User-Level Integration of Data and Operation Resources by means of a Self-Descriptive Data Model

two parts,
Part I

Lay out: T.D. Meijler and M. van der Tweel

Printed by: Krips Repro Meppel

NWO - SION financed this study, and gave support in printing this thesis

Cover: A Fish Made of Fish, design by: Prof. dr. H.A. Lauwerier and T.D. Meijler

USER-LEVEL INTEGRATION OF DATA AND OPERATION RESOURCES BY MEANS OF A SELF-DESCRIPTIVE DATA MODEL

(INTEGRATIE OP HET GEBRUIKERS-NIVEAU
VAN DATA- EN OPERATIE HULPBRONNEN
DOOR MIDDEL VAN EEN ZELF-BESCHRIJVEND DATA MODEL)

PROEFSCHRIFT
TER VERKRIJGING VAN DE GRAAD VAN DOCTOR
AAN DE ERASMUS UNIVERSITEIT ROTTERDAM
OP GEZAG VAN DE RECTOR MAGNIFICUS
PROF. DR. P.W.C. AKKERMANS M.LIT.
EN VOLGENS BESLUIT VAN HET COLLEGE VAN DEKANEN.
DE OPENBARE VERDEDIGING ZAL PLAATSVINDEN OP
WOENSDAG 22 SEPTEMBER 1993 OM 16.00 UUR

DOOR
THEO DIRK MEIJLER
GEBOREN TE AMSTERDAM

PROMOTIECOMMISSIE

PROMOTOR:	PROF. DR. E.S. GELSEMA
PROMOTOR:	PROF. DRS. C. BRON
OVERIGE LEDEN:	PROF. DR. E.J. NEUHOLD
	PROF. DR. J. VAN DEN BOS

Voorwoord

Tijdens een enthousiasmerende practische periode aan het eind van mijn studie in Delft bij prof. dr. A.F. Mehlkopf ontstonden de eerste ideeën voor dit project. Nadien is het grootste werk geweest die wilde en nogal vage ideeën te concretiseren, en ze te relateren aan bestaande informatica-theorieën. Het blijkt niet alléén maar een nadeel te zijn om als een onbeschreven blad een gebied in te stappen: Men is eenvoudiger in staat om de problemen met frisse blik te bekijken, en tot onconventionele oplossingen te komen. Het benaderen van een probleem met een blanco geest is zelfs aan te bevelen, en is geheel in stijl met de werkwijze van professor Mehlkopf, maar bijvoorbeeld ook met die van de beroemde fysicus Niels Bohr.

De moeizame weg tot concretisering en theoretische inkadering moet prof. dr. E.S. Gelsema, die dit proces heeft begeleid, zwaar op de proef hebben gesteld. Ik ben professor Gelsema dan ook zeer dankbaar voor het geduld dat hij aan de dag heeft gelegd. Verder dank ik hem voor het initiëren van dit project (dat overigens een andere, meer informatica-technische richting is opgegaan dan hij zich had voorgesteld) en de gigantische hoeveelheid werk die hij heeft besteed aan het doorwerken van mijn teksten. Ook prof. dr. J.H. van Bemmel ben ik dankbaar voor het tolereren van een theoretisch informatica-onderzoek in een praktische medische informatica-omgeving. Prof. drs. C. Bron wil ik graag bedanken voor zijn essentiele commentaar en de positieve wijze waarop hij dat heeft gedaan. Dr. M.L. Kersten heeft de moeizame begin periode meegemaakt, en heeft daarbij veel werk besteed aan het de goede kant op sturen van dit onderzoek. Ik ben hem daarvoor dankbaar. Henk Kuil heeft belangrijke onderdelen van het model (verder) uitgewerkt en geïmplementeerd. Zijn mee-denken is het werk sterk ten goede gekomen. Ik wil hem hiervoor bedanken en voor de kameraadschappelijke samenwerking.

Professor Neuhold and Peter Muth have given me, thanks to their (positive) appraisal of my work, the courage to go on, at one of my darkest hours. Wolfgang Klas gave his full dedication, and has shown his great insight and thoroughness in discussing the formalization during three wonderful intensive days in Berkeley. Allen Scher, I want to thank you for correcting my English.

Als we praten over beproeving moeten mijn ouders, zusters en rest van de familie genoemd worden. Mijn ups, en vooral ook downs werden vaak aan hen versterkt doorgegeven. Zonder hun ondersteuning had ik de eindstreep niet gehaald. Annejet (mijn oudste zuster) vooral ook jij dient gemeld te worden als een rots in de branding waaraan ik me altijd vast mocht houden.

Verder zijn er nog de velen die direct of indirect hebben bijgedragen. Prof. dr. L.H. van der Tweel (Oom Henk), die me letterlijk uit de brand heeft geholpen. Marjolein van der Tweel

bedankt voor de totaal als vanzelfsprekende hulp bij het "lay-outen". Professor Lauwerier, uw hulp bij het maken van de voorpagina was onmisbaar, en mij zeer aangenaam. Prof. dr. A. van der Sluis en prof. Dr. S.D. Swierstra hebben mij met Professor Bron in contact gebracht. Ik dank hen daarvoor en voor de getoonde interesse. Dr. A.M. (Astrid) van Ginneken was een onmisbare praatpaal. Freek van den Heuvel bedankt voor het vormgeven van de uitnodigingskaartjes en de rug van het proefschrift. Mees, Albert, Guus, dank jullie voor het lezen en becommentariëren van mijn stukken.

Freek, René (Idema), Trudy en vele andere (ex-)vakgroepgenoten, bedankt voor jullie vriendschap. Ten slotte wil ik ook alle andere vrienden bedanken, met name René (Flippo) en Annette, voor hun warmte en ondersteuning. Ook zonder jullie was dit niet mogelijk geweest.

Rotterdam, 24 juni 1993

Gelijk hebben is niks, gelijk krijgen is de kunst
Dr. Ir. Th. P. (Theo) Tromp

Wat je goed doet wordt gedeeld, wat je fout doet vermenigvuldigd
Prof. Dr. D. (Dirk) Durrer

Rigorous argument is usually the last step! Before that, one has to make many guesses, and for these, aesthetic convictions are enormously important.
Roger Penrose, in: The Emperors New Mind, p 421

Aan mijn ouders, zusters, en overige familie
Ter nagedachtenis aan de grootouders Meijler en Schendstok

Contents

Chapter 1

Introduction

This thesis describes the design of a system to integrate data access and various forms of data analysis and editing tools, -sets of operations-, under a single interactive user interface. The projected system is called YANUS (Yet ANother Unifying System). One of the major goals of this study is to create an environment for the user where he/she[1] has optimal freedom to combine operations and apply these directly to his data, and where he is being supported to do so correctly, i.e. in conformance with the meaning of the data and of the operations. Direct application means that the user does not need to copy or transform the data. Another major goal is the encapsulation of existing external systems. Certain operations may, without the user's knowledge, be executed in these existing systems. Also, data to which operations are applied may be stored in existing databases. Thus, the user should be able to analyze data from existing databases with existing tools with minimal effort. He does not need to know about the different interfaces of different systems and about possible data translations. Finally, the environment must be extensible, with respect to the data and operations which may be accessed by the user, and with respect to the databases and software packages to be used to provide data and operations in the integrated system. The main hypothesis supported in this thesis is, that a powerful self-descriptive object-oriented data model can play a central role in achieving these goals.

1.1 Terminology

Information processing on a computer system is based on a strong relationship between data stored in the computer and some *reference world*. The data have some meaning in the reference world and conversely, the reference world serves as a source for the acquisition of data. As a result, data may tell something about the reference world. By using appropriate data manipulation algorithms, it is possible to derive information about the reference world which is otherwise difficult to obtain directly. A requirement is, that data manipulation leaves the relationship between the data and the reference world intact (Klas 1990). We will refer to this as the information processing system representing (or reflecting) the reference world *faithfully*.

[1] We shall use "he" in general.

Note that the reference world may be any part of the real world, including information processing itself. This problem of representing the reference world faithfully in a computer system is the main subject of investigations on data modelling (Hull and King 1987; Zdonik and Maier 1990)

A user of an information processing system employs this relationship between the system and the reference world. In the following it will be assumed that an information processing system is used through the invocation of *actions*. This model of the user invoking actions on the computer corresponds to the perspective of the user being in charge; one could also think of a perspective in which the computer is in charge, which corresponds to a user providing input when the computer prompts for it. This latter view is not incorrect, but will not be adopted in this thesis.

Using an information processing system, the user can perform actions for entering and editing data, or otherwise perform actions to invoke so-called operations. In general an *operation* is applied to data. It may select or retrieve existing data, derive new data, or update the data to which it was applied in an indirect way.

The *user interface* to an information processing system, i.e. the way in which actions can be invoked by the user may take several forms:

1. The invocation of a fixed sequence of actions can be laid down in a (kind of) program that can be invoked as a whole. An example of such a program is a script to drive a statistical software package such as BMDP. The script describes the successive steps in the statistical analysis of a data set. This will be called a *batch interface*.
2. An *interactive interface* allows for more flexibility: actions are invoked, and dependent on the result (which may be viewed), the user decides which subsequent actions will be invoked. A *command interface*, where the user invokes actions by typing their names, is an example of such an interactive interface. More user friendly, and also very common, is a *graphical interactive* user interface: The user is given a choice of applicable actions graphically through menus, and the result of his actions is automatically shown. One step further in user friendliness is the *direct manipulation* interface (also a graphically interactive interface) (Shneiderman 1983): the user is given the feeling that he is directly manipulating the data. For example, the user may select a data element directly from the screen and apply an operation to it. The results are immediately visible and may be used for further manipulation.

In general, the use of a computer system is subdivided into the use of several separate information processing systems, so-called *software packages* or *applications*. A software

package may be characterized by the types of data it can handle and the actions/operations it provides, i.e., which reference worlds may be represented and how they are represented. The operations correspond to the questions which may be asked about the reference world and/or to the reference world events which may be simulated. Certain software packages allow the data types and/or operations they provide to be extended; these will be referred to as *environments*.

Certain types of data, such as numbers, tables, texts, etc. can be used in different contexts, that is, to describe various reference worlds. Software packages that provide storage or processing of these kinds of data will be referred to as being *general*.

A software package provides access to certain *resources*. A Database Management Systems (DBMS) specifically provides access to *data resources*, since such a system allows data to be stored in a structured way and provides powerful retrieval operations. A software package which is primarily used for applying operations to data, e.g. a statistical software package or a package for image processing, will be said to provide access to *operation resources*.

1.2 Problem description

A user would like to combine the resources provided by different software packages, and would thus like to regard the complete computer system as one information processing system. For instance in the Department of Medical Informatics, where this thesis has been developed, software packages are being used for the following functions:

- Database Management - Ingres
- ECG analysis - Means, developed at the department itself
- Image processing - e.g. SCIL
- Statistical analysis - the well known packages such as SPSS and BMDP
- Statistical pattern recognition - Ispahan, a package for interactive data exploration, developed at the department.

It should be possible to apply statistical analysis and/or pattern recognition to data stored in DBMS's. Also, applying statistical analysis or pattern recognition to numerical results obtained by image or signal processing is meaningful. Allowing such kind of combinations will increase the power and flexibility of information processing. Note that combination of software packages is especially interesting if, as in this example, the packages are quite general. Such combined use of software packages is problematic for the following reasons:

1. Stand-alone nature of software packages

Each software package is technically its own process, having its own data space. Thus, in order to apply an operation available in one package to data available in another, these data must be copied between programs. In general, for two arbitrary software packages such copying of data is not supported. It may even involve having to re-enter the data. But even when such data exchange **is** supported, it means that the faithfulness of the representation of the combined reference world breaks down. While in the real world there is only one entity, the user of the computer system has to create and use two data sets pertaining to the same entity, in order to do special processing with these data. This will be referred to as the difficulty of combining resources. Nilsson et. al. (Nilsson, Nordhagen et al. 1990) call a computer system that is subdivided into separate programs or applications *application oriented*. According to them, such a computer system not directed at the user's task.

2. Differences in user interface

In general, the interfaces of different software packages may be quite different. Not only may such differences be encountered as between a batch and a graphical interactive interface, but graphical interactive interfaces of two software packages may also be different in style (Nilsson, Nordhagen et al. 1990; Wiecha, Bennett et al. 1990). This demands that the user learns many different interfaces, which may lead to confusion and thus to errors.

3. Semantically heterogeneous resources

In general, the different resources to which different software packages give access are not tuned to one another at the semantic level, that is, according to the way in which they reflect the external world, since they were not designed to be used in combination. Specifically, different data resources may overlap, use different terms or measures for similar concepts etc. This aspect is called semantic heterogeneity, and is well described in the literature (Schrefl and Neuhold 1988; Hammer 1991).

In general terms, this thesis is concerned with the solution to problems encountered when using an application oriented environment as mentioned in 1 and 2 above. Although the problem of semantically heterogeneous resources, as mentioned under 3, will be encountered in the application of the system, this thesis is not directed to research in that area. In order to solve such problems as described above, the following **technical** problems are distinguished in this thesis:

A. Creating an environment without applications

The task is to create an information processing system in which a multitude of resources (operations and data) can be used interactively **by direct manipulation**, without the traditional subdivision into separate applications. The goal should be to allow resources to be used in

optimal combination. The concept of optimal combination is explained below. We claim that this can be achieved by creating one global environment in which all resources can be accessed, and in which the (combined) use of the resources is based on typing, i.e., classification of the resources on basis of a type as described below.

Typing is known from the area of programming languages, and from mathematical computation. The type of a data object or symbol retains what this data object or this symbol is supposed to represent in the reference world, e.g. a text or a matrix. By keeping track of types, incorrect use of the data objects/symbols can be prevented. Instances of incorrect use are e.g., applying arithmetic to a word, or computing the sine of a matrix (note however, that similarly named operations may be defined specifically to be correct and meaningful).

By comparing the use of data conform a typing scheme with the normal use of data within an application oriented environment, the idea of optimal combination of resources can be understood: In an application oriented environment only those operations are applicable (without having to copy data) that are offered by the application. When using a typing scheme, **all** those operations which are in conformance with the typing are applicable. Thus, the typing scheme allows a better, more faithful representation of the reference world.

The unifying system may thus be regarded as one big software package in which the optimal combination of resources on the basis of a typing scheme is directly reflected in the user interface. One major technical problem in building such a system lies in the area of user interface technology, given the fact that the number of resources, and thus of different data types and operations may be very large. In standard user interface technology, software which gives feedback about possible correct combinations for all data types and all operations grows exponentially with the number of types and operations. This is especially troublesome, given that such a system must be:

- adaptable to the specific resources needed;
- extensible, so that new resources may be easily added.

This means that it must be possible to introduce new data types as well as new operations; thus in both meanings this system is an environment.

An important aspect of such a system is user friendliness: The surveyability of the system should be equal to, or possibly better than that of a normal application oriented computer system. Resources must be offered to the user in an ordered way; the user must be guided in finding them and in using and combining them in a correct way. The use of resources must also be sufficiently specific, i.e., adjusted to their meaning in the reference world.

B. The encapsulation of existing software packages

The *encapsulation* of a existing software package allows the reuse of its resources within another software package; the interface of the encapsulated software package is hidden from the user of the other software package. An encapsulated software package is also referred to as an *underlying* software package. This thesis describes the encapsulation of software packages under an environment as sketched in A.

Note that the problem of encapsulation is basically one of mapping untyped isolated resources to typed integrated resources as described under A. To obtain a better insight in these encapsulation problems, it is convenient to subdivide the mapping into two parts:

a. A mapping which makes the resources from a software package available so that they can be treated as data types and associated operations. This involves a.o., driving the underlying software package.

b. A mapping which merges the resources, in terms of the data types mentioned under a, to one overall set to be offered to the user.
In this thesis the main focus is on merging data with operation resources, i.e. merging data types which correspond to similar data, but are coupled to different sets of operations (as corresponding to the applicability of different sets of operations to these same data in different software packages) into one data type. In the resulting data type the set of applicable operations is the union of the operations of the original data types. Enforcing this mapping involves, among other things, copying data automatically from one software package to another and maintaining consistency between these copies.

The encapsulation mechanism should allow adaptation to the resources needed by the user and should allow the extension of these resources.

Combining the aspects under A and B, the problems of using existing software packages in combination are resolved, while the resources they provide are retained. This is referred to as the user-level integration of data and operation resources -as possibly provided by existing databases, software and software packages- under one uniform direct manipulation user interface.

1.3 Scope

Within the general setting of the problem of the integration of existing software packages under one uniform direct manipulation user interface, this thesis basically focuses on the integration of general resources as available from DBMS's and general data analysis programs,

such as statistical software packages and statistical pattern recognition systems, to allow the application of such analysis techniques to these data resources. This limits the scope as follows:

- Due to the exploratory character of data analysis, the restriction is made that although the user should be guided in finding resources and combining/using them in a correct way, guiding him through a fixed sequence of different related tasks is not needed.
- Only systems that do not exhibit "autonomous" behaviour such as, e.g., measuring instruments or shared databases, will be encapsulated.
- The problems of merging resources may indeed be restricted to merging operation and data resources as mentioned in section 1.2 under Bb. Merging data resources, that is, creating a single semantically homogeneous data resource from several semantically heterogeneous and overlapping data resources as for example described by (Schrefl and Neuhold 1988) is not investigated. Note however, that the integration does include to allow for using such heterogeneous data resources in combination with each other.

Although an important aspect of this thesis is the implementation of the direct manipulation interface, the focus is on control and the incorporation of semantic feedback, rather than on the presentation. The assumption is that the presentation can be provided by so-called interaction toolkits.

In addition to a restriction in scope with respect to the resources which are to be presented in the environment, there is also a restriction with respect to the underlying programs to be encapsulated. For example, software packages that provide a purely graphical direct manipulation user interface cannot be encapsulated. These restrictions will be further specified in chapter 4.

1.4 Solving a specific problem: the direct application area

As described in the previous sections, this study has been directed at finding general solutions for general problems. However, this search for generality does not provide clear-cut requirements, i.e. criteria for incorporating certain features, nor does it provide a measure of the usefulness of the solution. Care must be taken that the general solution can at least handle a non-trivial specific problem.

The incentive for this study has been solving a specific integration problem. The specific problem is used in the sense described above, i.e. as a reference and test case for this research. We shall call this problem the *reference problem*, and call this area of application the *direct*

application area. Note however, that since this thesis is about a system **design**, the possibility of using this design to solve the reference problem can only be asserted in theory.

Integration of database access as provided by a DBMS (Ingres) and functionality for interactive statistical pattern recognition, as provided by Ispahan is the basic reference problem. Added to this is the application of simple arithmetic operations, as implemented by self-made software. The reason for this extension is, that two groups of functionality do not fully illustrate the need for optimal combination of operations on data. In the following characterization of the reference problem, the division is again made between how the resources should be modelled to the user and how the underlying packages should be integrated.

The user has access to tables with data derived from certain *individuals*[2]: each record describing one individual. The fields in the records correspond to different *features*[3] measured on each individual. The set of individuals is referred to as a population. The user should be able to browse through such a table, to create a sub-population by selection (i.e. through queries) and to apply pattern recognition functionality to such a population.

In pattern recognition, grouping of individuals into *classes*[4] on the basis of the feature values is of major importance. To this end each individual may be positioned in an n-dimensional feature space, where each dimension corresponds to one measured feature. Certain classes of individuals may have similar values for certain features; thus individuals in a certain class will be positioned in small hypervolumes in the feature space. When using sufficiently large populations, containing individuals from all different classes of individuals, such similarities may be found with pattern recognition techniques and may subsequently be used to classify new individuals.

In the pattern recognition system Ispahan, an important administrative data structure is the population tree. A population tree describes a hierarchical division of a population into sub-populations. The root node represents the complete population; descendant nodes represent subpopulations. These subpopulations have empty cross sections. A descendant node may again have descendants, etc. The user manipulates these trees: Rules for the classification of

---

[2] Individuals may for example be persons, animals, plants, certain named physical phenomena, storms etc.

[3] Measured features of a flower (the well known Iris problem) are for example the length and width of the petal and sepal leaves...

