (ret12,im04), (ret13,f700ps), (ret14,f800ps), (ret15,seat_ms), (ret16,seat_rb), (ret17,ferrari)}

The following diagram[105] presents a figurative depiction of the defined system state A (the call stack is altered to assist a figurative notion of a stack).The diagram mainly consists of ellipses denoting the object sets to classes. More precisely they depict a state's instance sets, containing all objects of the class and all subclasses. For instance the dashed area depicts the instance set of Tyre, containing all objects of subclasses Slick, WetTyre and Intermediate. At the border of these sets the so-called static instances are located, denoting the class' state, i.e. the values of all class attributes[106].



5.1.3. Dynamics / Sequence of actions

Now both the structure of the system and concrete system states can be represented. To introduce dynamics let us assume it starts raining and our (formula 1) car needs to exchange its tyres, i.e. his dry-weather Slick-tyres against wet tyres. To do so we call the suitable method "changeTyres()" on the existing Car-object "ferrari". Methods are not described by data states, but by sequences of transformations, i.e. the transformations between Call- and Return-action of a method.

The following segment lists all actions executed by method changeTyres() on object ferrari. Within action-calls concrete objects by denoted by syntactical terms, e.g. the variable "self" in transformations 2 through 15 denotes the concrete object ferrari.

[105] Herein after referred to as "system state diagram".

[106] Class attributes are known as "static attributes" in programming languages like Java or C++.

1. call(ret17.changeTyres(ret5,ret6,ret7,ret8)) $\in$ $Act_{DSig}(A,B)$

2. assign(false) $\in$ $Act_{DSig}(B,C)$
3. assign(true) $\in$ $Act_{DSig}(C,D)$

4. call(self.tyreFL.setInUse(ret_1)) $\in$ $Act_{DSig}(D,E)$
   4a. assign(self.inUse, arg1) $\in$ $Act_{DSig}(E, E^1)$
   4b. return(setInUse.return()) $\in$ $Act_{DSig}(E^1,E^2)$
5. assign(self.tyreFL, arg_1) $\in$ $Act_{DSig}(E^2,F)$
6. call(self.tyreFL.setInUse(ret_2)) $\in$ $Act_{DSig}(F,G)$

7. call(self.tyreFR.setInUse(ret_1)) $\in$ $Act_{DSig}(G,H)$
8. assign(self.tyreFR, arg_2) $\in$ $Act_{DSig}(H,I)$
9. call(self.tyreFR.setInUse(ret_2)) $\in$ $Act_{DSig}(I,J)$

10. call(self.tyreRL.setInUse(ret_1)) $\in$ $Act_{DSig}(J,K)$
11. assign(self.tyreRL, arg_3) $\in$ $Act_{DSig}(K,L)$
12. call(self.tyreRL.setInUse(ret_2)) $\in$ $Act_{DSig}(L,M)$

13. call(self.tyreRR.setInUse(ret_1)) $\in$ $Act_{DSig}(M,N)$
14. assign(self.tyreRR, arg_4) $\in$ $Act_{DSig}(N,O)$
15. call(self.tyreRR.setInUse(ret_2)) $\in$ $Act_{DSig}(O,P)$

16. return(changeTyres.return()) $\in$ $Act_{DSig}(P,Q)$

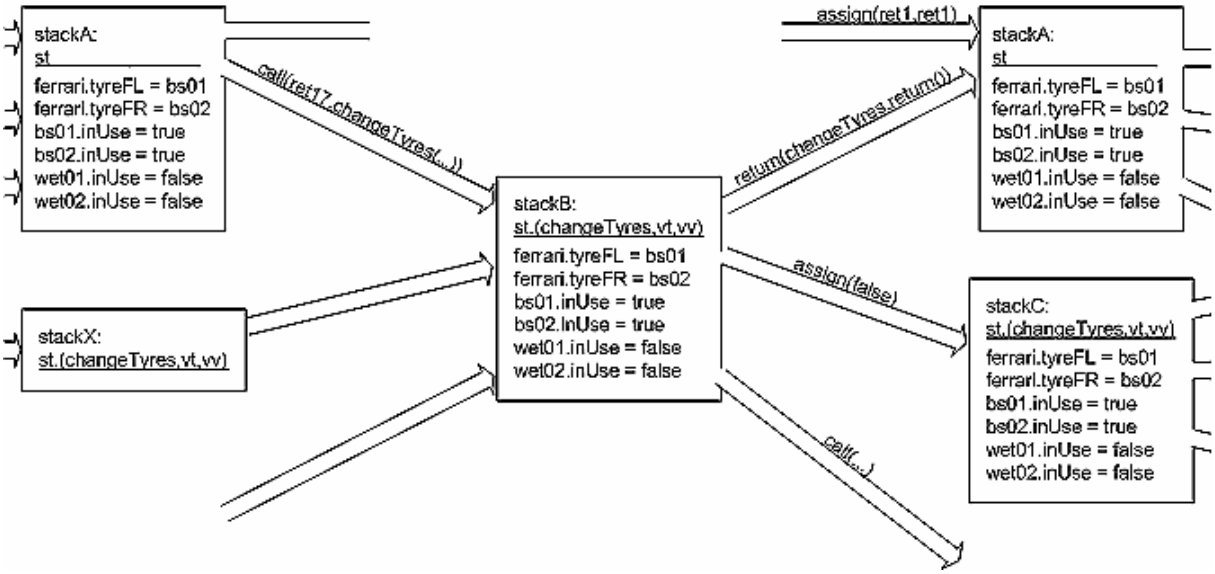
The previous sequence of actions describes the behaviour of method changeTyres(). Assuming state A is the state described before, τ is the method active in state A and Q. The described method changeTyres() is active between states B and P, lying on top of the call stack of these states and having its own environment. Thus local variables like ret_1 denote different values in state A and B.
Basically the method performs 4 steps: replacing the 4 tyres of the car with the delivered "new" tyres (denoted by input parameters ret5, ret6, ret7, ret8). In order to replace a tyre the method sets the attribute inUse of the "old" tyre to false, alters the respective attribute of object ferrari and sets the attribute inUse of the "new" tyre to true. To set the attribute inUse the method setInUse is called on each tyre, including at least a Call, an Assignment, and a Return-action as expressed by transformations 4, 4a, 4b. All other calls of setInUse are denoted by just one transformation (compare transformation 6).

Before we represent method changeTyres() and its behaviour by the defined Object-Oriented Transformation System I want to introduce a Transformation System's control graph and its relation to the data space (once more):

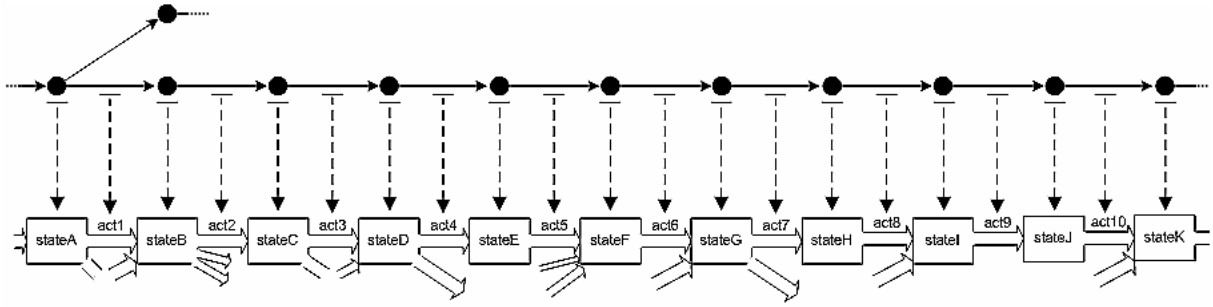
Every system state can perform several actions and can be reached from several other states. Without a mechanism, controlling which actions may follow on each other the possible executable number of actions for a system state can hardly be controlled, not to mention expressed in a diagram. Yet a very small section of the data space shows a good deal of connections between data states.