[4] This corresponds, for example, to several subspecies of a flower species such as Iris. For Irises, these classes are: Iris virginica, Iris versicolor, Iris setosa.

individuals into subpopulations as represented by a tree (calculated using pattern recognition techniques) are attached to the nodes in the form of *decision functions*; the tree is then called a *decision tree*. By applying a decision tree to another population, the individuals in the other population are classified, and this classification is again represented by a population tree. For further information see chapter 6, and (Gelsema 1980).

The use of arithmetic functions in this context allows the user to apply such functions to the numbers (feature values) presented in a population table. In the user interface the functions may be presented in a calculator. Optimally combining these operations on the data thus means that the user needs not copy (or retype) these data from the table to the input of the calculator.

Driving the underlying packages involves, a.o. the following:

- With respect to Ingres: sending queries to Ingres, and presenting the resulting numbers to the user in a table.
- In order to apply pattern recognition functionality, the table is automatically copied to Ispahan, which is initialized. The corresponding population trees may then be presented. At each manipulation of one of the trees, the system must send the corresponding command(s) to Ispahan, query Ispahan about the change that occurred in the tree and present this change to the user. Driving Ispahan also encompasses maintaining its state: Ispahan uses a menu tree to give access to its operations; at each moment Ispahan is in a certain state with respect to this menu tree. Graphics concerning the feature space as provided by Ispahan, can still be presented to the user.
- Driving the arithmetic software involves merely calling the appropriate functions and providing these functions with the appropriate parameters.

1.5 Related work

In principle, two points of view may be distinguished in relating this study to other work. One point of view concerns goals and achievements: "what does the approach yield?" the other concerns the ideas being used. Clearly, there may be related work that can be discussed in both categories; Projects that use similar ideas may well have (although not necessarily, (Sheth and Kalinichenko 1992)) similar goals.

In the following, we will mainly look at approaches that pursue similar goals.

Projects that offer one or both of the two following features are of interest:

1. "Blurring" the border between applications, or abolishing the application oriented architecture. Note that whether this architecture is adhered to or not (and if not, what kind of model is presented to the user) is independent of whether existing resources are integrated (see 2). For example a product such as Framework XE described by (Stevenson 1992) provides an alternative for the application oriented architecture, but it does not integrate existing software packages, i.e., all resources have been specifically implemented only for that product.
2. Integrating existing resources, and presenting them in a new form to the user.

Approaches that offer such features on an extensible basis are specifically of interest, i.e., those where the set of resources presented to the user may be extended and, in the case of integration of existing resources, the set of integrated packages may be extended.

This overview is not complete; completeness is not even attempted. It is more important to categorize certain kinds of approaches. By giving some of the (dis-) advantages of a certain approach, the ideas behind the YANUS approach may be clarified. We shall first shortly touch upon systems that provide feature 1; next on systems that mainly focus on feature 2; and finally on systems that provide a combination of the two.

Relatively simple examples of systems that abolish or blur the border between applications, are the so-called integrated packages (Stevenson 1992). Integrated packages offer an integrated combination of word processing, database management, a spreadsheet, and graphics and/or telecommunications. The different modules in an integrated package at least allow for easy data exchange. One such package (Framework XE) provides more than integration of these modules: it provides "unification": each application is always active and can always be used. Basically, this kind of overall accessibility of functionality, in an extensible form, is one of the aims in this study. A disadvantage of the Framework XE concerns its implementation: all data and software are constantly kept in working memory (also called "RAM" memory).

In the context of software engineering other work is being done to enable different applications to present different views of the same data to the user (or to several users) and allowing these different views to be edited concurrently such that all changes are broadcast to all views (Grudin 1991; Meyers 1991). Although this may also be called integration (and falls under feature 2) this is not a major effort of the present study.

Improved application oriented architectures are entering the market place in which the disadvantages of that architecture are diminished. Examples are NewWave, NeXTstep,

Windows NT, Apple system 7.0. These architectures are directed at yet other forms of integration of resources, namely "object embedding", using which the user may nest data from different applications, and "object linking" which allows the user to link data from different applications such that they depend on each other dynamically. These forms of integration are not specifically pursued in the present study. In these new architectures, data exchange between applications is also simplified but not rendered unnecessary. Of these products (at least?) NewWave provides some possibilities to encapsulation of existing applications, although only superficially. The amount of programming needed to encapsulate an existing package so that it can take part in communication protocols between applications equals or even exceeds the amount of work to create a whole new application (Crow 1989). A future development as described in (Lu 1992), is the so-called Document Oriented Interface. In such an interface, the data or documents rather than the applications are the central entities. Different modules, existing of objects presenting the data, and tools to process those data cooperate on the screen. No data exchange is necessary and it could thus be said that the idea of optimal combination of resources is also aimed at in that approach. However, according to Lu, a document oriented interface as is currently developed by companies such as IBM and Apple require a high level of cooperation between modules and corresponding standards. Integration of existing resources is not a major goal.

Other approaches of interest are the ones which provide both an application development environment and an environment in which existing software packages can be integrated (Nilsson, Nordhagen et al. 1990; Hornick, Morrison et al. 1991). The general idea is as follows:
The programmer of a new application who implements certain resources adds those resources to one large "pool" of resources. Resources already in the pool can be used by the programmer to implement new resources. A persistent[5] object space is part of this pool, so that applications have easy access to these data. This also allows for easy data exchange between the applications. Resources from existing software packages can also be added to this pool. The environments give support for building the necessary interface to (i.e., for encapsulating) these existing systems. As a result, all new applications can use the resources of all others, and of all encapsulated software packages and databases. This kind of integration where applications use each other's resources will be called "interoperability". Interoperability for the purpose of cooperation between different users and computers is a powerful form of integration and goes beyond the goal of this thesis. However, one user still has to deal with an application oriented environment, so that data must still be exchanged if certain functionality is not available in one application, but is in another. In comparison to these kinds

[5] Objects representing stored data.

of approaches, the present study could be described as aiming at creating a similar pool of resources for the (end-) user rather than for the programmer.

(Nilsson, Nordhagen et al. 1990) (also mentioned above) identify the problem of having several applications in an environment for one user; they introduce the term "application oriented". As an alternative they propose a so-called "task oriented" environment. However, tasks are again modelled as (non- overlapping) applications, and thus a separation is again made between them. Although in this specific well-defined application area (i.e. the use of tasks for Computer Aided Software Engineering, CASE) different non-overlapping tasks may well be identified, this will not be possible in general.

MW2000 (Van Mulligen, Timmers et al. 1990) is a project specifically directed towards user-level integration of tools and existing packages that work on information in (relational) data-bases. It provides a user model where the user applies operations to a certain data set, and as a result the data are transported to the corresponding software package for analysis. The present study may be viewed as a generalization with respect to the data types and corresponding software packages that may be incorporated in the integration, and it focuses more on interactivity and extensibility.

With respect to the other point of view, regarding the ideas that are being used, it suffices in this introduction to refer to the work of Sheth (Sheth and Kalinichenko 1992) and Klas et. al. (Schrefl and Neuhold 1988; Klas 1990; Klas, Neuhold et al. 1990). Although their work is directed at integrating existing resources (as will be mentioned in other chapters), the main importance of their work in the context of this thesis lies in the description and application of powerful, self-descriptive object-oriented data models. They show the flexibility and power of such approaches.

In other chapters where the principles behind YANUS are described, more attention will be given to the literature in which similar ideas are introduced.

1.6 Overview of the YANUS approach

In this section a short overview will be given of the main features of the (projected) system "YANUS" that is described in this thesis, and of the main principles behind it.

YANUS is a system for both the development and the operation of an environment as described in section 1.2. under A. In this sense, YANUS may be compared to a so-called 4th Generation Language (4GL) environment (Butler and Bloor 1989)

1.6.1 User interface based on type bookkeeping

The user interface of YANUS is based on type bookkeeping as mentioned in section 1.2. The user uses (data) objects. The type of an object determines the structure of the object, and thus indirectly, its allowed structure changes (in this sense, the use of YANUS may be compared to that of a syntax oriented editor (Teitelbaum and Reps 1981; Backlund, Hagsand et al. 1989)). The type also determines the operations applicable to the object. The use of an operation is determined by its signature, i.e.: the parameters it needs and their types. The use of objects and operations is modularized into so-called workspaces, in order to provide surveyability. This modularization does not harm the principle of optimal freedom to apply operations directly to objects on the basis of the object's type and the signature of the operation; the typing mechanism is independent of the modularization in workspaces. Thus, from the viewpoint of using the system through workspaces, operations may be applied globally across workspaces.

1.6.2 The development environment

As a development environment, the possibility of extending YANUS is based on the possibility of defining schemas, or specifications in general.

1.6.2.1 The specification of the user interface

A schema is used to define the types of objects, the operations (through their signatures as described above), and the modularization of the user interface into workspaces. A simple specification may also be used (but has not been studied) to define the presentation of these entities. The dynamics of the system, which guides the user while changing the structure of objects, applying operations, and opening and closing workspaces, is built-in in a dialogue manager which is schema independent, and does not need to be programmed.

1.6.2.2 Specifying the link to underlying packages

YANUS may be viewed as a shell on top of existing DBMS's and other software packages (fig. 1.1). Underlying packages may be driven through their user interface.

In the same way a schema is used to define the user interface (i.e. at the conceptual level), schemas are also used to define underlying resources. I.e., types of objects and signatures of operations are used to describe the resources an underlying package provides. This is the so-called implementation schema. In contrast to the conceptual data types and operations, these implementation data types and operations correspond to an implementation in the underlying software package. In order to allow the underlying package to be driven, other specifications

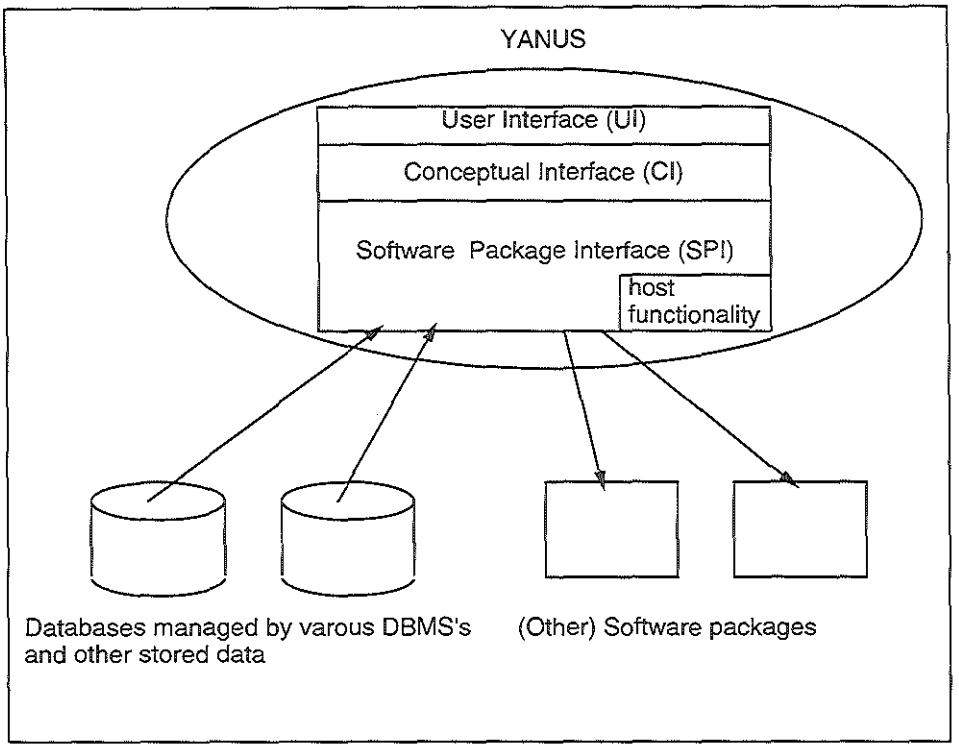

Fig. 1.1

Overview of YANUS. The user accesses the conceptual objects and operations in the conceptual interface (CI) through the user interface (UI). Conceptual objects and conceptual operations correspond to data and operations in the underlying resources. This correspondence is effectuated by the software package interface (SPI). The SPI interfaces to DBMS's (indicated by drums) and other software packages indicated by squares. The SPI also interfaces to host functionality. The arrows indicate the main flow of data: YANUS reads the data from the DBMS's and transports (and transforms) them to allow analysis in other software packages.

are used to describe a.o. how implementation operations must be mapped to input for the package, and output of the package must be mapped to objects in the YANUS representation. These specifications may also include the specification of a transition diagram describing the different states, and state transitions of the software package. The dynamics of driving the underlying software package is built-in, and adapts itself using the specifications. The dynamics of driving underlying software packages (i.e. executing operations requested by the user) involves a.o. selecting the right implementation operation and thus the appropriate software package to execute the operation, transporting and transforming the data on which the operation is applied in order to let it serve as input for the software package, generating commands for the package and parsing the output of the package.

1.6.3 The underlying principle: genericity

The extensibility of the YANUS system, based on the definition/specification of types and operation signatures, i.e., schemas or specifications in general, was described above. This concerns both the extension of the user interface and the extension of the set of encapsulated software packages.

Thus an essential feature is that it is not necessary to implement new software for each extension. To this end YANUS employs generic, schema independent software which implements the system's dynamics. The use of generic, schema independent software is based on the explicit run-time accessible representation of schemas. This is done by representing types, operation descriptions etc. as objects. The software adapts its behaviour by accessing and interpreting these objects. Note however, that in the following, when referring to objects, in general the objects as used by the user will be meant. Type objects and operation description objects will just be referred to as types and operation descriptions.

In conducting the user interface dialogue, this genericity is incorporated, e.g., in the following fashion (see fig. 1.2): The link between a type and a set of operation descriptions indicates which operations may be applied to a certain type of (user used) object. This link is employed by the generic software to guide the user in using the objects and operations conform the schema.

Similarly, in interfacing to underlying software packages, operation descriptions describing operations as offered to the user (i.e. at the "conceptual level") are coupled to operation descriptions at the implementation level describing operations as offered by specific software packages. This link is used to delegate the request for such a conceptual operation to the appropriate underlying software package(s).

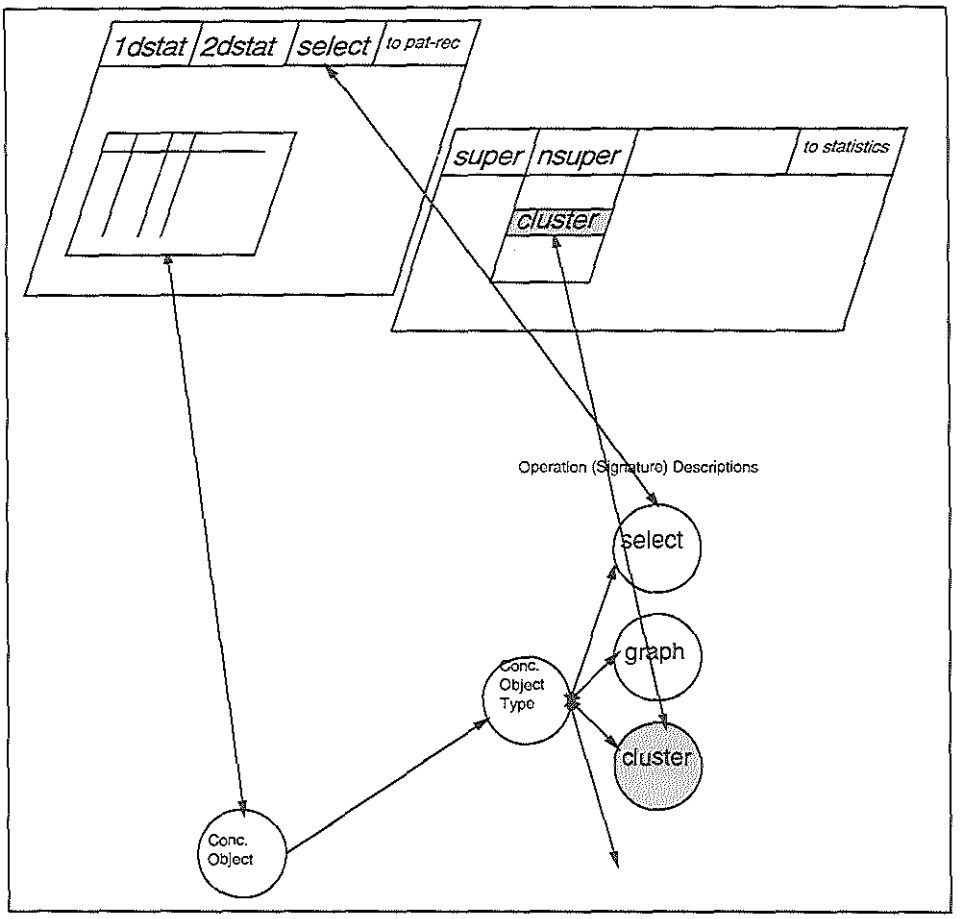

Fig. 1.2

Illustration of the use of schema information to control the dialogue. Objects presented on the screen are connected to the conceptual objects they represent. Objects presenting data (e.g. a table) are linked to their type. Using the explicit representation of the type, applicable operations (operation descriptions) can be found. These are represented as menu elements on the screen. Thus, when a data object is selected, using this knowledge structure, the appropriate menu elements may be made selectable. Reversely, if the user selects an operation through a menu element, objects can be found that may serve, given their type, as input for the operation. As shown, the mechanism can be used to control the user interface across workspaces.

As mentioned above, essential for employing genericity is the explicit representation of types as objects. It turns out however, that it should be possible to define different kinds of types, to which different generic behaviour is associated. For example, the dynamics of executing a request at the conceptual level and the implementation level are different. Moreover, this difference generally corresponds to a different structure for the type objects. To realize this, different types of types, so-called meta types must be introduced. Just as types describe objects, meta types describe types. Associated to a meta type, and thus to all types described by that meta type, is the generic software which employs the specific structure (as described by the meta type!) of those types. Meta types are thus the entities through which a descriptive piece of information, such as a type (but also an operation description) may be introduced, and through which the corresponding generic software may be implemented.

In a consistent line of reasoning, meta types must again have a type. All meta types have the same type. This metatype - type is its own type. Thus, the data model is self- descriptive.

1.7 Outline of the thesis

As may be clear from section 1.6, the data model forms the technological basis for the whole system. Chapter 2 presents this data model, i.e. what can be laid down in a schema, and what is the descriptive power of such a schema. The self-description of the data model is treated. Chapter 3 describes the control of the YANUS user interface using schema information. The workspace concept is formally introduced. The subject of chapter 4 is the execution of a request for operation execution, again by employing schema information. This encompasses invoking the appropriate package, creating (if necessary) the appropriate copy of the data, commanding the package to do what is required (possibly keeping the current "state" of the package into account) and incorporating the output of the package as a YANUS object, so that it can subsequently be used as input for further actions. In chapter 5 the application of the YANUS model to the reference problem is discussed. The implementation of the YANUS data model is discussed in chapter 6. Chapter 7 gives a discussion and conclusions.

Literature

Backlund, B., O. Hagsand, et al. (1989). Generation of Graphic Language-Based Environments. Swedish Institute of Computer Science.

Butler, M. and R. Bloor (1989). 4GL's: an Evaluation and Comparison.

Crow, W. M. (1989). "Encapsulating of Applications in the New Wave Environment." Hewlett-Packard Journal **40**(4): 57.

Gelsema, E. S. (1980). ISPAHAN: an Interactive System for Pattern Analysis: Structure and Capabilities. Pattern Recognition in Practice, Amsterdam, North-Holland Publishing Company.

Grudin, J. (1991). "CSCW." Communications of the ACM **34**(12): 31.

Hammer, J. (1991). The Identification and Resolution of Semantic Heterogeneity in Multidatabase Systems. First International Workshop on Interoperability in Multidatabase Systems, Kyoto, Japan, IEEE Computer Society Press.

Hornick, M. F., J. D. Morrison, et al. (1991). Integrating Heterogenous, Autonomous, Distributed Applications using the DOM Prototype. GTE Laboratories.

Hull, R. and R. King (1987). "Semantic Database Modeling: Survey, Applications, and Research Issues." ACM Computing Surveys **19**(3): 201.

Klas, W. (1990). A Metaclass System for Open Object-Oriented Data Models. University of Vienna.

Klas, W., E. J. Neuhold, et al. (1990). "Using an object-oriented approach to model multimedia data." Computer Communications **13**(4): 204.