Example
Theoretically in a system state B, i.e. right after method changeTyres() has been invoked (see above), the Return-action return(changeTyres.return()) can lead back to state A (without making any changes to the system's state. The latter can not be intended by method changeTyres() and thus does not make sense.

The following diagram should visualise a small section of the data space around state B, where dangling transformations represent the connection to other state, not shown in the diagram:



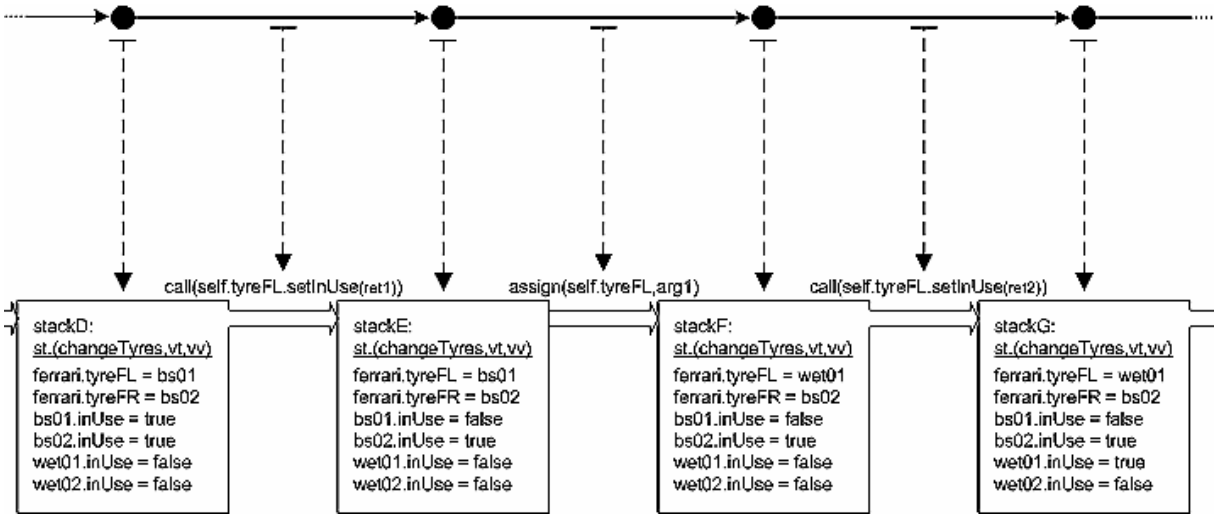
The above mentioned sequence of actions as term fort he behaviour of method changeTyre() will be realised by the control graph. Just as the data space the control graph of a Transformation System is a transition graph, consisting of edges and nodes. It is its purpose to represent (dynamic) activities of the system.
Actions, i.e. transformations[107], are (still) part of the data space, but their correlation is expressed by the control graph. Activities of the system, denoted by the edges of the control graph, are labelled with transformations of the data space; in-between states, denoted by the nodes of the control graph, are labelled with system states of the data space. Thus absurd sequences of actions are eliminated.
The following diagram shows how the control graph "chooses" the sensible transformations out of the "huddle" of system states and transformations. In contrast to the data space the control graph is almost determinate in its progression of transitions. Transformations of the data space that are not labels of transitions of the control graph become nonrelevant for the representation of the system, particularly for its behaviour. It has to be a constraint of the control graph that the same method, i.e. the execution of the same method, is always denoted by the same sequence of actions (by the control graph).



The previous diagram shows a rough representation of the changeTyres() method and its respective sequence of actions: the actions (transformations) of changeTyres() become the labels of an transition-sequence of the control graph.

[107] More precisely a transformation between two concrete system states A and B is given by an action leading from system state A to B.

The following diagram shows a short part of the sequence with system states described in more detail. System states are represented by their call stack and some attributes that are concerned in this section of the method's execution. The setting of attribute inUse (transition from system state D to E) is again shortened to 1 action even 3 actions are needed to manipulate the attribute (Call-action of setInUse method, Assignment-action and Return-action) to be exact.



Hereafter transformations of the data space that are not labels of any transition of the control graph disregarded as they do not server any purpose w.r.t. the system on the whole.

5.2. Object-oriented characteristics

After defining and instantiating Object-Oriented Transformation Systems it is now time to review the goals of this framework.
Object-Oriented TS claimed on the one hand to be capable of representing object-oriented systems and their features, and on the other to have an object-oriented structure themselves and thus embody these properties as well.
In order to verify this claim Object-Oriented Transformation Systems will be checked against the concepts and characteristics of object-orientation in section 2.3. (see appendix A)

5.2.1. The object as smallest unit

"object as atomic component"

A system state A is described by carrier sets A_s, static data functions fun_A, class states $static_{A,Cl}$, and a call stack $stack_A$[108]. Since data functions and call stacks only hold information about the system's dynamics, it is the carrier sets and the class states that describe the system's structure.
Both the elements of a carrier set A_{Cl} and the class state $static_{A,Cl}$ are objects of a class $Cl \in$ Classes. The class states are so-called "static instances" of their respective classes, separated from the other objects as they hold information and data values that apply to the class itself.
The remaining carrier sets A_s of data types $s \in DS$ shall be regarded as given a priori. These sets remain static throughout the system[109] and thus are not able to express any modification of the system. Their elements are solely used to express data values and do not carry any semantical significance. Eventually they do not have identities and thus can not represent entities of a system.
Since objects are the only (and atomic) components to constitute the system's structure the concept "the object as atomic component" holds.

"classes & type-system"

The type-system is expressed by the data space signature DSig, more precisely by its Classes. Every object has a well-defined type since the carrier sets of different classes are disjoint (constraint OTCidentities).

"data encapsulation"

All attributes and methods are defined as part of classes, within state signatures and behaviour signature, respectively. No activities happen and no data values are expressed outside of objects, as all information (data), describing a system's state, is encapsulated in objects.

"information hiding", "access levels"

Access levels are clearly implemented in Object-Oriented Transformation Systems as the set AL can be defined arbitrarily. The system's actions then restrict the access to methods and attributes as defined by the corresponding access classes.
The concept "information hiding" is slightly violated: since all of major object-oriented systems of the present make the access to attributes dependent on their access level, Object-Oriented Transformation Systems do as well. Thus attributes may be modified from the outside under certain conditions.

---

[108] $s \in DS \cup$ Classes, $fun \in DFun$, $C \in$ Classes.

[109] Actually primitive data sort DS and their carrier sets are identical for all instances of a certain kind of object-oriented system, e.g. like Java or the UML.

Though the latter may violate the original object-oriented idea of data-encapsulation and –abstraction, the concept is not contradicted, but rather extended, as the original idea can still be expressed by Object-Oriented Transformation Systems, setting the access levels of all attributes to "private".

5.2.2. Generalisation

"generalisation"

The set $Super_{Cl}$ of every class signature $Cl \in Classes$ defines the superclasses of Cl, where a superclass is a generalisation of Cl. Thus the type-system can be structured, establishing a hierarchy among classes. Object-Oriented Transformation Systems allow "multiple inheritance", i.e. a class having more than one superclass. However the generalisation is restricted by the constraint OTCselfinh, i.e. a class must not be the parent to one of its ancestors (preventing self-inheritance).

"inheritance"

Inheritance is secured by the constraint OTCinher: a class inherits all attribute- and method-symbols from its superclass(es). Nevertheless attributes and methods can be refined or overwritten by subclasses[110], modifying or even the originally intended semantics (see paragraph 5.2.4 for succeeding concepts polymorphy and delegation).

"multiple inheritance"

Multiple inheritance is also supported by Object-Oriented Transformation Systems, since a class can have several parents/superclasses. However self-inheritance is prevented by the constraints OTCselfinh and OTCdag, i.e. a class being ancestor of itself.

As a result of multiple inheritance, the concept of implementing interface-classes, known from the programming language Java, can be embodied by Object-Oriented Transformation Systems. Interface classes are represented by classes, whose object-sets (carrier-sets) are empty (see remark 4), since interface-classes must not be instantiated.