Lu, C. (1992). "Objects for End Users." BYTE **17**(14): 143.

Meyers, S. (1991). "Difficulties in Integrating Multiview Development Systems." IEEE Software **8**(1): 49.

Nilsson, E. G., E. K. Nordhagen, et al. (1990). Aspects of System Integration. The First International Conference on Systems Integration (ICSI), Morristown, New Jersey, IEEE Computer Society Press.

Schrefl, M. and E. J. Neuhold (1988). Object Class Definition by Generalization Using Upward Inheritance. 4th International Conference on Data Engineering, Los Angelos, IEEE Computer Society Press.

Sheth, A. and L. Kalinichenko (1992). Information Modeling in Multidatabase Systems: Beyond Data Modeling. First International Conference on Information and Knowledge Management, Baltimore,

Shneiderman, B. (1983). "Direct Manipulation: A Step Beyond Programming Languages." IEEE Software : 57.

Stevenson, T. (1992). "Low-Cost Integrated Software: The New Synergists." PC Magazine **11**(15): 311.

Teitelbaum, T. and T. Reps (1981). "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." Communications of the ACM **24**(9): 563.

Van Mulligen, E. M., T. Timmers, et al. (1990). Implementation of a Medical workstation for Research Support in Cardiology. Symposium on Computer Applications in Medical Care (SCAMC), Washington D.C., IEEE Computer Society Press.

Wiecha, C., W. Bennett, et al. (1990). "ITS: A Tool for Rapidly Developing Interactive Applications." ACM Transactions on Information Systems **8**(3): 204.

Zdonik, Z. B. and D. Maier, Ed. (1990). Readings in Object-Oriented Database Systems. The Morgan Kaufmann Series in Data Management Systems. San Mateo, Morgan Kaufmann Publishers, Inc.

Chapter 2

The YANUS datamodel

In chapter 1, the principle of using schema definitions to specify interfaces to the user and to external packages has been introduced. In this chapter the *data model* is described and formally defined, i.e.: "What can be defined in a schema, and what is the meaning of such a definition?"

To clarify what is meant by the "data model", the following parallel may be used: In the world of programming languages, the definition of the syntax (i.e. grammar) and semantics (i.e. how a certain expression in the programming language will be executed) may be compared to the data model, while the program may be compared to the schema (Neuhold, personal communication). The status of a running program corresponds to the objects, workspaces etc. that at a certain moment exist in agreement with the corresponding definitions in the schema; this level below the schema will be called the "instantiation level".

A major question is: how powerful must a data model be? What kind of environment should one[1] be able to define using a schema?

The data model must be sufficiently powerful to define in the schema an interface for interactive analysis of data from a database as described in chapter 1; more specifically, it must be powerful enough to define an interface for interactive pattern recognition and simple arithmetic operations on the data in a population table as described in section 1.4. A special point of attention is the organization of the interface into separate modules /workspaces; the foundations for that feature are laid down in this chapter, but further details will be discussed in chapter 3. Moreover, the data model must be powerful enough to describe its own types and operations through metatypes. As will become clear in chapter 4, and as already anticipated in chapter 1, the description of external software packages through so-called implementation schemas is not much different from the description of operations and types of objects that the user may use; therefore, this does not impose extra requirements on the data model. Special requirements are imposed by the fact that, when a user makes a request for the execution of an operation, the execution of that request must be delegated in a rather complex way

[1] A defining user or "integration implementor"

(including creating the right data representation and sending the right commands) to an external software package. Again the foundations for that feature are laid down in this chapter, but the mechanism itself is elaborated in chapter 4.

In addition to a discussion of the basic principles of the data model, a formalization of the data model is described in this chapter. The formalization is described in *italics*; the discussion refers to the formalization in the appendix.

2.1 Background and requirements

2.1.1 Introduction

The basic idea underlying this discussion is, that a system may support a user interactively editing complex object[2] structures and applying operations to these objects. In the technique used, each object is linked to a type object (or simply type) which contains information about the allowed structure of the object and about the applicable operations. A simple interpreter may guide the user in manipulating the objects correctly, i.e., in conformation with the information in the type object. The idea also permits types (type objects) to be created and edited using this same technique, which means that types (so-called metatypes) describing these types are also needed. In that way, such an editing environment may be extended using itself. This idea originates from the UDPX- project (Meijler 1985), in which the author took part. Note that the information in a schema as mentioned earlier contains primarily such type information as is described here.

Much has been written about data models. It is not the purpose of this section to give an overview. However, specific data models provide features that are useful, and even necessary in the context of YANUS as given above and in chapter 1. The most relevant features and the reason for their relevance will be discussed below. Since the YANUS data model is basically an object-oriented data model, much use has been made of overview articles about object-oriented data models (Stefik and Bobrow 1986; Atkinson, Bancilhon et al. 1989; Zdonik and Maier 1990).

[2] The term "object" has not yet been defined. The possibly better known and data model independent term "data structure" could also be used here. However, an object model will indeed be employed. Therefore, for the sake of continuity and simplicity the term object will already be used here.

2.1.2 Object Identity

In the literature a general consensus is that objects are data entities that have an identity independent of their respective values. (Khoshafian and Copeland 1986; Atkinson, Bancilhon et al. 1989; Zdonik and Maier 1990) and others. In the context of this thesis, this notion is important. One of the reasons is, that for the user who is interactively manipulating objects the position of a particular object within an object structure is of more importance than its current value. This position is best modelled by the way in which the other objects refer to it, and referencing can best be modelled by using object identity: if the value of the object changes, the other objects are still referring the same object.

2.1.3 Data and procedural abstraction

Data and operations must "hide" their implementation. This is called data abstraction and procedural abstraction, respectively. Abstraction is an important concept in computer science (Abelson, Sussman et al. 1991). It is introduced into a system to shield the user or programmer from unnecessary details.

Data abstraction provides the user with (abstract) data elements without him knowing in detail how these data elements are represented (Guttag 1977), e.g. a stack hides its implementation as a single linked list. Generally, this means that the interface to abstract data elements is defined in the form of operations, e.g. push and pop for a stack.

Procedural abstraction provides the user with abstract operations that "do something" independent of how these operations are implemented.

One of the reasons for requiring data and procedural abstraction is that data and operations in YANUS are implemented in the underlying resources; the user does not need to be aware of this. E.g., operations applied to a population tree in the direct application area are implemented by the underlying software package, Ispahan. In YANUS, there is a need for two levels of data abstraction:

1. The user is not interested in the implementation of the objects he is directly using, i.e. of the snapshot objects that YANUS maintains.
2. He is even less interested in the fact that these objects do correspond to data in an underlying software package or database: e.g. the population tree corresponds to a similar tree in Ispahan.

Procedural abstraction is especially needed for executing operations using the underlying packages.

These forms of abstraction imply well defined interface definitions. In an object-oriented data model, the user of an object may only use this object via certain pre-defined operations (Stefik and Bobrow 1986; Zdonik and Maier 1990) which are defined as part of its interface, and he therefore does not need, and is not allowed to apply ad-hoc defined operations to the object in which knowledge of the object's implementation is employed.

2.1.4 Dynamic type checking

The interface of legal operations of an object is strongly linked to an object's type or class (Cardelli and Wegner 1985; Stefik and Bobrow 1986; Zdonik and Maier 1990; Kent 1992). Type checking is the mechanism by which the legality of an operation application is checked (Cardelli and Wegner 1985). Since objects are used interactively in YANUS, which means that it is not known beforehand what operations will be applied to certain objects, the focus here is on run-time or dynamic type checking; this is in contrast to type checking at compile time which is called static type checking (Zdonik and Maier 1990). Using only dynamic type checking has important consequences, as in the data model O2 (Lécluse, Richard et al. 1988). This will be expanded upon in the sections about subtyping.

Note that since the user should be **guided** in applying the correct operations, the notion of dynamic type checking is extended to the provision of semantic feedback in the dialogue (see chapter 3), i.e. at any moment indicating the actions (e.g. selection of a certain object or operation) of the user which lead to a correct operation application are indicated.

In this thesis, the relationship type - object plays a key role. Objects which have a certain type are generally called *instances* of that type.

2.1.5 Complex objects

According to general consensus in the application of object-oriented data models for database management systems the model should support complex objects (Atkinson, Bancilhon et al. 1989). This means, at least, that an object may be requested to give access to (a selected set of) objects it refers to (Maier and Stein 1986; Lécluse, Richard et al. 1988).

2.1.5.1 High level associations

Albano et al. (Albano, Ghelli et al. 1991) describe some problems regarding the way in which complex objects are often modelled e.g. (Maier and Stein 1986).

Quote of (Albano, Ghelli et al. 1991):
"Associations between entities are modeled as properties of objects, i.e. as attributes whose values are [references to] the associated objects. [...] When objects in two classes a and b, whose elements have type A and B, are mutually related by an association, an attribute of type B is defined in the object type A, and vice versa, to model two one-directional relations. The constraint that the relation from a to b is the inverse of that from b to a [i.e., bidirectional], cannot be expressed declaratively in the object definitions, but must be coded in the methods which implement the association. [...] this approach has some limitations:

- the associations are conceptually a higher level abstract notion, their implementation should be decided by the DBMS; attributes on the other hand, force the programmer to choose a specific implementation for them;
- the association semantics is split between different objects;
  [...]
- associations are not necessarily binary [possibly between more than two objects] and they can have their own attributes; these aspects can only be modeled indirectly by means of attributes."

As (Kilov 1990) also notes, the meaning of an association cannot be explicitly modelled in the "plain" attribute approach. It is therefore necessary to have a model for higher level associations. Note that high level associations have already been introduced in semantic data models (Hull and King 1987), but have for a while been somewhat neglected in object-oriented data models due to the focus on "behaviour", i.e. allowing operations to be applied to objects.

In several object-oriented data models used in database management, these higher level associations have indeed already been modelled. In the ORION data model (Banerjee, Chou et al. 1987) "composite link" instance variables can be defined; these, however, are not bidirectional. In the VODAK data model (Klas, Neuhold et al. 1990), bidirectional relations can be defined, such that deleting one object does not lead to a dangling reference from the other, i.e. a reference to a non-existing object. Also, a bidirectional component-of relationship may be defined: an object which is a component of another object is deleted if that other object is deleted. Albano also defines a high level object-relationship data model.

In YANUS, modelling bidirectional relationships (here called "cross-reference relationships") and bidirectional component relationships, should at least be possible. This necessity may be derived from the direct application area; e.g. a node in a tree must be a component of the tree, since the node must be deleted if the tree is deleted. A node in a population tree must have cross-reference relationships to the objects describing individuals, since the individuals must not be deleted if the tree is deleted. Since the use of these two modelling primitives are important tools in the definition of hypermedia (Botafogo, Rivlin et al. 1992), these primitives already seem to provide a high level of generality.

2.1.5.2 Generic access

In several data models (Loops (Stefik and Bobrow 1986), ORION (Banerjee, Chou et al. 1987), the COMANDOS data model (Bertino, Negri et al. 1990)), the definition of certain properties automatically results in the availability of operations to read and write these properties. This is a useful feature for YANUS, since it simplifies the creation of new types. This feature must be combined with the feature of high level associations between objects.

Since it is useful to allow the definition of new types "on the fly" in YANUS, i.e., without having to recompile the system, these sort of access functions can best be implemented as "generic" functions, which use parametric polymorphism (Cardelli and Wegner 1985): i.e., general functions that treat the type of the object as a parametrization such that if the type defines a property, it can automatically be accessed by that function on instances of that type.

2.1.6 Subtyping, inheritance, polymorphism and late binding

A well known group of related concepts in object-oriented data models, which is used in the Yanus data model, is that of subtyping, inheritance, polymorphism and late binding.

2.1.6.1 Subtypes and substitutability

A subtype can be created by incrementally modifying a type, its supertype. Cardelli (Cardelli and Wegner 1985) defines types as corresponding to sets; subtypes correspond to subsets. There are several forms of type modification (Wegner and Zdonik 1988). The most general one is where a predicate is used to restrict the set of the supertype to the set of the subtype. An example is Int as a subset of Real. (Only predicates are applicable which allow for some form of "behavioral compatibility", i.e. such that similar operations applied to similar objects in the subtype yield similar results, see below). More specific ones are so-called signature-compatible modifications. Generally used is horizontal (signature) extension where extra properties or attributes are defined on the subtype in comparison with the supertype. Another signature compatible modification is the vertical extension, where certain properties are

restricted in the values that they can reach. An example is the subtype Retiree of the type Person, where for a Retiree the age must be older than 65.

Since the subtype corresponds to a more specific set of objects than the supertype, the user (In YANUS the user, in other systems the programmer) would expect that all operations that may be applied to objects of the supertype may also be applied to objects of the subtype. This is called the principle of substitutability.

The principle of substitutability is defined as (Wegner and Zdonik 1988): An instance of a subtype can always be used in any context in which an instance of a supertype was expected. When allowing for update operations, problems may occur, since the update operation may change the object in such a way, that its type definition is violated: For example, the age of a Retiree could, inadvertently be assigned to be 20. Also, Wegner and Zdonik warn for operations that are not closed on a subtype; e.g. a subtype such as Int(1..10) is not closed under addition, since 6 + 5 is undefined for this type, while + is clearly defined for its supertype int. Moreover, when allowing for very "wild" modifications, e.g. deleting certain properties, certain operations defined on the supertype -even read-only ones- may be undefined on the subtype.

Due to the requirement that a YANUS schema must be able to describe other schemas, it appears that (as will be shown in the rest of this chapter) YANUS type definitions, and the corresponding subtyping mechanisms must also include the use of predicates to restrain the objects in the corresponding set. Thus the problems mentioned here with respect to substitutability and closure are relevant. The use of dynamic type checking will appear to be essential.

2.1.6.2 Inheritance and dynamic binding

Bypassing the problems mentioned above, we shall assume that all operations defined on the supertype must, and will be defined to be applicable to instances of the subtype. Since in non-problematic cases operations do not need to be re-defined, nor re-implemented, the subtype is said to inherit the operations of the supertype. Similarly, since neither the predicates, nor the signature modifications are in general allowed to modify the subtype in such a way that properties are deleted (we shall adhere to this restriction), subtypes inherit properties from their supertypes. These forms of inheritance are used extensively in Object-oriented data models.

However, subtypes may in general redefine or "specialize" the properties and operations they inherit. This has already been seen for vertical signature modification, where a property (in the case of the Retiree: Age) can be redefined to allow for more specific values. Similarly,

operations may be redefined to work specifically on a subtype. This is also called overriding (Stefik and Bobrow 1986; Atkinson, Bancilhon et al. 1989).

In connection to overriding, the feature of *late binding* (Atkinson, Bancilhon et al. 1989) may be needed. This is the case when objects are created at run-time, the common supertype of these objects is known, but the precise type of these objects is not known beforehand (i.e. at compile time). When a common operation must be applied to all such objects, the late binding mechanism is employed to determine the appropriate implementation of the operation to be used for each of those objects. A well known example is that of graphical objects: various objects of various subtypes of "graphical object", such as "square", "triangle" and "circle" will be created at run-time, and it must still be possible (as specified by the supertype "graphical object") to apply the operation "display" to all those different objects, although the implementation of this operation is different for each of the subtypes.

In many programming languages the focus will be on determining statically what operation must be executed (Mugridge, Hamer et al. 1991). In YANUS, since all objects are created at run-time, late binding is the only mechanism.

2.1.7 Generalized object model/multi methods

In the literature a distinction is made between so-called "classical object models" and "generalized object models" (Snyder 1991; Kent 1992; OMG 1992). In a classical object model, there is one target object which has to handle a request for operation execution (this is generally called a message). In a generalized object model a request contains zero or more parameters, any of which might identify an object. The binding of a request to an appropriate implementation (i.e. the late binding mechanism) can be based on the type or class of any or all of the parameters; there is no distinguished parameter. The service might be provided by one of the identified objects, or by a third party (Snyder 1991). In CLOS such a generalized object model is employed (Stefik and Bobrow 1986). Note that the classical object model is a special case of the generalized model, since the binding is only based on the first argument.

Arithmetic operations such as addition and multiplication which have two parameters with conceptually the same meaning, are modelled much better in a generalized model. Moreover,

- a user model in which the user is allowed to first select an operation and then for each parameter the objects, seems appropriate for a general user interface model such as YANUS, and
- the user model of globality of operations across workspaces (chapter 1, section 1.6),

fits better with a form of data models in which operations are not owned by the type, but are modelled separately. This separate modelling of operations corresponds to a generalized object model (Kent 1992).

2.1.8 Meta types

The principle of using meta types has already been introduced in Smalltalk-80. Other programming languages in which meta types are used are, a.o. Loops, CommonLoops, CLOS and ObjVlisp. An overview has been written by Cointe (Cointe 1987).

Cointe shows, that the introduction of meta types raises the level of abstraction and flexibility. This higher level of flexibility is introduced due to the fact that the behaviour of a class (or type) can be defined as the behaviour of a regular object reacting to message passing. The meta type defines how the type reacts to the instantiation message, which requests the type to create a new instance of itself, or how so-called class-variables (variables that may be used by all instances of the class) may be initialized.

Both in the VODAK (Klas, Neuhold et al. 1990) data model and in the model described by Sheth (Sheth and Kalinichenko 1992) meta classes and meta types can be used to define special kinds of types and classes, which again specify specific objects. In VODAK, a meta class may define special types of relationships for the instances of its instances: e.g. one object may be the copy of another, or one object may be a version of another. These special relationships correspond to the way in which certain messages are handled: e.g. if an object which is a version of another object (which may again be a version of some other object) cannot handle a certain message, the message is delegated to the predecessor. By defining this at the meta level, each class which is an instance of such a meta class defines automatically instances which have this special relationship. Sheth describes how a special kind of type (instance of a special meta type) may have a running state machine (i.e. a state machine "in" a certain state) as its instance; the type defines the states of the object, and defines operations and resulting state transitions for the object.

In section 2.1.1 it was indicated that in YANUS it was envisioned to use meta types to be able to extend the system using itself. The possibility, as described above, to make special kinds of types will prove to be very useful too. Moreover, meta types, as already introduced in chapter 1, play an essential role in incorporating genericity in the system. A summary of these applications of meta types will be given in chapter 7.

2.2 The Data model

2.2.1 Values and Objects

The use of YANUS is centred around *objects*. Objects have an identity independent of their value, and refer to their type. Objects have named *properties*. Via a property name, the value of that property can be accessed. This so-called *property value* is either a sequence of identifiers of other objects, i.e., references to other objects, or a sequence of (other kinds of) values. These other kinds of values are usually referred to as *values* in general. Values may be "simple" numbers, or booleans, but may also be constructed[3].

Since the value of a property may either be a sequence of identifiers, or a sequence of values, a generalized term is used: a property value contains a sequence of *elements*.

Since objects are basically also constructed values, a more principal distinction between objects and values must be made. Such a distinction lies in the type checking mechanism (Note, that it is now necessary to anticipate on the definitions of types, operations etc.) The correctness of applying a certain operation to a certain object can be checked since the type can be directly accessed from the object. In contrast, the type of a value can only be known by virtue of its context. Since the value is an element of a certain property value of a certain object, and since the type of that property is laid down as part of the type definition of that object, the type of the value can be inferred.

As a result, the user of the system is only allowed to use objects (and not values) directly, since the use of objects can be regulated through supportive type checking. Values which are attributes of objects can only be entered and changed via the object; operations may not be applied directly to values.

In the following some definitions in the formalization will be discussed formally introducing values and objects. The following general remarks are relevant in that context:

- The idea of type checking on the use of objects cannot yet be introduced. The reason is, that types are also objects; type checking is a level of abstraction where the object

[3] Type constructors for value types are not explicitly modelled in the YANUS data model. This simplification does not result in essential restrictions, since type constructors of the programming language can be used in which YANUS is implemented. Normal procedure is to define a value type formally in VDM, using the type constructors set, tuple, sequence etc. and to map that to the structures in the language.

is used in coherence with its type. First the low- level, type independent -or "generic"- concept of an object must be introduced[4] in terms of its structure.