A class Cl_1 "implements" an interface-class ICl_2, simply by being a subclass to ICl_2: $ICl_2 \in Super_{Cl}$. Thus Cl_1 inherits the (abstract) method symbols of ICl_2 that have to be implemented (see chapter 9 of [Java00]).

Possible ambiguities, resulting from repeated inheritance for instance, have to be addressed, when translating a concrete object-oriented system into Object-Oriented Transformation Systems. Algorithms for the general case and/or considerations of the individual cases resolve the task.

5.2.3. Aggregation

"aggregation"

Aggregation is realised by attribute symbols, since the type of an attribute symbol can be another class: an attribute's type corresponds with the class that is aggregated (see definition 2). The respective (partial) attribute values of objects refer to the aggregated objects (see definition 4).

"aliasing"

However objects do not actually become parts of other objects, but attributes act as pointers to objects. Thus the attributes of an object act as substitutes, i.e. aliases, for other objects.

[110] In doing so the originally intended semantics of methods may be modified or get lost.

"multiplicities"
The set Types introduces arrays of existing types, including data sorts and classes. Thus it is possible for attributes to be an alias for an array of objects, i.e. pointers to multiple other objects, realising the concept "multiplicities".

5.2.4. Delegation & polymorphy

"polymorphy"
Polymorphy allows objects of subclasses to replace instances of the superclass. However in doing so the object should remain an object of the subclass with all the properties of a subclass-object.
Polymorphy is realised by the instance sets of classes: the value of an attribute with type Cl can be any element of the Cl-instance set AI_{Cl}. In the same way the input parameters of invoked methods can be given as instances of the desired types.
Assuming a class Cl1, being a generalisation of another class Cl2, and a Cl1-method meth, being invoked on a Cl2-object obj that acts as (replaces) a Cl1-object. Because of OTCinher, the method-symbol meth has been inherited by Cl2 and thus meth is a part of obj. Hence meth can be invoked on obj, executing the meth of class Cl2 that may differ from meth of class Cl1, being refined or overwritten in Cl2.

"delegation", "dispatching"
As stated before dispatching is not an advantage of object-orientation, but a task that has to be resolved in order to realise polymorphy.
However Object-Oriented Transformation Systems constitute a kind of delegation: delegation of responsibility. Objects or rather methods pass on the responsibility of their thread between each other, being the active object / method of their respective thread[111]. This concept is realised by the call stacks of system states tracking the active method, its environment and the associated active object.

5.2.5. Reutilisation

"genericity"
Genericity's purpose mainly is code-reuse: abstracting a method's definitions from the type of used variables / parameters; defining new structures (classes) with a placeholder-type. Both definitions feature an abstract, so-called "generic" type that has to substituted by a concrete class, when applied, e.g. let us assume there is the a generic type "list[α]", where α is a placeholder-type, later replaced by a concrete classes, for example "String", yielding a new types "list[String]".
These new (concrete) classes can be translated into Object-Oriented Transformation Systems like ordinary classes.

[111] Non concurrent object-oriented Transformation Systems contain only one thread that is running from the initialisation to the finalisation state of the system.

"overloading"

Overloading denotes the multiple use of a method symbol, i.e. a method-name, for methods whose purposes diverge. Since names are not significant in mathematical structures like Object-Oriented Transformation Systems and can be exchanged without changing the system's structure or its behaviour. Nevertheless overloading can clearly be expressed by Object-Oriented Transformation Systems, since a method symbol can be an element of multiple behaviour sets, i.e. it is possible that meth $\in$ $BehSig_{C,al1,arg1,type}$ and meth $\in$ $BehSig_{C,al2,arg2,type}$ for a C $\in$ Classes, al1,al2 $\in$ AL, arg1,arg2,type $\in$ Types.

OTCtyping secures that methods have to differ in name or argument-type.

Multi methods do not impose a problem for Object-Oriented Transformation Systems, since methods are distinguished on the basis of all and thus more than one parameter anyways.