- Basic generic functions are defined to retrieve elements of the structure of an object. Such functions, but also, e.g., operations to set certain values (see section 2.2.9.3), are not used explicitly by the user, but implement the generic dynamics of the system. From now the overall term *methods* will be used for these kind of functions and operations in general[5]. In contrast, the functions and operations which are explicitly invoked by the user will in general be referred to as *operations*. Such an operation is invoked by means of a request, in which the parameters must be filled in, see section 2.2.8.

The first introductory definitions, defining values and objects are given in definition 1 to 9. First (definition 1), the set Domains is defined which is the set of all sets containing values. Special sets in Domains are Z (the set of integer numbers), $\mathbb{R}$ *(the set of real numbers) and* B *(the set of boolean values), but also (definition 1a) all kinds of sets containing symbols (NAMES, PNAMES, TNAMES, etc.). These symbols may be represented by strings; this will not be worked out. Values from these sets will be used as names for types, properties etc. One special set of symbols, BVTNAMES, contains the names of all sets in Domains (definition 1b), in such a way, that there is a mapping (called domain) from each name in BVTNAMES to each set in Domains. For example, "Real" is a symbol in BVTNAMES, and domain "Real" =* $\mathbb{R}$ *Note that the set BVTNAMES also contains the symbol "BVTname" such that domain "BVTname" = BVTNAMES.*
The set of all values, D, is the union of all sets in Domains.

Next, (definition 2) the set of object identifiers ID is introduced; the set of identifiers is not contained in Domains. In definition 3, the set of element values is defined; this is the union of D, ID and NIL indicating that an element value of a property is either a value (in D) an identifier or the value Nil. More specifically (definition 4), a property value is defined as being either a sequence of identifiers, or a sequence of values. If the sequence is empty, retrieving the first element (see definition 9), results in a returned Nil value.

In definition 5, a definition of an object value is given: An object value is a mapping from a set of property names (PNAMES) to a set of property values. This mapping mechanism also models the access of a property value via its name. In definition 6, an object is defined as

---

[4] Basically, at this point the object is described as a value!

[5] These functions and operations are implemented as methods and sometimes as (overloaded) functions in the host programming language; see chapter 6.

having an object value, a type (which is a reference to another object) and a validity. Validity indicates the level to which the description given by the objects type is fulfilled; this will be explained in further detail in section 2.2.2. In definition 7, the mapping γ from identifiers to objects is given, in such a way that a mapping exists from each identifier in the domain ($i\gamma$) of γ to an object as defined in definitton 6. In definition 8, some special methods are defined for retrieving the object's value, type and identity. In definition 9, more general retrieval methods are defined. The method get retrieves the property value of an object for a given name of the property; it transforms this property value into a set (and thus information about the sequence of the elements is lost). The method singet retrieves the first element of a property value; if the property value is empty it returns Nil. Note that the unguarded use of get and singet may lead to errors, since the signature of these methods (i.e. working on any identifier in $i\gamma$, and on any symbol in PNAMES) does not forbid an attempted retrieval of undefined properties of an object. Thus, get and singet may only be used in the context of higher level methods, which keep track of the type and thus keep track of the properties which have been defined for that object.

2.2.2 The structure of an object as dependent of its type

The next step in describing the YANUS data model is, how a YANUS type describes (or prescribes) the structure of an object in terms of the properties which have been defined for that object or shortly: what properties the object "has", and what kind of elements the corresponding property values are supposed to contain. As will become clear, this definition of structure is one of the most fundamental mechanisms in YANUS: Even in operation application, the operation description may be viewed as a type, describing the structure of a request, i.e., a special kind of object which is used to request the execution of that operation.

In the YANUS data model the correspondence between what a type prescribes and the properties of an object is a generic relationship, i.e. it is valid for each pair of an object and its type, and thus also for the type and its type, a meta type. However, a type describes the properties of its instances through the contents of its own property "PropDescriptors"; this property must again be prescribed by the meta type etc. Meta types describe themselves. In order to break this recursion, types shall first be introduced in an ad-hoc way.

2.2.2.1 Introducing types

An essential feature of the YANUS data model is that types are also objects. In this section, types shall be described in an ad-hoc way: what properties they have, what the element values of these properties are etc. At a later stage, definition of types shall be given through their meta types.

Of relevance here are only valid types (i.e. where the validity is valid, see section 2.2.2.2).

At this point, essential types are:

- Types (in general)
- Value types
- Basic value types
- Property descriptors
- Object types
- Operation descriptors.

Each type in general has a (not more than one) supertype as given in its property "Supertype". If the property is empty the supertype is the *empty type* iϕ. Each type has a name as given by its property "Name". Basic value types and property descriptors together form the set of value types. A basic value type has a name from the set of symbols BVTNAMES. Such a name identifies a corresponding set of values (see formalization section 2.2.1). A property descriptor has a name from the set of symbols PNAMES. Such a name can also be used to access the corresponding property value in an object (see formalization section 2.2.1).

Property descriptors are special types used to describe properties of objects. In addition to the properties "Supertype" and "Name", property descriptors have the properties:

- "EltType", which refers to either a basic value type or an object type
- "Minelt", a natural number
- "Maxelt", a natural number, the value of the property "Maxelt" is equal to or larger than that of "Minelt"
- "Changeable", a boolean
- "Unique", a boolean

The meaning of these properties will be explained in sections 2.2.2.2 to 2.2.2.4.

Extra properties of object types are:

- The property "PropDescriptors". The elements in this property refer to property descriptors as given above. Since names are used to access property values, two property descriptors referred to by the same object type may not have the same name.
- The property "Operations". Elements in this property refer to operation descriptors as introduced above. For reasons not further given here (see chapter 4) two operations

referred to by the same object type may not have the same name. The set of operations referred to by a certain object type will also be called the set of applicable operations.

Operation descriptors are defined at a later stage in this chapter.

In Provisional definition 1 to 3 and the lemma's p1 to p5, sets are introduced from which types, property descriptors, object types etc. are drawn. The sets are:
IO, IO', IO'' sets containing objects (and thus also types)
IT, IT', IT'' sets containing types
IVT, IVT', IVT'' sets containing value types
IBVT, IBVT', IBVT'' sets containing basic value types
ISPD, ISPD', ISPD'' sets containing property descriptors
IOT, IOT', IOT'' sets containing property descriptors

The introduction of these sets is done at several levels, corresponding to the levels of validity of objects in general (and thus types). The first level (prov. definition 1) corresponds to a validity of invalid or higher: at this level '' is used as postfix to the set-name (e.g. IO'', IT''). At this level it is known what properties the objects (types) in the sets have, as given in lemma p2. Lemma p1 gives the subset relationships between the sets. The second level (prov. definition 2) corresponds to a validity of syntactic valid or higher: sets have the postfix ' to their name. At this level, it is known the elements held by the property values are known (lemma p4), e.g. an element in the Supertype property is again a reference to a type. At this level, special access methods are defined (Definitions 10 to 12) retrieving elements from specified properties, using either the get or the singet method (see formalization section 2.2.1). The third level (provisional definition 3) corresponds to valid objects and types. At this level, even more is known about the values of the various properties (lemma p5), e.g. the fact that the value of the property "Maxelt" is equal to or larger than that of "Minelt". The difference between these levels of validity will be explained in detail in sections 2.2.2.2 to 2.2.2.4.

2.2.2.2 Several levels of validity. The first level: satisfying the template

In section 2.2.2.1 (valid) types were discussed, among these valid object types. Object types describe objects. Essential for what follows is that an object which does not refer to a valid object type is useless, since no type checking can be done for it. Objects that do refer to a valid object type will be called *legitimate objects*. When referring to an object in general, the object will assumed to be legitimate.

There are three levels at which an object may satisfy the description of a type. Each higher level encompasses the description at the lower level. At the first level the object has the properties prescribed by the type; the object is said to satisfy the *template* prescribed by the

type. It refers to that type but it is labelled invalid. At the second level, the property values conform to the syntactic description in the type: the object is syntactic valid. At the third and highest level the object satisfies extra requirements (i.e. predicates ranging over the property values) and it is valid. The calculated validity of an object is stored in a special state variable called validity.

At the first level of description the type prescribes the properties of its instances.

At this point, it is appropriate to introduce the creation of new objects using the method New. New is a generic method, i.e., it creates new objects for arbitrary **valid** object types, taking that type as an input parameter, and creates them following the prescribed template. For the object created in this way the following holds:

- It "has" the type which was used as input of the method; i.e. it refers to that type (since the type is valid, the object is legitimate).
- The properties it has (or, more correctly: for which it is defined) are the ones prescribed in the property "PropDescriptors" of the type. The name of a property corresponds to a name of a property descriptor. To be more precise: The name of the property corresponds to the contents of the "Name" property of a property descriptor;
- The property values for these properties are empty.
- The validity is syntactic valid if the type allows for empty property values (this will be outlined in section 2.2.2.3), otherwise the validity is invalid.

In definition 13, New is defined using pre and post conditions as conform to VDM (Jones 1990). The post condition describes (as a predicate) the relationship between the input type iot, the new object, and between the old and the new version of γ. *It ensures that the identity of the new object (inewo') is unique in* $i\gamma$; *furthermore it ensures that in the new version of* γ, *which is called* γ', *a mapping exists between inewo' and an object (in the sense of definition 6) which is defined for all properties corresponding to names of propdescriptors of the input type iot. Note that the new object is an instance of IO''.*

Lemma 1 is the essential lemma with respect to this section, stating that, provided that New is used as defined, each object from IO'' has the properties defined by its type.

2.2.2.3 Syntactic validity: the second level

At the syntactic level a type describes for its instances:

For each property descriptor (held in the property "PropDescriptors" of the type):

- The required type of the elements in the corresponding property value of an instance as referred to by the property "EltType" of the property descriptor.
  Here two cases must be distinguished:
  1. The element type is a member of the set of object types. The elements must be syntactic valid and have as their type that element type or a subtype of that type.
  2. The element type is a member of the set of basic value types which means that the element type prescribes a value. The elements must be in the domain indicated by that value type.
- The number of elements in the corresponding property value. This must be larger than or equal to the property "Minelt" of the property descriptor and less than, or equal to the property "Maxelt" of the property descriptor.
- If the property "Unique" of the property descriptor is set to true, the elements in the property value must have unique (distinct from each other) values.

Testing whether an object syntactically satisfies the description of its own type is done using a generic test on syntactic well-formedness. When the object satisfies this test, its validity state is set accordingly, i.e., it is set to syntactic valid. After each change of a property value of the object (as will be shown in section 2.2.9.3) syntactic well-formedness is tested, and the state is set accordingly. By using the validity state, the user may also at any other moment be informed of the syntactic well-formedness of the object, without having to use the test on syntactic well-formedness for that purpose. When stating that an object "is syntactic valid" this means that its validity state is set in that way and, (see also lemma 2) the object would pass the test on syntactic well-formedness at that moment.

It should be stressed, that an object is set to be syntactic valid, only if it is syntactic well-formed with respect to its own type. A test for syntactic well-formedness with respect to another type exists, but this test cannot be used generically, since it assumes that the object being tested has all properties that are prescribed by the type with respect to which the test is done. As will become clear later, testing syntactic well-formedness with respect to another type makes only sense if that other type is a supertype of the type of the object.

A change in a property value may result in the object's validity set to invalid. Specifically, invalidity may occur if the user exceeds the minimum and maximum number of elements as prescribed by Minelt and Maxelt. This is a necessary aspect of the model: If the user would not be allowed to exceed these boundaries, the contents of a property value could not be changed in case of equality of Minelt and Maxelt, since he would not be allowed to remove or add new objects.

It should be noted that the type of an object is not something that depends in any way on its current state. The type describes what the object is meant to be.

Definition 14 defines the relationships templ_inst_of which stands for "a template instance of" and synt_inst_of which stands for "syntactically an instance of". Definition 15 gives a formal definition of the method synt_wf which tests for syntactic well-formedness: synt_wf requires that synt_wf_wrt is true with respect to the object's own type. For each of the property descriptors of the input type, synt_wf_wrt requires that propreq is true. propreq tests whether the corresponding property value (the property "Name" of the property descriptor is used to access the corresponding property value) satisfies the requirements stated in the general discussion above. Uniqueness is tested using the test unqelt.

Lemma 2 states that if an object satisfies the predicate synt_wf, it is syntactic valid, and reversely. This is built-in in the dynamics of the system as will be discussed in the formalization of section 2.2.9.3.

2.2.2.4 Validity: the third level

The model allows the definition of extra tests, so-called *extra requirements* on a type which must be applied to an object in order to test its well-formedness in general. These tests are used to augment the syntactical description and to state precisely what are valid values of the object. In general, while the syntactic description concerns a single property, these extra requirements will be used to relate property values to each other, e.g. in a date, if the month is february, the day may not exceed 28, or, if the year is a leap year, it may not exceed 29 etc. Other examples of extra requirements will be encountered in the definition of meta types, see section 2.2.4.

In contrast to syntactic well-formedness, well-formedness is not checked at each change of one of the property values. Well-formedness of an object is only checked if the user attempts to apply an operation to an object which requires a well-formed parameter, or if the user saves an object. Well-formedness can only be checked if the object is already syntactic valid (and thus syntactically well-formed); when it satisfies the test, the validity of the object is set to valid. Due to this "lazy" checking of well-formedness, an object may at any moment only be (have the state) syntactic valid, while it is in fact well-formed, that is, the object **could** satisfy the extra requirements.

This lazy mechanism implicates that operations that require well-formed parameters are considered to be applicable to objects that are syntactic valid, since the object may prove to be well-formed after checking.

Definition 16 defines the relationship inst_of, which stands for "instance of". Definition 17 introduces extra requirements and the predicate that checks for well-formedness, wf. In definition 17a extra requirements (erequirements) are introduced as a mapping between an object type and a set of requirements, where requirements are predicates that may be applied to syntactic valid objects (elements of IO'). All extra requirements (allerequirements) is the union of all requirements defined on all supertypes. Like in synt_wf, wf uses wf_wrt which has a type as an extra input parameter. It is called by wf with the object's own type as the value for that parameter. wf_wrt requires that the object satisfies all extra requirements (the predicate: allerequirements) and that for all property descriptors of the type, the elements in the corresponding property values are valid, if they are objects.

Lemma 3 states that if an object is valid, this means that it is well-formed, i.e., it satisfies the predicate wf. Due to the lazy mechanism, the reverse is not true. Again, this is built-in in the dynamics of the system as will be discussed in section 2.2.9.3.

2.2.3 Subtyping

2.2.3.1 Subtyping and subsets, the modification imposed by a subtype

Until this point, subtyping/supertyping has been introduced as a reference relationship between types: a type refers to its supertype. In this section the meaning of this relationship will be discussed, and the requirements concerning the types that participate in the relationship, i.e., the modifications which a subtype may introduce with respect to the supertype. Note that there may only be one supertype.

In sections 2.2.2.2 to 2.2.2.4, three levels at which a type may describe its instance were given. Similarly (reversely) there are three ways in which an object may be viewed as an instance of its type:

- It is always a template instance of its type;
- If it is at least syntactic valid, it is a syntactic instance of its type;
- If it is valid, it is an instance of its type

Essential, however, for these instance relationships in YANUS is that an object is also an instance of all the supertypes of its type (this is so for all three levels). As a result, all instances of a subtype are also instances of the supertype, and thus the instances of the subtype form the same set or a subset of the instances of the supertype. This is conform to the literature (Cardelli and Wegner 1985), i.e. that subtypes correspond to subsets.

Modifications that a subtype may introduce are the following:

I. Syntactical or "signature" (Wegner and Zdonik 1988) modifications

1. Horizontal Extension

   Objects that are instances of the subtype may be prescribed to have extra properties with respect to objects that are instances of the supertype. In other words: the subtype may add another property descriptor with a distinct name. As an example, the type Person1 may be defined to have one property "Name"; a subtype Person, may be defined to have an extra property "Age". Basically, this is a modification at the template level.

2. Vertical modification

   A restriction may be imposed on the possible property values. For this, there are several possibilities:

   a. Restricting the possible element values

   - limiting the type of objects which may be referred to a subtype
   - restricting the domain of values of elements, e.g. for a subtype of Person, Retiree, the age may be restricted to > 65

   b. Restricting the variation in the number of elements; i.e. narrowing the range between the minimum and maximum number of elements

   c. Restricting the possible variation between values of elements, by requiring element values to be unique.

   For all these cases this is done by using an overriding property descriptor in the subtype, i.e. a property descriptor having the same name of the property descriptor of the supertype, which is thus used to access the same property in an instance. For the above three cases this is done in the following way:

   a. The element type (the property "Elttype") is set to a subtype of that of the overridden property descriptor.

   b. The boundaries "Minelt" and "Maxelt" lie between the values "Minelt" and "Maxelt" of the overridden property descriptor.

   c. "Unique" is set to true, while in the overridden property descriptor it is false.

II. (Other) predicate constraints

3. The subtype may add extra requirements

III. Modifying the set of operations

4. A subtype may extend the set of applicable operations; new operations (having distinct names) may be added to the set of operations applicable to the superset.
5. A subtype may override an applicable operation; this is done by referring to another operation description which has the same name as the operation description referred

to by the supertype. This is used to indicate that the overriding operation has a more specific implementation, tuned to that particular subtype (and its subtypes).

Even in cases where vertical modification, or extra predicate constraints are employed, objects are not allowed to change type. Thus, even if the age of an object of the type Retiree would change to 45 (which could only happen using some operation, see section 2.2.3.2; in the normal use of the system the user can be prevented from making such mistakes), the result would only be that the object became invalid.

Definition 18, 19 and lemma's 4 and 5 formalize the principle of subtypes corresponding to subsets. In definition 18, the concept of a model of a type is introduced, which is the set of all instances of the type. In definition 19, a view of subtyping as corresponding at least with a subset relationship between the models is given. Two types have the relationship $\leq_{st}^{1}$ *if the model of the first is indeed a subset of the model of the second. In lemma 4 and 5 it is shown, that the YANUS definitions of inst_of and of subtype_of indeed result in the feature, that if one type is a subtype of another type (by referring to that other type directly or indirectly through the "Supertype" property), they also have the relationship* $\leq_{st}^{1}$*. Lemma 4 prepares for this, by showing that any instance of a type is also an instance of the supertype.*

Definitions 20 to 22 describe relationships $\leq_{st}^{2}$, $\leq_{st}^{3}$ *and* $\leq_{st}$ *which are used to describe correct type modifications.* $\leq_{st}^{2}$ *is a relationship that describes correct modifications in the area of structural description, corresponding to categories I and II above. Notice, that a correct modification of a value type is one where the domain of the new type is a subset of the domain of the old type. The modification of object types corresponds to either adding a property descriptor, or introducing an overriding property descriptor, where the overriding property descriptor must have the similarly named relationship* $\leq_{st}^{2}$ *with the overridden property descriptor. For property descriptors, this relationship corresponds to modifying one of the properties "EltType", "Minelt", "Maxelt" and/or "Unique" in the manner described above.* $\leq_{st}^{3}$ *is a relationship that prescribes that the operations of the subtype must include those of the supertype. See section 2.2.4 for the description of the operation overriding mechanism, which is not revealed by this relationship.* $\leq_{st}$ *is the relationship that indicates that a type is correctly modified with respect to the other type, with respect to structural description and with respect to applicable operations. Finally, lemma 6 states, that for any pair of object types where one type is a subtype of the other through the "Supertype" property, the relationship* $\leq_{st}$ *indeed holds between these types. Notice, that this is stated as a lemma: it is (will be) enforced through meta types.*

2.2.3.2 Subtyping and the substitutability principle

As is clear from the discussion in section 2.1.6.1, the given forms of subtype modification may introduce certain problems with respect to substitutability.

Substitutability purely on the basis of whether an object of the subtype may serve as input is ensured by the modifications given in section 2.2.3.1. This can be seen from the formalization, where it is shown that all objects which are a template, resp. (syntactically) well-formed with respect to the subtype are also a template, resp. (syntactically) well-formed with respect to the supertype, and may therefore still serve as input for operations that expect objects of the supertype.

Substitutability with respect to operations (or actions of the user) that change objects can also be ensured due to the run-time type checking mechanisms:

- Run-time checking of syntactic well-formedness. If an operation or action of the user changes an object so that it is no longer well-formed, this is "noticed" by these testing mechanisms, and the validity of the object is set accordingly.
- Run-time checking of applicability of operations: operations that require syntactic well-formed or well-formed input (how this requirement is expressed will be discussed in section 2.2.8.1) may not be applied to invalid objects.
- Run-time checking on the well- formedness of an object, when an operation requires well-formed input.

Thus typing errors cannot occur.

Still, changes resulting in invalidity of the object must be prevented as much as possible. This is, or can be done in the following ways:

- Creating special integer, real etc. object types in which the constraints on the value of the object's only property (which contains the value of the object) is made explicit in properties (e.g. "Minval" and "Maxval") of the type. This information can be used to prevent the user from entering incorrect values
- Defining, when possible, read-only operations; the result is the output. The user may be allowed to try to overwrite the old value of the input by the value of the output, and can then be warned about possible invalidity of the resulting object;
- Define and implement, when possible, operations that allow types to be closures, e.g., a specialized operation on integers, that produces another integer.

Lemma 7 states and proves that an object which is a template instance of some type iot, (this means that iot may be a supertype of the object's type), has all the properties prescribed by iot. Clearly, this is a result of the fact that properties may only be added or overridden.

Lemma 8 states and proves that an object which is well-formed with respect to a type iot, is also well-formed with respect to the supertype of iot. As part of the proof, this is also proved for syntactic well-formedness.

2.2.4 Introducing Meta types

Now that types have been introduced, and the relationship between a type and its instance has been described, meta types can be introduced as a special kind of object types (see diagram 2.1).

Using the meta types, sets that were introduced before may now be properly defined:

- The set of (legitimate) objects:
  Dependent of the level of validity, objects are template,- syntactic,- or just instance - of object types (see below).
- The set of types:
  Objects that are instances of the meta type "Type_type"[6].
- The set of value types:
  Objects (one could also say: types) that are instances of the meta type "ValueType_type".
- The set of basic value types:
  Objects (/types) that are instances of the meta type "BasicValueType_type".
- The set of (structural) property descriptors:
  Objects (/types) that are instances of the meta type "StructPropDescr_type".
- The set of object types:
  Objects (/types) that are instances of the meta type "ObjectType_type".
- The set of operation descriptions:
  Objects (/types) that are instances of the meta type "OperationDescr_type".

[6] In this text, object and meta types are identified by their names; in the formalization a symbol from ID is used as identifier. Reason for not using names in the formalization is, that types in general cannot be identified by their names: property descriptors and operation descriptions cannot be identified by their names, since similar names are used to indicate subtyping.

Diagram 2.1

InitTypeSet ∈ IOT-set

InitTypeSet =
{ iObject_type, iType_type, iValueType_type, iBasicValueType_type, iStructPropDescr_type, iObjectType_type, iOperationDescr_type, iMetaType_type }

InitPropDescriptors ∈ ISPD-set

InitPropDescriptors =
{ iName, iName', iName'', iSupertype, iEltType, iMinelt, iMaxelt, iChangeable, iUnique, iPropDescriptors, iOwnPropDescriptors, iOperations, iOwnOperations, iPrimalType, iPrimalProp_descr }

InitValueTypes ∈ IBVT-set

InitValueTypes =
{ iBasicValue_type, iBool_type, iPosInteger_type, iTname_type, iBVTname_type, iPname_type }

iObject_type inst_of iObjectType_type
name(iObject_type) = Object_type
iObject_type subtype_of iϕ

iType_type inst_of iMetaType_type
name(iType_type) = Type_type
iType_type subtype_of* iObject_type
propdescriptors(iType_type) =
{ iName, iSupertype }
ownpropdescriptors(iType_type) =
{ iName, iSupertype }
operations(iType_type) =
{ }
ownoperations(iType_type) =
{ }
primaltype(iType_type) = iϕ

```
        iName inst_of iStructPropDescr_type
                name(iName) =  Name
                iName subtype_of* iPrimalProp_descr
                elttype(iName) =
                                iTname_type
                minelt(iName) = 1
                maxelt(iName) =
                                1
                changeable(iName) =
                                false
                unique(iName) =
                                true
        iSupertype inst_of iStructPropDescr_type
                name(iSupertype) =
                                Supertype
                iSupertype subtype_of* iPrimalProp_descr
                elttype(iSupertype) =
                                iType_type
                minelt(iSupertype) =
                                0
                maxelt(iSupertype) =
                                1
                changeable(iSupertype) =
                                false
                unique(iSupertype) =
                                true

iValueType_type inst_of iMetaType_type
        name(iValueType_type = ValueType_type
        supertype(iValueType_type) =
                                iType_type
        propdescriptors(iValueType_type) =
                                { iName, iSupertype }
        ownpropdescriptors(iValueType_type) =
                                { }
        operations(iValueType_type) =
                                { }
        ownoperations(iValueType_type) =
                                { }
```

```
iBasicValueType_type inst_of iMetaType_type
        name(iBasicValueType_type =
                                BasicValueType_type
        supertype(iBasicValueType_type) =
                                iValueType_type
        propdescriptors(iBasicValueType_type) =
                                { iName', iSupertype }
        ownpropdescriptors(iBasicValueType_type) =
                                { iName' }
        operations(iBasicValueType_type) =
                                { }
        ownoperations(iBasicValueType_type) =
                                { }
        primaltype(iBasicValueType_type) =
                                iBasicValue_type
                iName' inst_of iStructPropDescr_type
                        name(iName') = Name
                        iName' subtype_of iName
                        elttype(iName') =
                                        iBVTname_type
                        minelt(iName') =
                                        1
                        maxelt(iName') =
                                        1
                        changeable(iName') =
                                        false
                        unique(iName') =
                                        true

iStructPropDescr_type inst_of iMetaType_type
        name(iStructPropDescr_type) =
                                StructPropDescr_type
        supertype(iStructPropDescr_type) =
                                iValueType_type
        propdescriptors(iStructPropDescr_type) =
                                { iName'', iSupertype, iEltType, iMinelt, iMaxelt, iChangeable, iUnique }
        ownpropdescriptors(iStructPropDescr_type) =
                                { iName'', iEltType, iMinelt, iMaxelt, iChangeable, iUnique }
        operations(iStructPropDescr_type) =
                                { }
        ownoperations(iStructPropDescr_type) =
                                { }
        primaltype(iStructPropDescr_type) =
                                iPrimalProp_descr
```

```
iName'' inst_of iStructPropDescr_type
        name(iName'') =
                        Name
        iName'' subtype_of iName
        elttype(iName'') =
                        iPname_type
        minelt(iName'') =
                        1
        maxelt(iName'') =
                        1
        changeable(iName'') =
                        false
        unique(iName'') =
                        true
iEltType inst_of iStructPropDescr_type
        name(iEltType) =
                        EltType
        iEltType subtype_of* iPrimalProp_descr
        elttype(iEltType) =
                        iType_type
        minelt(iEltType) =
                        1
        maxelt(iEltType) =
                        1
        changeable(iEltType) =
                        false
        unique(iEltType) =
                        true
iMinelt inst_of iStructPropDescr_type
        name(iMinelt) = Minelt
        iMinelt subtype_of* iPrimalProp_descr
        elttype(iMinelt) =
                        iPosInteger_type
        minelt(iMinelt) =
                        1
        maxelt(iMinelt) =
                        1
        changeable(iMinelt) =
                        false
        unique(Minelt) =
                        true
```

```
iMaxelt inst_of iStructPropDescr_type
        name(iMaxelt) = Maxelt
        iMaxelt subtype_of* iPrimalProp_descr
        elttype(iMaxelt) =
                        iPosInteger_type
        minelt(iMaxelt) =
                        1
        maxelt(iMaxelt) =
                        1
        changeable(iMaxelt) =
                        false
        unique(Maxelt) =
                        true
iChangeable inst_of iStructPropDescr_type
        name(iChangeable) =
                        Changeable
        iChangeable subtype_of* iPrimalProp_descr
        elttype(iChangeable) =
                        iBool_type
        minelt(iChangeable) =
                        1
        maxelt(iChangeable) =
                        1
        changeable(iChangeable) =
                        false
        unique(iChangeable) =
                        true
iUnique inst_of iStructPropDescr_type
        name(iUnique) =
                        Unique
        iUnique subtype_of* iPrimalProp_descr
        elttype(iUnique) =
                        iBool_type
        minelt(iUnique) =
                        1
        maxelt(iUnique) =
                        1
        changeable(iUnique) =
                        false
        unique(iUnique) =
                        true
```

```
iObjectType_type inst_of iMetaType_type
        name(iObjectType_type) =
                                ObjectType_type
        supertype(iObjectType_type) =
                                iType_type
        propdescriptors(iObjectType_type) =
                                { iName, iSupertype, iPropDescriptors, iOwnPropDescriptors, iOperations,
                                iOwnOperations }
        ownpropdescriptors(iObjectType_type) =
                                { iPropDescriptors, iOwnPropDescriptors, iOperations, iOwnOperations }
        operations(iObjectType_type) =
                                { iDeriveProperties, iDeriveOperations }
        ownoperations(iObjectType_type) =
                                { iDeriveProperties, iDeriveOperations }
        primaltype(iObjectType_type) =
                                iObject_type
                iPropDescriptors inst_of iStructPropDescr_type
                        name(iPropDescriptors) =
                                        PropDescriptors
                        iPropDescriptors subtype_of* iPrimalProp_descr
                        elttype(iPropDescriptors) =
                                        iStructPropDescr_type
                        minelt(iPropDescriptors) =
                                        0
                        maxelt(iPropDescriptors) =
                                        ∞
                        changeable(iPropDescriptors) =
                                        false
                        unique(iPropDescriptors) =
                                        true
                iOwnPropDescriptors inst_of iStructPropDescr_type
                        name(iOwnPropDescriptors) =
                                        OwnPropDescriptors
                        supertype(iOwnPropDescriptors) =
                                        iPrimalProp_descr
                        elttype(iOwnPropDescriptors) =
                                        iStructPropDescr_type
                        minelt(iOwnPropDescriptors) =
                                        0
                        maxelt(iOwnPropDescriptors) =
                                        ∞
                        changeable(iOwnPropDescriptors) =
                                        false
                        unique(iOwnPropDescriptors) =
                                        true
```

```
        iOperations inst_of iStructPropDescr_type
                name(iOperations) =
                        Operations
                iOperations subtype_of* iPrimalProp_descr
                elttype(iOperations) =
                        iOperationDescr_type
                minelt(iOperations) =
                        0
                maxelt(iOperations) =
                        ∞
                changeable(iOperations) =
                        false
                unique(iOperations) =
                        true
        iOwnOperations inst_of iStructPropDescr_type
                name(iOwnOperations) =
                        OwnOperations
                supertype(iOwnOperations) =
                        iPrimalProp_descr
                elttype(iOwnOperations) =
                        iOperationDescr_type
                minelt(iOwnOperations) =
                        0
                maxelt(iOwnOperations) =
                        ∞
                changeable(iOwnOperations) =
                        false
                unique(iOwnOperations) =
                        true

        iDeriveProperties inst_of iOperationDescr_type
        iDeriveOperations inst_of iOperationDescr_type

iOperationDescr_type inst_of iMetaType_type
        propdescriptors(iOperationDescr_type) =
                        { iName, iSupertype, iPropDescriptors, iOwnPropDescriptors, iOperations,
                        iOwnOperations }
        ownpropdescriptors(iOperationDescr_type) =
                        { }
        operations(iOperationDescr_type) =
                        { iDeriveProperties, iDeriveOperations }
        ownoperations(iOperationDescr_type) =
                        { }
```

```
iMetaType_type inst_of iMetaType_type
        name(iMetaType_type) = MetaType_type
        iMetaType_type subtype_of iObjectType_type
        propdescriptors(iMetaType_type) =
                                { iName, iSupertype, iPropDescriptors, iOwnPropDescriptors, iOperations,
                                iOwnOperations, iPrimalType }
        ownpropdescriptors(iMetaType_type) =
                                { iPrimalType }
        operations(iMetaType_type) =
                                { iDeriveProperties, iDeriveOperations }
        ownoperations(iMetaType_type) =
                                { }
        primaltype(iMetaType_type) =
                                iType_type

                iPrimalType inst_of iStructPropDescr_type
                        name(iPrimalType) =
                                        PrimalType
                        supertype(iPrimalType) =
                                        iPrimalProp_descr
                        elttype(iPrimalType) =
                                        iType_type
                        minelt(iPrimalType) =
                                        0
                        maxelt(iPrimalType) =
                                        1
                        changeable(iPrimalType) =
                                        false
                        unique(iPrimalType) =
                                        true
```

Auxiliary Type definitions:

```
iPrimalProp_descr inst_of iStructPropDescr_type
        name(iPrimalProp_descr) =
                                PrimalProp_descr
        supertype(iPrimalProp_descr) =
                                iϕ
        elttype(iPrimalProp_descr) =
                                iϕ
        minelt(iPrimalProp_descr) =
                                0
        maxelt(iPrimalProp_descr) =
                                ∞
        changeable(iPrimalProp_descr) =
                                true
```

```
        unique(iPrimalProp_descr) =
                                    false

iBasicValue_type inst_of iBasicValueType_type
        name(iBasicValue_type) =
                                    Basic_val
        Basic_val subtype_of* iϕ

iBool_type inst_of iBasicValueType_type
        name(iBool_type) =      Bool
        supertype(iBool_type) = iBasicValue_type

iPosInteger_type inst_of iBasicValueType_type
        name(iPosInteger_type) =
                                    PosInteger
        supertype(iPosInteger_type) =
                                    iBasicValue_type

domain PosInteger = ℕ

iReal_type inst_of iBasicValueType_type
        name(iReal_type) =      Real
        supertype(iReal_type) = iBasicValue_type

iChar_type inst_of iBasicValueType_type
        name(iChar_type) =      Char
        supertype(iChar_type) = iBasicValue_type

iString_type inst_of iBasicValueType_type
        name(iString_type) =      String
        supertype(iString_type) = iBasicValue_type

iTname_type inst_of iBasicValueType_type
        name(iTname_type) =     Tname
        supertype(iTname_type) =
                                    iBasicValue_type

iBVTname_type inst_of iBasicValueType_type
        name(iBVTname_type) = BVTname
        supertype(iBVTname_type) =
                                    iBasicValue_type
```

iPname_type inst_of iBasicValueType_type
name(iPname_type) = Pname
supertype(iPname_type) =
iBasicValue_type

(See also definition 1b).

Auxiliary methods:

ownpropdescriptors : IOT` → ISPD'-set
ownpropdescriptors(iot) ≙ get(iot.OwnPropDescriptors)

ownoperations : IOT` → IOD'-set
ownoperations(iot) ≙
get(iot.OwnOperations)
IMT` = M'(iMetaType_type)
primaltype : IMT` → IT`
primaltype(imt) ≙
if(singet(imt.PrimalType) = Nil) then iϕ
else singet(imt.PrimalType)

Extra requirements:

req_{bvt1} : IBVT` → B
req_{bvt1}(ibvt) ≙ domain name(ibvt) ⊆ domain name(supertype (ibvt))
erequirements(iBasicValueType_type) = { req_{bvt1} }

req_{spd1} : ISPD` → B
req_{spd2} : ISPD` → B
req_{spd3} : ISPD` → B

req_{spd1}(ispd) ≙ elttype(ispd) inst_of iObjectType_type ∨ elttype(ispd) inst_of iBasicValueType_type
req_{spd2}(ispd) ≙ maxelt(ispd) ≥ minelt(ispd)
req_{spd3}(ispd) ≙ (elttype(ispd) inst_of iObjectType_type)
=> unique(ispd) = true)

erequirements(iStructPropDescr_type) = { req_{spd1}, req_{spd2}, req_{spd3} }

req_{iot1} : IOT` → B
req_{iot1}(iot) ≙
∀ ipd_1 ∈ propdescriptors(iot) •
¬(∃ ipd_2 ∈ propdescriptors(iot) •
$ipd_1 \neq ipd_2 \wedge$ name(ipd_1) = name(ipd_2))

req_{ot2} : IOT` → B
req_{ot2}(iot) ≙
    ∀ iod_1 ∈ operations(iot) •
        ¬(∃ iod_2 ∈ operations(iot) •
            $iod_1 \neq iod_2 \wedge$ name(iod_1) = name(iod_2))

req_{ot3} : IOT` → B
req_{ot3}(iot) ≙
    ∀ ipd_2 ∈ propdescriptors(supertype(iot)) •
        ∃ ipd_1 ∈ propdescriptors(iot) •
            ipd_1 subtype_of* ipd_2
            ∨
            $ipd_1 \leq_{st}^2 ipd_2$

req_{ot4} : IOT` → B
req_{ot4}(iot) ≙
    ∀ iod_2 ∈ operations(supertype(iot)) •
        iod_2 ∈ operations(iot)

erequirements(iObjectType_type) = { req_{ot1}, req_{ot2}, req_{ot3}, req_{ot4} }

req_{t1} : IT` → B
req_{t1}(it) ≙ let imt = type(it)
    it subtype_of* primaltype(imt)
req_{t1} ∈ erequirements(iType_type)
■

Diagram 2.1. Definitions of important meta types and types

The method:

name()	accesses the value of the property "Name"
supertype()	accesses the value of the property "Supertype"
propdescriptors()	accesses the value of the property "PropDescriptors"
ownpropdescriptors()	accesses the value of the property "OwnPropDescriptors"
operations()	accesses the value of the property "Operations"
ownoperations()	accesses the value of the property "OwnOperations"
primaltype()	accesses the value of the property "PrimalType"

etc.
it1 subtype_of* it2 indicates that it1 is a subtype of (directly or indirectly) it2
If the property "PropDescriptors" of the type iType_type contains the property descriptors iName and iSupertype, having the names "Name" and "Supertype" respectively, this defines that instances of iType_type have the properties "Name" and "Supertype". If the property "Elttype" of the property descriptor iName has the value iTname_type, this defines that the corresponding property "Name" of an instance of iType_type must have element values which lie in the domain of iTname_type (which is TNAMES, not in this diagram). see further the text.

Also may be defined:

- The set of meta types
  Objects (/types) that are instances of the meta type "MetaType_type".

As a result, everything that was assumed to be true about those sets, especially in section 2.2.2.1, is indeed enforced by the meta types given the template mechanism and the rules for well-formedness. Some examples:

- Following the property descriptors of "ObjectType_type", an object type does indeed have the properties "Name", "Supertype", "PropDescriptors" and "Operations", and also others which will be discussed in section 2.2.4.1 and 2.2.5.
- Following the property descriptor named "PropDescriptors" of "ObjectType_type", the element value of that property in an object type must indeed be an instance of "StructPropDescr_type", and is thus an element of the set of structural property descriptors as defined above.
- Following one of the extra requirements of "StructPropDescr_type", the value of the property "MaxeIt" of an instance is equal to or larger than the value of the property "MineIt".
- Following one of the extra requirements of "ObjectType_type", all property descriptors in the supertype correspond to a property descriptor in the subtype, which is either the same, or a subtype of that property descriptor.

Meta types conform to their own description: meta types themselves are instances of "MetaType_type", but since "MetaType_type" is a subtype of "ObjectType_type", they are also instances of "ObjectType_type" (and thus object types). As can be seen from the definition, all meta types do indeed have the properties described by "ObjectType_type", and the property described additionally by "MetaType_type". It can also be seen that meta types (including "ObjectType_type") have been defined in conformance with the allowed type modifications between subtypes and supertypes, as is enforced by the extra requirements defined on "ObjectType_type"; e.g. all property descriptors described by "Type_type" have a corresponding (equal) property descriptor in "ObjectType_type", etc.

Definition 23 a-c are the same as those given in diagram 2.1. req_{ot3} and req_{bvtl} enforce a relationship between subtype and supertype so that the $\leq_{st}^{2}$ relationship is satisfied. Through req_{ot4} the $\leq_{st}^{3}$ relationship is satisfied. Thus lemma 6 is certified. In definition 24, the final definition of the sets IO'', IO', ..., IOT which were introduced in section 2.2.2.1, and of IMT'', IMT', IMT is given. Lemma 9 states that all meta types are well-formed (that is, with respect to their own type). This can directly be seen from the definition.

2.2.4.1 Supporting correct subtyping: Inheritance

If a subtype overrides some property descriptors of the supertype, and adds others, it must also define the property descriptors of the supertype that were not overridden. This also goes for operations. It is therefore logical, as already discussed in section 2.1.6.2, to let subtypes inherit the property descriptors and operations from their supertype, i.e., the subtype does not need to re-define or re-declare the properties and operations of the supertype if these are not overridden.