5.2.6. Extensions

"class attributes and methods"

Class attributes and class methods are realised by the class state signatures $ClStSig_C$ and the class behaviour signatures $ClBehSig_C$ of every class C $\in$ Classes. Furthermore several actions deal with invocation and returning of class methods as well as the manipulation of class attributes; these are definition 5(b), definition 6(b), and definition 7(b).

"concurrency"

Concurrency has been realised as an extension of Object-Oriented Transformation Systems, constituting "concurrent Object-Oriented Transformation Systems" (see section 4.6.). In order to express simultaneous threads the call stack of system states is replaces by a set of equitable call stacks. Synchronisation is realised by new actions that allow methods to lock and unlock objects (see definition 9).

"exceptions"

Exceptions have not been realised and thus have to be part of future work.

For now, an exception has to be constituted by a prematurely called Return-action of the method that the error occurred in.

Chapter 6

Conclusion

The aim of this paper has been to define a general semantic framework for the integration of object-oriented systems as an instantiation of generic Transformation Systems by M. Große-Rhode. Object-Oriented Transformation Systems should constitute a formal mathematical semantic that, on the one hand, subserves a formal modelling of object-oriented systems, and on the other, permits an appropriate practical application on the other hand.
The intention was to have an object-oriented structure for Object-Oriented Transformation Systems, resembling the structure of the present's major object-oriented systems, in order to facilitate the translation process, i.e. the integration. Therefore the second chapter distinguished the terms "object-orientation" and thus "object-oriented system" by identifying the characteristics of today's (major) object-oriented systems. The latter determined the definition of Object-Oriented Transformation Systems.

In order to prove that the definition of Object-Oriented Transformation Systems constitutes an instantiation of generic Transformation Systems by M. Große-Rhode, in section 4.6 it was shown that the structure of data space signatures and their respective models, i.e. sets of system states, form an institution in the sense of [GB84] and [GB92]. Hence existing theorems like the composition of Transformation Systems and development relations can be applied to Object-Oriented Transformation Systems.

Furthermore, it was also shown that Object-Oriented Transformation Systems can be extended to describe concepts like concurrency without altering their fundamental structure.
Finally the fifth chapter showed that the identified characteristics can be expressed by Object-Oriented Transformation Systems, thus it should be possible to express object-oriented systems and its properties.

Proof of the latter is yet to be shown and it remains to be examined how good Object-Oriented Transformation Systems work in a practical application, for example in a sweeping case study. It will be the next step of this approach to integrate concrete object-oriented systems into Object-Oriented Transformation Systems.
Nevertheless, whether it is Java, the UML or another object-oriented system, translating the general properties has to be done only once, including the system's access modifiers, primitive data-types and data-functions. The translation can then be used by any instance of the system (like a Java-program or a UML-specification).[112]
In addition object-oriented systems often provide several predefined classes within so-called packages.

[112] These definitions are determined by the object-oriented system, e.g. the possible access levels are the same for all Java-systems.

Example

One reason for the predominant position of the programming language Java is the extensive supply of packages, including predefined classes that are used by virtually any Java-project / Java-software system. While the package "java.lang"[113] *is automatically available in any Java-program, others are usually included as well, e.g. java.util*[114]*, java.io*[115]*.*

Meanwhile classes are rather irrelevant for specification languages like the UML. However, since UML-specifications will be implemented by programming languages, lots of structures like packages and classes are simply presumed by the specifications. Eventually even the UML predefines some classes like "Enumeration", "DateTime" that can simply be used within specifications on the one hand, and on the other have somehow to be realised within the used programming language and thus be represented in Object-Oriented Transformation Systems.

To complete the translation into an Object-Oriented Transformation System the remaining classes[116] to be interpreted individually / manually, unless there is a translation algorithm. These "integration mechanisms" have to be found / defined once for every object-oriented system.

As a result of section 4.5., verifying Object-Oriented Transformation Systems to be an instantiation of generic Transformation Systems, the composition of Object-Oriented Transformation Systems can be achieved based on theorems that have been proven by M. Große-Rhode in [Gro01]. However, it might be desirable to specialise these relations in Object-Oriented Transformation Systems at some time.