This is supported in object types by distinguishing between the property descriptors (contained by the property "PropDescriptors") in general (i.e. all property descriptors), and the object type's own property descriptors (contained by the property "OwnPropDescriptors"). Similarly, by distinguishing between the properties "Operations" and "OwnOperations". A defining user only needs to fill in the properties "OwnPropDescriptors" and "OwnOperations"; special operations (as defined on "ObjectType_type": DeriveProperties and DeriveOperations) can be used to derive the complete set of property descriptors and operations. Note, that this form of inheritance is not dynamic, i.e., adding new properties to the supertype is not possible if the properties of the subtype have already been derived.

The inheritance of operations will be discussed in more detail in section 2.2.8.2, in connection to the subtyping of operations.

In definition 25, the methods mk_propdescriptors and mk_operations are defined. mk_propdescriptors derives the total set of property descriptors for the subtype given the set of property descriptors of the supertype and own property descriptors of the subtype. mk_operations similarly derives the total set of operations for the subtype, given the operations of the supertype and own operations of the subtype. Note that ownpropdescriptors was already defined in definition 23. Using these methods, req_{ot3} and req_{ot4} may be reformulated. Moreover, the post condition of the operation (as defined in definition 23) DeriveProperties may be given. Note that as a result of req_{ot2} and req_{ot4} overriding of operations is indirectly determined: When an operation in "ownoperations" of a certain type has the same name as one of the operations of its supertype, then it must be subtype of that other operation. Otherwise, due to req_{ot4}, it would become part of the operations of the subtype, together with that same named operation, as a result of which req_{ot2} would be violated.

2.2.5 Meta types and primal types: defining operations, properties and methods applicable to the instances of the instances of the meta type

In this model, as in the VODAK data model (Klas, Neuhold et al. 1990) and in the model described by Sheth (Sheth and Kalinichenko 1992), a meta type cannot only be used to describe types and to treat types as objects, but also to indirectly define properties and behaviour for all objects that are instances of these types. Here, this is done through primal types.

Each meta type has a so-called *primal type*. Types which are instances of a certain meta type, are all subtypes of its primal type, and thus inherit property descriptors and operations from this primal type. Through this inheritance of property descriptors and operations from the primal type, the type's instances are affected following the normal inheritance principle.

A particular relevant example in the direct application area is that of table types: By defining a meta type of all table types, the primal type of all table types may define operations such as select and project which are applicable (and generic) for all table types. Newly defined table types will inherit these operations, which are thus applicable to the corresponding table. This is further detailed in chapter 5.

An example which can already be seen from diagram 2.1, is that "Object_type" is the primal type of "ObjectType_type". This means that all object types are subtype of "Object_type", and that thus all legitimate objects are (template) instances of "Object_type".

To understand other relevant applications of meta types and primal types in this thesis, it should be realized that special methods (methods such as tests for well-formedness, creation of new objects etc., see the formalization) may be defined to be applicable to the set of instances of a primal type[7], thus introducing special behaviour applicable to those objects. Since the types of all objects to which such a method is applicable are all instances of the same meta type, the method may incorporate knowledge about the structure of the type (as defined by the meta type) and thus access the information in the type in order to adapt its behaviour. This is called genericity; no special method needs to be implemented for each different type. Generic creation of objects through the method New (see formalization definition 13) is a simple example of such genericity: those properties are initialized that correspond to property descriptors in the object type. Genericity is employed throughout the YANUS system, e.g. for executing a request for an operation by means of underlying software

[7] In this thesis, this is done (as e.g. for ISPD and IOT, etc.) by introducing a special VDM set, corresponding to the set of instances of that primal type and defining methods on that set.

packages, as described in chapter 4. It is the technological basis for the extensibility of YANUS.

One form of genericity which is sometimes used is the so-called bottom up testing, i.e. where a method behaves differently depending on the meta type of the type. For instance, in testing whether the elements of a property satisfy the element type as indicated in the property value, a difference is made between a case where the element type is a basic value type and a case where the element type is an object type. Which case is appropriate, is tested on the basis of the meta type of the type. Another possibility could have been, to overload the test "syntactic instance of", defining two implementations, one for testing whether the element value is in the domain of the basic value type, and one for object types, having the usual meaning, as defined in 2.2.2.3. These two implementations could be inherited from the primal type of the meta types of "ObjectType_type" and "BasicValueType_type", respectively. Bottom up meta type it is not objectionable since the meta type hierarchy is not likely to change very much. It is sometimes (implementation technically) the only possibility[8].

In definition 23, the property "PrimalType" is defined through a property descriptor of iMetaType_type. req_{11} is sufficiently powerful to introduce the primal type mechanism described above: it requires that a type, which is an instance of a certain meta type, is always a subtype of that meta type's primal type (it may also be that primal type itself).

2.2.6 Relationships

From now on, extra features in the data model will be introduced by defining new, or altered meta types, and by describing what this means for the corresponding types and the instantiation levels. In fact, no other mechanisms will be used than those described in section 2.2.5.

An example of such an extension is the introduction of bidirectional relationship properties (see also section 2.1.5.1). To that end, different types of properties, as corresponding to different types of property descriptors are introduced. The different meta types of which these different property descriptors are instances are all subtypes of "StructPropDescr_type".

One type of property is an attribute, corresponding to an attribute descriptor. An attribute contains basic values; attribute descriptors are property descriptors but have an element type

[8] In the example, methods should have been made applicable to values, which is not possible.

which must be a basic value type. All attribute descriptors are instances of "AttributeDescr_type". Within the properties that contain references a difference is made between:

- (Simple) reference properties. Reference properties correspond to reference property descriptors, which are instances of "ReferenceDescr_type".
- Relationship properties, which may be subdivided in:
  - Component properties which correspond to component descriptors, instances of "ComponentDescr_type".
  - Cross reference properties, corresponding to cross reference descriptors, instances of "CrossRefDescr_type".

If a certain object has a relationship property, this means that each object that it refers to via this property refers back to this referring object. Each (legitimate) object has the special (simple) reference properties "SuperObj" and "RefObjs" through which it refers back to the objects that refer to it via a relationship property:

- If an object X has a component object Y, i.e. it refers through a component property to object Y, object Y refers back to X via its property "SuperObj". X is also called the *super object* of Y. Y can be a component of one object only.
- If an object X refers to an object Z via a cross reference property, Z refers back to X via the property "RefObjs". Since several objects may reference to one object (e.g. to Z) through cross reference properties, "RefObjs" may contain several back references.

Components are "owned" by their super object: if the super object is deleted, the component object is deleted as well (see section 2.2.9.3).

As may be clear from the previous discussion, relationship properties and the corresponding back references do not have an equal importance. Objects which are referenced through relationship properties are considered to be part of the "structure" of the referring object. This is not so for back references. Specifically, the (syntactic) validity of an object depends on the validity of the object it refers to through relationship properties. It does not depend on the objects it refers back to. Thus, a component object may well be syntactic valid, while its super object is not. The reverse is not possible.

Thus, back references are not "first class" references (the user does not notice them), but they are useful for the following purposes:

- If the user attempts to delete an object, it may be inferred whether there is another referring object which does not allow this object to be deleted; i.e. that other referring object refers to this object via a non- changeable relationship property.

- At deletion of an object, all objects that refer to it must remove their reference, in order to prevent the occurrence of dangling references.
- If the validity of an object changes, either from invalid to syntactic valid or from syntactic valid to invalid, or if the object is deleted, all referring objects must recalculate their validity (section 2.2.9.2).

The coherence between the elements of certain relationship properties, i.e. which objects are referenced in them, and the corresponding back references, is laid down in the rule for syntactic well-formedness. Both the rule for syntactic well-formedness and the rule for well-formedness incorporate that an object's (syntactic) validity does not depend on the validity of the objects it references through back references (and through all other simple references), although it does depend on the validity of objects it references through relationships. Both these rules test on the meta type (the type of the property descriptor) in order to determine what must hold true for a certain property, and thus use the bottom up meta type testing mechanism as mentioned in section 2.2.5.

The rule for syntactic well-formedness forbids the occurrence of (dependency) cycles, i.e. it forbids that an object A has a cross reference to another object B, of which object A is directly or indirectly a component. If this is not prohibited, the following situation might occur: suppose that the validity of A depends on B, as well as on an object C, while the validity of B also depends on that of A. If C is invalid, A is also invalid and B is invalid too. When the validity of C changes into syntactic valid, A cannot become syntactic valid because of its dependency on B, while B cannot become syntactic valid because of its dependency on A.

Definition 26 introduces the meta types iReferenceDescr_type and iAttributeDescr_type and the corresponding primal types. iReferenceDescr_type is the type of (simple) reference descriptors. Next, in definition 27, the properties "SuperObj" and "RefObjs" that all objects have are introduced, by defining the corresponding property descriptors for iObject_type. In definition 28, the meta types iCrossRefDescr_type and iComponentDescr_type are finally defined. In definition 29 the proper meta type and supertype are given of the property descriptors of the set InitPropDescriptors (see definition 23). In definition 30, the methods propreq and wfprop are redefined to take relationships into account as described above in the general text. In the method propreq (using the definition of allbackrefs, which is the set of all objects directly or indirectly referred via back references), the prohibition of dependency cycles is incorporated: an object referenced through relationship properties may not be an element of the set given by allbackrefs. These methods are used by the methods synt_wf (see definition 15) and wf (definition 17), respectively for testing on (syntactic) well-formedness.

Note, that although objects referred to by a reference property are not checked on their validity, a reference property must still contain the correct number of references, etc.

2.2.7 Value propagation

As described by Banerjee et. al. (Banerjee, Chou et al. 1987), the data model supports a form of value propagation: Value propagation is the mechanism through which values of properties of an object are by default propagated to the properties of the components. A well known example is the following: A car body is described as having components such as doors, a chassis etc. The colour of the car body is by default propagated to these components. Relevant applications of this mechanism in this thesis will be encountered in chapter 3.

Here, as in the model of Banerjee et. al., values can only be propagated if the properties of the super object and the corresponding property of the subobject have the same name. Moreover, the range of possible values for the property of the super object, as prescribed by the corresponding property descriptor, must be the same as, or smaller than the range of possible values prescribed for the same named property of the component object.

The propagation is realized in a simple way, by using a special object creation method, that uses the super object of the object to be created as an extra parameter. The propagated value of the component object can thus be overwritten afterwards (if the property is changeable, see section 2.2.9), and will not be changed any more if the corresponding property of the super object is changed at a later time. Moreover, the propagation is only stepwise, from super object to component.

Value propagation is implemented by the methods NewComp and Propagate as given in definition 46. (Note that at this point the numbering of the definitions does not follow the sequence of the text). NewComp uses the super object as an extra parameter. The requirement that the range of possible values of the property of the super object must be within (or equal to) the range of possible values of the property of the component object, and that the names must be equal is expressed by the required relationship $\leq_{st}^{2}$ *between the two property descriptors.*

2.2.8 Operations and their parameters

2.2.8.1 Operation description and requests

An operation description describes the structure of a request, an object which is used to request the execution of this operation. Basically, this structure definition corresponds to the signature of the operation. The relation operation description - request is similar to the relationship type - object. The operation description defines parameter descriptors (operand and setting descriptors, see section 2.2.8.2) such that a request contains for each of these descriptors *actual* parameters, such as an object contains property values for each property descriptor in the type. For reasons not further detailed here (see chapter 3), it should also be allowed to define "normal" properties for a request. As a result, filling in the parameters of a request is similar to editing structure of a normal object. The user may submit a request for execution; the request is only executed after having been proved to be well-formed, and/or the validity is set to valid.

For the user, there is an advantage in this approach of letting a request be an object (be it a special kind of object). The user is allowed to "work on" several requests at the same time, and is allowed to submit a request for execution when he considers this to be appropriate. Moreover, requests that were already executed can also be reused, for example in order to try out the same request several times, varying one parameter each time. Technically, it is also advantageous to let a request be an object: the execution method may be viewed as a method that manipulates this object.

As in the definition of property descriptors, parameter descriptors for a certain operation refer through their element type to the required type of that parameter. Object types, on the other hand, refer to operation descriptions. As a result, a kind of bidirectional relationship may exist: An object type which is referenced as being a required type for a parameter, references back to that operation description as being an applicable operation. This is, however, only the case for those parameters which are called *operands*. For other parameters, the so-called *settings*, a type is referenced as being required for that setting, but the operation is not referenced in the set of applicable operations. This difference can be explained as follows: Operands are more important parameters than settings. Operands are subject of the operation, while settings are more like adjectives: they describe in more detail what should be done. For example, when creating a histogram on a population in the direct application area, the sub- population (node in the population tree) is an operand, while e.g. the (integer) number of bins in which the histogram must be presented is a setting. It is clear that in this example creating a histogram cannot be considered to be applicable to an integer number, while it is applicable to a sub-population.

Operation descriptors are introduced in the data model through a special meta type "OperationDescr_type", see diagram 2.2. "OperationDescr_type" is a subtype of "ObjectType_type", such that all operation descriptors are also object types[9]. "OperationDescr_type" prescribes for its instances the usual properties of an object type, and in addition the properties: "OtherPropDescriptors", "OperandDescriptors" and "Setting-Descriptors".

The property "OperandDescriptors" is prescribed to contain operand descriptors. Operand descriptors are special property descriptors. They are instances of the meta type "OperandDescr_type" which is a subtype of "StructPropDescr_type". "OperandDescr_type" prescribes an operand descriptor to have, in addition to the normal properties of a property descriptor, the properties "Sort" and "Oprvldity". The property "Sort" indicates whether the corresponding actual operand serves as input of the operation, as output of the operation, or as both, the latter meaning that it will be changed by the operation. The property "Oprvldity" may have one of the value invalid, syntacticvalid, or valid; it indicates what validity (or: what level of well-formedness) is required by the operation. Thus, if the property "Oprvldity" of an operand descriptor is invalid, the request for the operation may be well-formed (and may thus be executed) even if the object(s) which serve as the corresponding actual operand are invalid. Similarly, if oprvldity is syntactic valid, the request may be well-formed even if the object(s) which serve as the corresponding actual operand are only syntactically well-formed.

The property "SettingDescriptors" of an operation descriptor contains either cross reference descriptors or attribute descriptors.

In the previous sections, the methods for creating new objects and testing on well-formedness were all based on using the "PropDescriptors" property of an object type, defining all properties in an object; the mechanism of inheritance of property descriptors was realized using the properties "OwnPropDescriptors" and "PropDescriptors" of an object type, while in this section special property descriptors, such as setting descriptors, operand descriptors and other property descriptors of an operation type are all contained in special similarly named properties. At the beginning of this section it was stated, that the actual parameters (operands and settings) of a request should be treated similarly as normal properties, and that (thus) parameter descriptors should be treated similarly as normal property descriptors. It is therefore not desirable (also for reasons of minimality of the system) to introduce special methods for creating requests, testing on well-formedness, and implementing the inheritance specifically

[9] Note that thus, an operation may again define a set of operations applicable to its instantiations, which are requests. This is not wrong, but also not natural. It is due to another design criterium, being that the model, and thus the meta type hierarchy should be as small as possible.

using the properties "OperandDescriptors", "SettingDescriptors" and "OtherPropDescriptors" of an operation description. Instead, an operation "DeriveProps" is redefined for operation descriptions (i.e. the operation "DeriveProps" for object types in general is overridden), which calculates the union of the contents of "OperandDescriptors", "SettingDescriptors" and "OtherPropDescriptors", and makes the property "OwnPropDescriptors" refer to this collective set of property descriptors (i.e., including operands and settings). Next it derives, using the normal inheritance mechanism, the property "PropDescriptors". Thus, other than being defined separately, operand and setting descriptors are, in general, not treated differently than other property descriptors. However, similarly as with other more specialized property descriptors such as component and cross-reference descriptors, syntactic (and semantic) well-formedness is somewhat different for operands. In this case this is due to the property "Oprvldity" of the operand descriptor as stated above.

Definition 31 is the definition of the meta type of all operand descriptors iOperandDescr_type. Note, that it is a subtype of iCrossRefDescr_type. req_{opd1} requires that when the sort of an operand is "out", minelt must be 0. As a result, a request can be (syntactically) valid before the output has been filled in by the execution. Definition 32 redefines iOperationDescr_type. It prescribes that all operation descriptions have the properties "OtherPropDescriptors", "OperandDescriptors" and "SettingDescriptors", in addition to the normal properties of an object type. An extra requirement req_{od1} specifies that the property "SettingDescriptors" may only contain cross reference and attribute descriptors. The primal type of iOperationDescr_type is defined: iRequest_type. The extra requirement req_{ot5} is formulated, which requires that for each operation linked to an object type, that object type must occur as (or as subtype of) the element type of an operand descriptor of that operation. Note that this does not forbid the situation that an operation descriptor is **not** *part of the set of applicable operations of the element type of an operand descriptor. This allows the definition of general operations (i.e. having operand types high in the supertype/subtype hierarchy) which are defined to be* ***applicable*** *to object types lower in the hierarchy. This is used in chapter 3.*

Also, the post condition of the operation "DeriveProperties" is given which overrides the post condition of that same operation as applicable to object types. Finally, in definition 33, the re-definition of propreq and wfprop is given, in order to take the occurrence of operand descriptors into account.

2.2.8.2 Subtyping of operations

An operation description f is considered to be a subtype of an operation description g, if each **request** for f may also be treated (and thus executed) as a **request** for g. This is close to the normal subtype-implies-subset principle between object types. With respect to modifications,

as compared to modifications of "normal" object types, a special requirement is, that an operation description of a subtype may not add extra operand descriptors. Thus, a modification of the form I.1 (section 2.2.3) is not allowed. The reason is that, due to the importance of operands, it does not seem logical to treat a request having an extra operand in a way similar to a request of the supertype: This would mean totally ignoring the extra operand when executing the request as a request for the supertype operation. The use of vertical modification (section 2.2.3, subtyping of the form I.2) is more appropriate: A subtype operation is introduced, when it corresponds with a more specific implementation for more specific operands. Since vertical modification is also allowed for output operands, the operation may also produce more specific output. Introducing extra predicate constraints (extra requirements) is also allowed.

This idea of subtyping between operation descriptions (which indeed allows for treating operation descriptions as object types) differs from the "classical" definitions of data type theory (Albano, Cardelli et al. 1985) with respect to functional types. In that theory, a more specific function may be defined and used, if it may be applied in all cases where the supertype function may be applied, i.e. its domain is larger than the domain of the supertype function, and if its possible results (its codomain) lies within the codomain of the supertype function. The approach described above is similar to the approach described by Lécluse (Lécluse, Richard et al. 1988). As in their work, it is made possible due to dynamic type checking: i.e. the run-time verification of the applicability of a subtype operation to an object.

Note, that this form of subtyping between operation descriptions is one of the reasons why it is possible to treat operation descriptions as specific object types.

At this point, it is appropriate to consider the inheritance and overriding of operations in object types more closely: This is laid down in the extra requirement req_{oi4} in diagram 2.1. In contrast to property descriptors, the overall set of operations does not contain the set of operations that is overridden; overriding operations are only contained by the property own operations. An overriding operation must be the subtype (following the requirements described above) of the overridden operation. Thus only operations at the top of the operation type hierarchy are considered to be applicable. The reason for this is that the user should not have to know that there are several similar operations which could be executed depending on the type of the operands; for him there is only one operation, which is in fact the supertype of all operations that could be executed. Part of the request execution mechanism involves therefore (see chapter 4) determining the operation description type in the operation type hierarchy which fits best with the actual operands of the request.

Definition 34 gives an extra requirement for operation descriptions: if the operation description is not a subtype of iRequest_type, the name of the supertype must be the same as the name of this operation description (it is basically the "same" operation). Moreover, for each operand descriptor that is defined by this operation description, there is a corresponding operand descriptor defined by the supertype, or defined by one of its supertypes (and therefore possibly only to be found in the overall set of property descriptors), such that this operand descriptor has the relationship $\leq_{st}^2$ with that operand descriptor, i.e. prescribes a smaller range of appropriate objects.