One result of this paper is that the consistency check between object-oriented systems can be stated more precisely. In order to automate the integration process translation mechanisms have to be found for each individual object-oriented system.

For the project IOSIP the interest is directed towards heterogeneous specification techniques, mainly the UML, being the prevalent one of today. Therefore it is intended to define a formal description of the semiformal syntax of the UML by graph-grammars[117]. In order to integrate the latter into Object-Oriented Transformation Systems an integration-algorithms has to be found in a second step.

[113] Contains classes like String, Object, Math, Thread, System.
[114] Contains classes like Calendar, Collection, Date, Hashtable, Properties, Vector.
[115] Contains classes for input and output operations with the file-system or other data sources such as databases.
[116] All the classes that have not been covered by the packages, mentioned before.
[117] Future work of the project IOSIP.

Bibliography

[BDMN73] Graham M. Birtwistle, Ole-Johan Dahl, Bjoern Myhrhaug, and Kristen Nygaard: *Simula Begin*, Lund, Sweden, 1973. ISBN 91-44-06211-7; Petrocelli/Charter, New York, 1975. ISBN 0-88405-340-7.S

[BKS04] Benjamin Braatz, Markus Klein, Gunnar Schröter: *Semantical Integration of Object-Oriented Viewpoint Specification Techniques*, Technische Universität Berlin, 2004.

[GB84] Goguen, J. A. and Burstell, R. M. *Introducing Institutions*. Springer LNCS 164. 1992.

[GB92] Goguen, J. A. and Burstell, R. M. *Institutions: Abstract Model Theory for Specification and Programming*. Article. Journals of the ACM. 1992.

[Gro98a] Martin Große-Rhode: *Algebra Transformation Systems and their Composition.* In E. Astesiano (ed.), *Fundamental Approaches to Software Engineering (FASE '98)*, pages 107-122, Springer LNCS 1382, 1998.

[Gro98b] Martin Große-Rhode: *First Steps Towards an Institution of Algebra Replacement Systems. Applied Categorial Structures,* 6(4):403-426, 1998.

[Gro01] Martin Große-Rhode: *Semantic Integration of Heterogeneous Formal Specifications via Transformation Systems*, Technische Universiät Berlin, 2001

[Jac01] Ivar Jacobsen: *Object-Oriented Software Engineering: A Use Case-Driven Approach*, Addison-Wesley, ISBN 0201544350

[Java00] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification, Second Edition*, Sun Microsystems, Inc. 2000, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[Kay80] Alan Kay, Smalltalk-80: Blue Book: The Language and Its Implementation ISBN 0-201-11371-6

[Mey01] Bertrand Meyer: *Object-Oriented Software Construction*, Prentice Hall, ISBN 0136291554

[RBP01] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen: *Object-Oriented Modeling and Design*, Prentice Hall, ISBN 0136298419 [Par01] Daniel Parnitzke: *On Formal Semantics of Object Systems with Data and Object Attributes,* Technische Universität Berlin, 2001.

[Ten01] Jennifer Tenzer: *A Formal Semantics of UML Class Diagrams based on Transformation Systems,* Technische Universität Berlin, 2001.

[UML03] OMG Management Group: *OMG Unified Modeling Languages Specification,* Version 1.5, March 2003.

[WKWC94] U. Wolter, M. Klar, R. Wessäly, F. Cornelius: *Four Institutions – A Unified Presentation of Logical Systems for Specifications.* Technical Report. TU Berlin, 1994.

Appendix

A – Characteristics of object-orientation

The following list resumes the characteristics of today's object-oriented systems that have been identified in section 2.3.. These characteristics serve as definition of object-orientation for the scope of this paper.