Note that the inheritance and overriding of operation descriptions is laid down in definition 25 (the definitions of mk_operations, req_{ot2} and req_{ot4}).

2.2.9 Dynamics in the model

2.2.9.1 Unchangeable properties and freezing

In section 2.2.2.1 the property "Changeable" of a property descriptor has been introduced. This can be used to render the value of a property unchangeable. The property value should not be unchangeable from the start, i.e., immediately after creation of the object. An important reason for this is that objects are created empty[10]. The user should be allowed to fill in and enter the property values and/or, operations should be applicable to initialize the object, before its property values are set to be fixed. This for example allows the filling in and editing of type objects, before their use as types in coherence with instantiations. Similarly, in the direct application area, a table that is shared with others is defined to be unchangeable, but such a table should first be initialized by reading in the corresponding external data; only after that should the properties be fixed.

To realize this, the principle of "freezing" objects is introduced. An object has a "Frozen" property which is a boolean. When the object is created, it is not yet frozen, all its properties may be edited; also, operations may be applied to it which might change the properties. Only when the object is set to valid, the object is frozen (i.e. the value of "Frozen" is set to true). As a result, the unchangeable properties can no longer be changed, even though other properties might still be changed. Operations that would change the object can no longer be applied. Once the object has been frozen, it remains frozen.

[10] This allows the creation method to be generic: when incorporating an initialization in the creation method, genericity can no longer be maintained.

Definition 35 defines the frozen property on the type iObject_type; thus all objects may be frozen. Lemma 10 states that all valid objects (i.e. element of IO) that have an unchangeable property are set to frozen.

2.2.9.2 Propagating (syntactic) (in) validity

As stated in section 2.2.6, an object's well-formedness depends on the well-formedness of the objects it refers to through relationship properties. Moreover, as stated in section 2.2.3, syntactic validity or invalidity reflects syntactic well-formedness or not- well-formedness at any time. For these reasons, a change in the validity of an object, either from syntactic valid to invalid or vice versa, as referenced through a relationship property of another object must result in the recalculation of the syntactic validity/invalidity of that other object. If the validity of that other object changes, this must again result in the recalculation of the validity of those objects that refer to that other object. Remember that the objects referring to a certain object through a relationship property are given by the back references. As a result, such changes in validity will be propagated upward in the object structure through back reference links.

The situation for validity is different: since the setting of validity is lazy, propagation of validity in the way described above does not occur. However, when attempting to set the validity of a certain object, it is not sufficient to check whether the objects it refers to through relationship properties are valid, since these objects may not have the validity which corresponds with their well-formedness. Thus, in this case, the mechanism should first attempt to set the validity of these objects; as a result of which an attempt may also be needed to set the validity of the objects they refer to through relationship properties etc. Only if this (recursive) attempt succeeds, the extra requirements may be checked and the validity of this (referring) object may be set if those are satisfied. It could be said, that the syntactic validity (or invalidity) propagates upwards in the hierarchy, while validity is set in a depth first manner.

Definition 36 gives the definitions of the method that attempts to set syntactic validity set_cond_synt_valid and the method that attempts to set validity, set_cond_valid.

2.2.9.3 Other methods

Other methods are used to add, remove, objects or values from property values, change values in a property value, and delete objects. In adding and removing objects from relationships, references and back references are maintained in conformance with syntactic validity. In deleting an object, also the back references play an important role: through these it is checked which other objects refer to this object, and whether these allow the object to be deleted.

Moreover, each change in a property leads to the recalculation of the syntactic validity and thus possibly to (upward) propagation of this change in the validity (see section 2.2.9.2).

Since these methods are generic (type independent), these are the only methods a user needs to edit, create and delete differently typed objects. One other major generic method is used to execute operation requests. This method is defined in chapter 4.

In the appendix several layers of methods are defined; for most of these methods pre- and post conditions are given. pre and post conditions at a higher level use pre and post conditions at the lower level.

Level 1: sequences
Methods that implement the removal, addition or changing of an element from/to/in a sequence.

Level 2: elements in property values: definition 39.
Methods that implement the removal, addition or change of an element in a property value. Standard pre- conditions are based on whether the property is changeable and/or the object is not yet frozen, and on whether the element values must be unique.

At this level, the predicate RestObjSameVal is introduced. This predicate is generally used as part of a description of a certain change in the state of an object, to indicate which part of the object stays the same; this always includes the object's identity. Specifically, the predicate is used to describe that all properties of the object stay the same except the ones named in the name-set which is a parameter of the predicate. It is for example used in all post-conditions at this level.

Level 3: objects (definition 40) and values (definition 41) in property values
At this level a difference is made between objects and values. The specific type of the property descriptor (i.e. Reference descriptor, Attribute descriptor, Cross reference descriptor or Component descriptor) is not taken into account. For adding objects, specifically the pre-condition with respect to its type (corresponding with the element type) is introduced; for values the value must be in the required domain.

Level 4: References (definition 42), attributes (definition 43) and relationship properties, subdivided in cross references (definition 44) and component properties (definition 45)
The different types of property descriptors are taken into account. For relationships, an extra pre-condition with respect to adding a component is that the object is not already a component of some other object; special post-conditions describe adding or removing back

references to or from the property "SuperObj" and "RefObjs". At this level, also a delete method is specified. The pre- condition tests, given the objects that refer to this object, and the property in which they do that, whether the object can indeed be deleted. The post-condition specifies the deletion of all component objects, if they can be deleted. This delete uses a delete at a lower level of abstraction, which merely removes the object from γ.

Literature

Abelson, H., G. J. Sussman, et al. (1991). Structure and Interpretation of Computer Programs. Cambridge, Massachusetts, The MIT Press.

Albano, A., L. Cardelli, et al. (1985). "Galileo: A Strongly-Typed, Interactive Conceptual Language." ACM Transactions on Database Systems **10**(2): 230.

Albano, A., G. Ghelli, et al. (1991). A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. 17th International Conference on Very Large Data Bases, Barcelona, Spain.

Atkinson, M., F. Bancilhon, et al. (1989). The Object-Oriented Database System Manifesto. International Conference on Deductive and Object-Oriented Databases, Kyoto.

Banerjee, J., H. Chou, et al. (1987). "Data Model Issues for Object-Oriented Applications." ACM Transactions on Office Information Systems **5**(1): 3.

Bertino, E., M. Negri, et al. (1990). "An Object-Oriented Data Model for Distributed Office Applications." : 216.

Botafogo, R. A., E. Rivlin, et al. (1992). "Structural Analysis of Hypertexts: Identifying Hierarchies and Useful Metrics." ACM Transactions on Information Systems **10**(2): 142.

Cardelli, L. and P. Wegner (1985). "On Understanding Types, Data Abstraction, and Polymorphism." ACM Computing Surveys **17**(4): 471.

Cointe, P. (1987). Metaclasses are First Class: The ObjVlisp Model. Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), Addison Wesley.

Guttag, J. (1977). "Abstract Data Types and the Development of Data Structures." Communications of the ACM **20**(6):

Hull, R. and R. King (1987). "Semantic Database Modeling: Survey, Applications, and Research Issues." ACM Computing Surveys **19**(3): 201.

Jones, C. B. (1990). Systematic Software Development using VDM. New York, Prentice Hall.

Kent, W. (1992). "User Object Models." OOPS Messenger **3**(1): 10.

Khoshafian, S. and G. Copeland (1986). Object Identity. 1st ACM OOPSLA conference, Portland, Oregon,

Kilov, H. (1990). From semantic to object-oriented data modelling. The First International Conference on Systems Integration (ICSI), Morristown, New Jersey, IEEE Computer Society Press.

Klas, W., E. J. Neuhold, et al. (1990). "Using an object-oriented approach to model multimedia data." Computer Communications **13**(4): 204.

Lécluse, C., P. Richard, et al. (1988). O2, an Object-Oriented Data Model. ACM SIGMOD conference, Chicago,

Maier, D. and J. Stein (1986). Development of an Object-Oriented DBMS. Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), Addison Wesley.

Meijler, T. D. (1985). Enkele Aspecten van het UDPX Systeem. T.U. Delft. (Report)

Mugridge, W. B., J. Hamer, et al. (1991). Multi-Methods in a Statically-Typed Programming Language. ECOOP '91 European Conference on Object-Oriented Programming, Geneva, Switzerland, Springer Verlag.

OMG (1992). Object Management Architecture Guide. Object Management Group, Inc. Framingham, Massachusetts.

Sheth, A. and L. Kalinichenko (1992). Information Modeling in Multidatabase Systems: Beyond Data Modeling. First International Conference on Information and Knowledge Management, Baltimore,

Snyder, A. (1991). The Essence of Objects: Common Concepts and Terminology. Hewlett Packard Software and Systems Laboratorium.

Stefik, M. and D. Bobrow (1986). "Object-Oriented Programming: Themes and Variations." The AI Magazine 6(4): 40.

Wegner, B. and S. B. Zdonik (1988). Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. ECOOP '88 European Conference on Object-Oriented Programming, Oslo, Norway, Springer Verlag.

Zdonik, Z. B. and D. Maier, Ed. (1990). Readings in Object-Oriented Database Systems. The Morgan Kaufmann Series in Data Management Systems. San Mateo, Morgan Kaufmann Publishers, Inc.

Chapter 3

The User Interface model

3.1 Introduction

In the integration described in this thesis, the user interface takes a key position: YANUS may be viewed as a user interface (development) system, in which the resources (data and operations) provided to the user via the user interface may originate from (implemented in) underlying systems. The resources are offered through the user interface in such a way, that they may be optimally combined conform a typing scheme. This chapter focuses on this user interface.

In chapter 2, the data model was discussed, i.e., the modelling of the representation of the external (or reference) world and operations applicable to objects in that representation through types and operation descriptions in a schema. From now on the representation of the reference world will be called the *model world*. From the description in chapter 2 is also clear, that these types and operation descriptions directly specify the conceptual interface of the system, i.e., the interactive use of these objects and operations. This is due to the run-time availability of these types and operation descriptions, which are used to enforce and support the correct interactive use of the objects and operations.

This chapter extends the data model in order to provide for a credible conceptual interface. So-called workspaces are introduced to structure the interactive use of objects and operations. It furthermore shows how the run-time availability of types and operation descriptions can be used to drive the user interface, specifically to drive the dialogue between user and system. Notably, this chapter does not describe the specification and implementation of the presentation of model objects; the assumption is that for this purpose user interface toolkits or widgets (Linton, Vlissides et al. 1989; O'Reilly and Associates 1989; Pausch, Conway et al. 1992) can be employed.

This chapter focuses on technical problems; psychological models of end-users and aspects of cognition, nor empirical evaluation of the interface, are part of this chapter. The chapter does provide a theoretical framework: implementation is the next step.

In section 3.2, the requirements which YANUS must satisfy from the point of view of user interface development and functioning are discussed. Some of these were discussed in previous chapters, but from a viewpoint less focused on the user interface. Next, (section 3.3) some other approaches to user interfacing, resp. other projects are discussed. Standard approaches to user interfacing, such as user interface toolkits and user interface management systems do not easily satisfy the requirements discussed in section 3.2. Some approaches in which the conceptual interface is both of key importance in specification and functioning of the user interface come closer. In section 3.4, the conceptual interface as has been described in chapter 2 is extended to incorporate workspaces and their types, modules. Next (section 3.5), a tentative description is given of a possible presentation and dialogue which may fit the conceptual interface. Section 3.6 shows how a dialogue manager may use the run-time available type and operation descriptions to conduct the proposed dialogue. Some aspects regarding presentation will be discussed in section 3.7, especially how changes in the presentation must be effectuated. Finally, the described approach will be compared to other approaches in which the conceptual interface is of key importance. The YANUS approach appears to offer a richer conceptual model, which may still be extended due to the use of meta types. The conceptual model is, in our view, also more natural.

In certain sections, a description is also given of the formalization. This is done in *italics*.

3.2 Problem description/requirements

Problems that are tackled by, and requirements for the user interface (development) approach of YANUS are the following:

1. Creating a rich environment, where the use is based on a typing scheme
The YANUS approach abolishes the use of separate applications, by creating one environment, which can accommodate the user interface of many different resources (i.e. which is very "rich"), as normally provided by different applications. The problem of creating such an environment has already been mentioned in chapter 1. It lies especially in the complexity: There are principally unbounded sets of differently typed objects and different operations at the user's disposal. How can the user be guided in using these resources correctly? If an arbitrary object is selected, which operations are applicable? If an operation is selected, which objects may be used as input?

2. Powerful possibilities for the user: direct manipulation
The focus of this research is on providing data analysis to be applied directly to the available data resources. That area, and specifically the direct application area (the test integration, see

chapter 1) where interactive pattern recognition is part of the resources, requires a highly interactive user interface. The user should be able to use "direct manipulation" (Shneiderman 1983; Hutchins, Hollan et al. 1985). He should be able to select arbitrary parts of the data interactively, apply operations to such groups of data, view the results, and also be able to use the results for further analysis/manipulation.

Hutchins et. al. (Hutchins, Hollan et al. 1985) state that in a direct manipulation interface the user has the idea of direct engagement, that he is directly manipulating the objects of interest. Foley (Foley 1988) describes the direction in which such interfaces will develop, which he calls "artificial reality". A user may, for example, wear a special set of glasses, through which he sees the model world in three dimensions and use a data glove to manipulate the objects in that world.

From the description of Hutchins et. al. and of Foley, an essential aspect of a direct manipulation interface appears to be that the model world is presented in such a way, that the user may manipulate different relevant objects separately although still within the context of the larger whole. It must still be possible to view a combination of these objects as one compound (complex) object and to manipulate that complex object, in the same way as, e.g., a simulated person may be viewed as a complex object consisting of a hierarchy of different limbs, but as a whole may be given the command to walk (Zeltzer 1985).

3. Ease of specification/extensibility

The user interface of YANUS must be easily extendable/adaptable to a certain application area. It is said that up to 90% of code of a "classical" application can be taken up with pure user interface matters (Pemberton 1990). In order to allow for ease of adaption or extension, this should not be the case for user interface development in YANUS. Much work has already been done in this area, see also section 3.3. Note however, that this requirement of extensibility is severe due to the requirement under point 2.

4. Consistency

Wiecha (Wiecha, Bennett et al. 1990) mentions the problem of consistency of user interfaces across multiple applications. Clearly, this consistency is deeply rooted in the design of the system (see chapter 1), which may be viewed as being one environment or even one application. In point 5, this problem will again be shortly touched upon.

5. Modularization

In points 1 through 4, general problems were mentioned which are tackled in the YANUS user interface approach. In the following points some problems will be mentioned that occur given the context of the data model/conceptual model described in chapter 2.

In chapter 2, the modelling of the model world in terms of complex objects and its use through operations (i.e. of the conceptual interface) has been described. The model as described there is not extensive enough to describe a complete, useful conceptual (user) interface. As yet, the main shortcoming of the model is **a lack** of modularization. All objects together form one large object structure, starting at the level of directories, at a lower level consisting of documents, below that consisting of structured objects such as texts, drawings etc. and at the lowest level of words (or characters), graphical objects, etc. As a result:

- Invalidity of an object at the lowest level leads to invalidity of the overall object structure; the user is not allowed to try out certain manipulations on certain data without changing the rest. This is called a *lack of locality.*

More technically:

- All data and functionality have to be in memory all the time and/or
- Since the user is in principle allowed to create cross references all over the structure, it is difficult to store certain clusters of data separately and maintain the referential integrity of the system (See for example (Haan, Kahn et al. 1992))

As already mentioned, workspaces will be introduced for this purpose. An important matter is the modelling of these workspaces in the data model.

In this context, point 4 above may be reformulated as: maintaining consistency across workspaces. Also the typing scheme as mentioned under point 1 should remain valid across workspaces.

3.3 Related work/background

3.3.1 Layered Model

When discussing specification and functioning of user interfaces, the subdivision of this specification and functioning in parts is of importance. No complete consensus has been reached in this respect in the literature. Here, a layered model shall be used in which most approaches may be captured, although not all approaches will have all layers. The model is close to that of (Wiecha, Bennett et al. 1990), but it also has similarities to the Seeheim (Pfaff 1985) model. The layers are, in sequence from "high" to "low":

1. I/O layer
   This layer controls the physical communication between the input output device and the computer system. Software used at this level is the windowing system.
2. Presentation layer (Presentation or interaction objects)
   Presentation objects react on the low- level events of the windowing system, change their state and corresponding presentation and indicate relevant changes to the lower layers. Examples of presentation or interaction objects are: windows, scroll bars, buttons, list boxes, etc.
3. Dialogue layer (Syntactic layer)
   This layer controls the dynamics of the dialogue with the user; it controls and coordinates the presentation objects. The name dialogue layer may be confusing, since interaction objects may also be said to conduct dialogues, although at a lower level of abstraction.
4. Conceptual layer/model world
   At this layer the model world is maintained; operations are started to manipulate it, its validity is maintained etc. This layer is sometimes also called the semantic layer; that term is not used here since the term semantic validity has already been used in chapter 2.
5. Computation layer
   In this layer operations are executed.

The layers 2, 3 and 4-5 (together) may also be recognized in a "simple" command interface, corresponding with lexical analysis, syntactic analysis (Aho, Sethi et al. 1986) and execution respectively. There is, however, a difference: the correspondence is correct in cases where presentation objects provide parameters for operations to manipulate objects in the model world. Presentation objects may however also directly represent objects in the model world; in such cases changes in a presentation object may be directly communicated to the conceptual layer. The whole "dialogue" for effectuating such a change is conducted by the presentation object itself; and the syntactic layer is not used.

3.3.2 Other approaches

Ease of specification is an important objective in the work that is being done on user interfaces. For the various approaches, the question whether they indeed allow for the easy specification of user interfaces, and in addition satisfy requirements 1 and 2 as given in section 3.2 will be examined.

Major lines of work in the literature are:

1. Work on dialogue specification languages and User Interface Management systems UIMS's (Green 1985; Jacob 1986; Van Den Bos 1988; Wiecha, Bennett et al. 1990).
2. The work on tool kits, i.e., the creation of extensive interaction object libraries (Linton, Vlissides et al. 1989; O'Reilly and Associates 1989; Pausch, Conway et al. 1992).
3. Work on building user interfaces by direct manipulation (Webster ; Van Den Bos and Laffra 1990). This third line basically uses the technology mentioned under 1 and 2. Since specification of the presentation is not the focus of this research, this line will not be discussed any further. Graphical specification of the presentation seems to be a useful mechanism.

The problem in the implementation of a dialogue for a graphical user interface lies in the asynchronicity of such a dialogue (Hartson and Hix 1989): Many tasks (threads) are available to the end-user at the same time; sequencing of each thread is independent of the others. For example, at any time the user may choose to work with any of the presented presentation objects. Asynchronous, multi-thread dialogue is sometimes also called event based dialogue because end-user actions that initiate dialogue sequences (e.g., clicking the mouse button on an icon) are viewed as input events. The system provides responses to each input event (Hartson and Hix 1989). Asynchronous dialogue does not imply concurrent execution in the computational layer: A dialogue which allows this is called a concurrent dialogue.

The dialogue specifications languages described by (Green 1985; Jacob 1986; Van Den Bos 1988) are general in the sense that they can be used both to specify the dialogue at the presentation layer level as well as at the syntactic layer. Systems in which they are employed are called User Interface Management Systems (UIMS's). Such systems are based on the principle that the dialogue should be separate from the conceptual and computational component (In the literature about UIMS's these two layers together are called "the application"). In contrast, tool kits require that the dialogue/syntactic level are implemented together with the lower levels (in that context all levels below the presentation layer are called "the application").

In a UIMS in which the handling of the dialogue is strictly separated from the lower levels, the user interface is defined and functions in terms of dialogue states. This means that only user interfaces can be specified in which the result of actions of the user leads to a next well specified dialogue state[1]. This is for example a problem in the following scenario, which,

[1] Due to the asynchronicity of a graphical user interface, several states may be active at the same time.

according to requirements 1 and 2 must be easily accommodated by a useful user interface specification approach: The user is allowed to dynamically create new objects in the model world. The user may choose the type of the created objects, and may subsequently apply operations to them, or to combinations of them, as dependent of the type. This is for example the case in a conceptual interface where a drawing may be created using differently typed graphical objects such as circles, triangles, squares etc. It is clearly not possible to define each possible combination of objects as a different dialogue state. A solution seems to be to define each type of object as corresponding to a type of dialogue state[23]; that however, would not allow for applying operations to combinations of objects. Hartson & Hix describe this as the difficulty to model semantic feedback in UIMS's. As shown by the example, it may be stated that direct manipulation interfaces cannot be modelled using a description of dialogue states. Both in the specification and in the functioning of the user interface a stronger coupling is needed between the state of the model world and what the user is allowed to do with that model world.