The following characteristics are referred to within this paper by CX, where X ∈ [1;6]:

C1 – The object as smallest unit:	• object as atomic component • classes & type-system • data encapsulation • information hiding • access levels
C2 – Generalisation:	• generalisation • inheritance • multiple inheritance
C3 – Aggregation:	• aggregation • aliasing • multiplicities
C4 – Delegation & polymorphy:	• polymorphy • delegation • dispatching
C5 – Reutilisation:	• genericity • overloading • multi methods
C6 – Extensions:	• class attributes and methods • concurrency • exceptions

B – Constraints of Object-Oriented Transformation Systems

This appendix lists all constraints that have to be satisfied by every Object Oriented Transformation System.

The following constraints are referred within this paper by OTCxx, where xx stands for a short form of the constraint's intention.
The columns "Definition" and "Page" denote, where a constraint has been introduced.

Constraint	Definition	Page	Description
OTCsorts	1	35	Obligatory data sorts bool and nat bool, nat $\in$ DS
OTCterms	1	35	Obligatory data functions true, false, zero, succ $\in$ DFun
OTCobject	1	35	Root-class Object Object $\in$ Classes
OTCdag	2	38	Hierarchical DAG-structure of type-system $Super_C \neq \emptyset \quad \forall C \in Classes \setminus \{Object\}$
OTCselfinh	2	38	Preventing self-inheritance $C \notin Ancestors_C$
OTCinher	2	38	Inheritance is secured by constraints OTCinher1, OTCinher2, OTCinher3, OTCinher4; short OTCinher.
OTCtyping	2	38	Typing is secured by constraints OTCtyping1, OTCtyping2, OTCtyping3, OTCtyping4; short OTCtyping.
OTCconstructor	2	39	Obligatory class method of all classes. constructor $\in ClBehSig_{C,C}$
OTCself	3	44	Always existing local variable self self $\in$ X
OTCabool	4	46	Interpretation of obligatory sorts bool and nat $A_{bool} = \{T, F\} \wedge A_{nat} = \mathbb{N}$
OTCexists	4	46	Obligatory attribute exists obj.exists $\in A_{bool} \quad \forall obj \in A_C, \forall C \in Classes$
OTCidentities	4	47	Realising object identities, i.e. individual objects $A_{s1} \cap A_{s2} = \emptyset \quad \forall s1,s2 \in Classes$
OTCeval	4	48	Evaluation of obligatory terms true, false, zero, succ(t) $eval_{bool}(true) = T, \quad eval_{bool}(false) = F, \quad eval_{nat}(zero) = 0, \ldots$
OTCconstds	Remark 4	50	Constant carrier sets of primitive data typese $B_s = A_s \quad \forall s \in DS, \forall A, B \in States$
OTCconstdfun	Remark 4	50	Constant data functions $f_B = f_A \quad \forall f \in DFun, \forall A, B \in States$
OTClocked	4b	75	Obligatory attribute locked obj.locked $\in A_{bool} \quad \forall obj \in A_C, \forall C \in Classes$

C – Paper conventions

The following table lists all conventions of this paper. Paper conventions are requirements that do not have to be satisfied by an Object-Oriented Transformation Systems, but their satisfaction is assumed within this paper.

Paper conventions are referred to within this paper by PCONxx, where xx stands for a short form of the convention's intention.

Convention	Definition	Page	Description
PCONdisjoint	Remark 1	37	$x \cap y = \varnothing \quad \forall x,y \in \{DS, DFun, AL, Classes\}$
PCONlevels	Remark 2	41	Existence of access levels "private" and "public" $private, public \in AL$
PCONprivate	Remark 2	41	$AC_{C,private} = \{C\} \quad \forall C \in Classes$
PCONpublic	Remark 2	41	$AC_{C,public} = Classes \quad \forall C \in Classes$
PCONalself	Remark 2	41	$C \in AC_{C,al} \quad \forall al \in (AL \cup \{Class\})$
PCONatt	Remark 2	42	No equally named instance-attributes ($att \in StSig_C$) and class-attributes ($catt \in ClStSig_C$)
PCONmeth	Remark 2	42	Instance- and class-methods do not match in name and argument-list.
PCONvoid	Remark 2	43	Alternative notation of the empty type λ $void := \lambda \in Types^*$