In contrast, incorporating the control (and creation) of the presentation objects in the "application" in tool kits, does give the programmer the freedom to give feedback at the conceptual level when necessary, but this means that the programmer must again program all dialogue himself, which will not be an easy task. Application frameworks give some support in this respect, since the framework already incorporates a dialogue model. In the case of ET++ (Weinand, Gamma et al. 1989) this is a model where the user each time gives commands, as a result of which the model world will be changed. The system can be programmed in such a way that the model world "determines" what the user is allowed to do. This must however be laid down in the programming language of the application framework (in the case of ET++ this is C++); in the author's experience, the principles of such an application framework do not seem to be mastered easily.

Next, some systems will be described where the conceptual interface is of key importance. The user's next possible actions **are** coupled to the state of the model world, while the specification of such a user interface, corresponds basically to modelling that model world:

In various Hypertexts (Akscyn 1988; Marchionini and Schneiderman 1988), the model world is an object structure:

---

[2] As a Frame (Wiecha, Bennett et al. 1990) or interaction tool (Laffra, Van Den Bos et al. 1990).

[3] And thus in terms of the objects in the model world.

Quote of (Akscyn 1988):
"Information about the reference world is chunked in small units, variously called notecards, frames, nodes etc. Units may contain textual information. In Hypermedia systems, units may also contain other forms of information such as graphics, sound and animation. Units of information are displayed one per window. Units of information are interconnected by links. Users navigate in a hypermedia database by selecting links in order to travel from unit to unit."

Each unit of information may be viewed as an object; the possible (navigation) actions of the user, depend on the object on which he is focused and thus depend on the model world. Creating such a model world (called "authoring") is (meant to be) easy. There is, however, no way to specify dynamic structure manipulations such as described in the scenario. The only structure manipulations that are allowed are laid down in the authoring process, which is fixed, and does not incorporate typing of the objects. In other words: there is no meta level at which the possible structure changes and operation applications can be specified.

Several projects, particularly those described in (Hudson and King 1986; Pemberton 1990), describe the model world in terms of typed objects with constraints. A type definition specifies (possibly derived, i.e. calculated) attributes of an object; objects may be complex (i.e. refer to other objects). Object types may define constraints on objects, prescribing how attributes of (possibly different) objects must have related values. The user may edit objects, create new objects, create new relationships between objects, etc. The system constantly evaluates the new situation.

Related to constraint based systems are so-called syntax oriented editors (Teitelbaum and Reps 1981; Backlund, Hagsand et al. 1989) which define possible structures of the model world in terms of grammars; derived attributes and constraints may also be defined. The user may edit and create sentences (or even graphical objects (Backlund, Hagsand et al. 1989)) that conform to the grammar and he is supported to do so correctly.

Constraint based and syntax oriented editors alike do not use the concept of applying operations. In order to derive new information, a constraint may be specified, describing the relationship between the required result and the input. A further discussion will be given in section 3.8.

Notably, in all these systems where the conceptual interface is of key importance, the dialogue does not need to be specified; it serves the conceptual interface of the system.

3.4 Extending the conceptual interface

3.4.1 Introducing workspaces, modules and context

In order to overcome the lack of modularity of the conceptual model described in chapter 2, as mentioned in section 3.2, workspaces are described in this section.

3.4.1.1 The role of workspaces in the conceptual interface

Workspaces (re)introduce the concept of a separate context in which the user can edit certain data and select certain operations. In a graphical interface, such a workspace will therefore be presented as a separate window, having its own menus etc. However, global applicability across workspaces of operations on basis of type, as described in chapter 2, remains possible. Thus while the disadvantages of application oriented architectures are overcome, the advantages of its modularity are retained. Workspaces may be compared-, but are not equal to running applications. Their features will be detailed below.

1. Workspaces in their function of edit spaces for data

As already mentioned in section 3.2, all data accessible to the user, given the data model of chapter 2, may be viewed as one big persistent component hierarchy, from the root object, which contains directly or indirectly all other data, to the smallest data entities such as numbers strings etc. For, a.o., purposes of surveyability, at any moment only parts of this whole hierarchy will be presented[4] to the user. A presented object may hide its contents, i.e., its direct or indirect components are not presented. Such an object is called a *stub*. In order to inspect or edit these contents, the user has to open a workspace, in which these components will be presented. What was a stub before the opening of the workspace, will be subsequently the top of that (presented) part of the hierarchy, and is called the *context object*, or just context of those objects. Objects presented in that workspace (= in that context) may again be stubs, which may also be inspected in a workspace etc. The user may have several workspaces open at any moment. However, the same stub/context object cannot be inspected at any time by more than one workspace. The user may close a workspace, as a result of which the context object becomes stub again. One all-encompassing workspace, the *root workspace* is always open.

[4] By presentation of an object is meant the availability of the object so that it can be edited, or that operations may be applied to it. Similarly, the presentation of an operation allows its instantiation, i.e. allows the creation of a request for that operation.

2. Workspaces encapsulate the edited data from the rest of the model world

A workspace encapsulates the part of the data hierarchy that is edited/inspected:

- Invalidity of an object in that context does not lead to invalidity of objects in the complete component hierarchy that are in principle (indirectly) super objects of that object but which are outside the workspace. Without workspaces, invalidity (or syntactic validity) would propagate all the way up in the hierarchy until the root directory (chapter 2, section 2.2.9.2).
- At closing of a workspace changes effectuated on the data presented in the workspace may either be made persistent (saved), or discarded. The user may thus undo his changes.
- The workspace may have its own properties, which serve as local variables, e.g. counters. When opening a module, the user may also be requested to specify certain parameters (settings) which will also function as local variables in the workspace. The use of this will be further explained in 3).
- The workspace contains so-called *free objects*, i.e. objects that were removed as components from one of the objects in the hierarchy and are therefore no longer part of the hierarchy. However, there may still be cross references to free objects. If a free object is no longer referenced through cross reference relationships by other objects in the hierarchy, it is said to be *completely* free. Free objects are supposed to be contained by the workspace. Free (and completely free) objects held by a workspace are still said to have the same context, being the context object of the workspace. If the workspace is closed, all free objects are deleted; cross references to such objects are cut and therefore not stored[5]. This, however, can only be done if all cross-references can indeed be cut, which is specifically not possible if any one of the referencing properties is a non-changeable property.

Part of the encapsulation a workspace offers is the forbidding of cross-references between objects in different contexts, so that the contents of different workspaces may be easily stored separately (see section 3.2, point 5):

- If certain objects in a workspace have cross reference properties, these properties may not be set to refer to objects outside the context of the workspace.
- An object may only be moved (*migrate*) from its current workspace to another, and thus change context, if it is completely free. Note that when migrating objects, their type remains intact.

[5] The assumption is made, that in general free objects cannot be stored, even if referenced by other objects which are still in the hierarchy.

3. Workspaces in their function of operation palette

Apart from giving access to a certain part of the object hierarchy, a workspace may also give access to a set of operations. In this sense, the workspace may be viewed to be an operation palette: operations presented in this workspace may be applied both to objects presented in the same workspace and to objects in other workspaces. In general, the operations presented in a certain workspace will be particularly appropriate for application to objects presented in that workspace.

When the user creates requests by selecting an operation presented in the workspace, local properties and settings of the workspace may propagate from the workspace to the request, so that the request will by default have the same values for similarly named settings. This feature may for example be used as follows: A workspace specifically tuned to creating and editing drawings existing of different graphical objects such as lines, squares, rectangles, circles, ellipses etc. may have properties or settings[6] such as line thickness, filling pattern, interval for dotted lines etc. When creating a graphical object, a request for a New operation is first instantiated, to which these properties are propagated. By default the object which is created has these values for the corresponding properties.

A workspace may either present objects only, or present operations only, or present both. An example of a workspace presenting objects only is one in which the contents of a folder or directory is presented. An example of a workspace presenting operations only is a calculator. An example of a workspace presenting both is a workspace for pattern recognition: both population trees and pattern recognition operations are presented; see chapter 5.

3.4.1.2 Modelling workspaces through modules

Workspaces are introduced in the data model/conceptual interface through a special meta type, Module_type. The instances of Module_type are so-called *modules*, and instances of modules are workspaces. The primal type of Module_type is Workspace_type; it is the supertype of all modules (see chapter 2) and, a.o., defines the possibility to open a workspace, close it and save the contents, or close it and discard the contents. These are implemented as methods (see chapter 2) which are thus generically applicable to all workspaces.

Modules are modelled as a kind of operations, such that, in order to open a workspace, the corresponding module must be **applied to** the object to be inspected. This can be explained as follows: An operation which changes an object in a certain way (the object is both input and output operand of the corresponding request) may be viewed as an indirect form of editing

[6] Settings must be set before opening the workspace; properties are set in the workspace itself and may continuously be changed

action on that object. Reversely, the composition of editing actions on the (components of the) context object in a workspace may be viewed as the application of an (ad-hoc) operation; after closing the workspace, and saving the result, the object has changed. Modelling modules similar to operations makes it possible to define settings (see chapter 2) that must be specified in order to open the corresponding workspace. These are the settings that were encountered in section 3.4.1.1 points 2 and 3. Furthermore, it allows to have different modules that can be applied to the same conceptual object, so that the same conceptual object may be inspected by means of different kind of workspaces.

Therefore, the type of all modules, Module_type, is a subtype of OperationDescr_type. The primal type of Module_type, Workspace_type is a subtype of Request_type. Modules inherit from Workspace_type the definition of two operands for their instances (according to Module_type, modules are not allowed to define more than two operands): InspectObject and NewInspectObject: The operand InspectObject of a workspace must contain the object (the stub) that is selected in order to be inspected. When the user opens the workspace, NewInspectObject will be made to contain the context object and its (indirect) components which now form part of the presented part of the hierarchy. These contents are a copy of the information referred to (but not presented) by the original stub. All editing changes are effectuated on this copy. If these changes are discarded, InspectObject still holds (refers to) the original data. When saving the contents of the context object, those original data will be overwritten.

The operations and the (type of) objects to be presented in a workspace can, as prescribed by Module_type, be modelled in a module: The module refers to the corresponding operation descriptions and object types. These types and operations are said to be defined or declared within the module[7]. In general, the module will also refer to the Save operation as applicable to the context object, so that the user may use this operation in the corresponding workspace to save intermediary changes (The save operation is, however, applied to NewInspectObject, so that the contents of the object as it was before opening the workspace can still be recovered).

Requests created by selecting an operation presented in a certain workspace are made components of that workspace. In the creation of such a request, value propagation may be used (see chapter 2, section 2.2.7): settings and properties of the workspace (which must be specifically defined in the module) will be copied to the request, if (a.o.) the names and element types of those settings and properties match with those of the request.

[7] The difference between defining and declaring will be given later.

In definition 1, a subset of objects is defined, the so-called conceptual objects as represented by the type iConceptualObject_type. Conceptual objects are used at the conceptual level. Conceptual objects are further subdivided into request objects as represented by the type iRequest_type and data objects as represented by the type iDataObject_type. Data objects represent the model world; the reference Context to the context object is defined for these objects. A special subtype of iDataObject_type, iDataContextObject_type corresponds to the data objects that can serve as stubs and context objects. The boolean attribute Stub is defined for these objects, and the operation Save (iSaveObject) is applicable to these objects.

In definition 2, the type of all modules, iModule_type is defined. Module_type defines that each module owns (contains) certain types through the property Conttypes; these are the types "defined" in the module. Each module contains certain operation descriptors through the properties Contimpops and Contops; these are the operations defined in that module. A module also refers to other operation descriptors through the property CRefops; these are the operations declared in that module. All operations referred to by a module through the properties Contimpops and CRefops will be presented and can thus be selected in a workspace which is an instance of that module. Creatable Module_type is specifically a meta type for which new instances (new modules) can be created. See also the discussion of definition 5. The primal type of Module_type is iWorkspaceReq_type. WorkspaceReq_type is the supertype of all modules, so all modules inherit the following property descriptors and functionality, i.e. all workspaces have the corresponding properties, and the functionality may be applied to all workspaces.

Properties that each workspace has are described by the properties Ownrequests and Freeobjects. The property Ownrequests contains the requests that were created within the workspace, by selecting an operation presenting in that workspace. The property Freeobjects contains the free objects of the workspace. Each workspace has the (actual) operands InspectObject and NewInspect, as described in the text.

Definition 3 defines some operations, such as StubCopy, InspectObject and SaveObject, all applicable to data context objects. StubCopy copies the stored contents of one stub to another. InspectObject reads the stored contents represented by a stub, and creates the internal presentation, i.e. the hierarchy of presented components. SaveObject saves the contents of a context object (the presented part of the hierarchy). These are defined as operations, since their effect depends on the underlying stored representation of the data. For example, in the case of StubCopy for population trees, the contents of the corresponding files must really be copied, since the package Ispahan does not provide a discard, which means that closing the workspace (which also includes finishing the Ispahan session) always results in saving the

data; if the original data are not saved in a separate file, these will always be overwritten. These operations will be used in the following.

Definition 4 defines methods applicable to all workspaces (to the set of instantiations of WorkspaceReq_type). pre_Open gives the pre conditions for opening a workspace. post_Open gives the post conditions. Specifically, the operations StubCopy and InspectObject (as described above) are used to create a copy of the stub, and to make it present its contents. pre_ClosenSave gives the pre conditions for closing a workspace and saving its contents. post_ClosenSave gives the post conditions. Specifically, the operation SaveObject is used to save the contents of the operand NewInspectObject of the workspace; the predicate post_Shrink indicates that the operand in NewInspectObject is shrunk, i.e., its components are no longer presented; StubCopy is again used to copy the saved representation of the operand NewInspectObject to the operand InspectObject, so that the original stored data are overwritten. The workspace and its components (including the free objects and the contained requests) are deleted. pre_ClosenDiscard gives the preconditions for closing and discarding a workspace.

In the description above the following problem remains. Workspaces, as described above, can be used for the inspection of one object only. However, other kinds of workspaces are also needed, having no, or possibly two or more operands. This can be done by creating special subtypes of WorkspaceReq_type, which have corresponding overriding Open and Close methods. A method for opening a workspace with two inspected objects, must, for example, apply the operation InspectObject to both objects.

3.4.1.3 Using the system to extend itself: A workspace for editing modules

As described in chapter 2, the data model incorporates the concept of meta types. Since types describe how objects may be edited and otherwise manipulated through operations, and determine in this way the use of these objects, meta types in the same way determine creating and editing types. This allows the system to be extended using its own interface. As described in the previous sections, special types of workspaces are used in order to edit and manipulate certain types of objects. Such workspaces are defined through their type which is a module. Similarly, a module can be defined so that types and operation descriptions can be edited and manipulated in the corresponding workspace. This module is called "Module_module". In Module_module the meta types are defined, and the operations which may be applied in editing types: the operations for creating new types, operations and modules, the operations for deriving properties etc. To allow for the introduction of this module, the following is further needed:

As with (other) objects, editing of types in a workspace is based on editing parts of the component hierarchy. Therefore, types are embedded in a component hierarchy, not to be confused with the subtype/supertype hierarchy. It turns out to be natural (leading to the smallest design) to let types and operations be components of the module in which they are defined (if another module refers to an operation which is a component of another module, the operation is said to be declared in that other module). Since modules are viewed as a kind of operations, modules themselves are again component of other modules etc. At the root of this component hierarchy is the root module, which itself is a component of the root object mentioned in section 3.4.1.

All types, operations and modules will be edited within one context, that is, within one workspace. This is a result of the restriction that cross references may not exist from one context to another, while there are many cross references between types, both due to the subtype/supertype hierarchy and due to the element type references[8].

Since all types, operations and modules are components of the root module, the root module serves naturally as this context object. Being the context object also means that the root object must serve as the InspectObject operand of the workspace used to edit these types, operations and modules. As introduced earlier, that workspace is an instantiation of Module_module.

The root module is a special module in the sense that it is the only module to which Module_module may be applied. The root workspace (see section 3.4.1.1 point 1) is an (the) instantiation of the root module. Since the root module contains directly or indirectly all other modules, all these modules may be directly or indirectly instantiated from the root workspace.

Definition 5 first introduces the type iEditableModule_type, which is a subtype of iModule_type. Specific for iEditableModule_type is, that no New operation is defined for this type. There is only one instantiation of this meta type, which is iRoot_module. iEditableModule_type defines in its set of applicable operations the operation (module) Module_module. As a result, iRoot_module is the only module which can be edited in a workspace, namely in a workspace that is an instantiation of Module_module. Root_module contains all other modules (including Module_module itself). Root_module itself is contained by Root_object, the root of all objects. Instantiations of Root_module are used to edit folders; Root_object itself is such a folder. One specific instantiation of Root_module is Root_workspace, which is always open and edits Root_object.

[8] One type is set as the element type of properties defined by another.

Definition 6 is the definition of iModule_module, of which instantiations can be used to edit the type hierarchy. Module_module contains all meta types, and defines a.o. the operations to create new (object) types, new operations, new modules and operations to derive the properties and operations for an object type.

3.4.2 An overview of the user's actions

Since each action of the user will be coupled to a dialogue in the user interface, an overview of the user's actions is given as a preparation for the description of the dialogue in section 3.5. All actions will be described in terms of pre-conditions (which must hold before the action) and post-conditions (which hold after the action). Note that all these actions are implemented as generic methods.

3.4.2.1 Actions concerning workspaces

Open a workspace:

pre-condition:

Object in the operand InspectObject (i.e., object which is going to be inspected) must be a stub. Whether an object is a stub or not is given by a boolean attribute of that object.

post-condition:

A new object is created, contained by the operand NewInspectObject of the workspace, which contains (a copy of) all the information referred to by the original stub object. The new object is expanded, i.e. its components etc. are also presented to the user. Both the original object and this object are no longer stubs.

Close a workspace (and save):

pre-condition:

Operands in InspectObject and NewInspectObject may not be stubs. All free objects are allowed to be deleted. If there are any workspaces which are component of this workspace (see section 3.4.1.2: requests created by instantiating operations presented in the workspace are components of the workspace; moreover, certain requests can be workspaces) they may not be open.

post-condition:

The contents of the operand NewInspectObject is saved. The original object refers to these newly saved data and is a stub again. The workspace and its contents is deleted.

Close a worskpace (and discard):
pre-condition:

Operands in InspectObject and NewInspectObject may not be stubs; contained workspaces may not be open.

post-condition:

The operand NewInspectObject is discarded. The workspace and its contents are deleted.

Migrate an object from one current context to another:
pre-condition:

The object is completely free in the workspace corresponding to the current context.

post-condition:

The object is now in the other context (it refers to that context object!) and is a completely free object in the corresponding workspace.

3.4.2.2 Edit actions/operations

The following actions and operations are all actions and operations to create or delete objects, edit their structure etc. Basically, these actions have already been introduced in chapter 2; however in this section a higher level of abstraction for these actions is introduced, which incorporates, a.o., workspaces. The actions at this abstraction level are directly employed by the user via the dialogue described in section 3.5.

In chapter 2, and in the previous sections the assumption was made that the definition of (cross reference, component and attribute) properties implies the possibility to edit the values of these properties. However, in certain cases, the values of certain properties may only be changed by executing certain operations. For example, in pattern recognition the structure of population trees may only change due to the application of certain decision functions. Therefore the applicability of edit actions must be controlled. This is done in a way similar to the control of the application of operations: All edit actions are allowed if an operation called "StandardStructChange" is part of the set of applicable operations. Otherwise, adding values or objects to properties is allowed if an operation "Add" is defined; removing values or objects is allowed if an operation "Remove" is defined; changing values is allowed if an operation "Change" is defined[9]. These specifically defined Add, Remove and Change operations may have specific implementations (see chapter 4), which involves changing the

[9] This mechanism could be further refined to control at a more detailed level what edit actions may be applied; note however that such a mechanism could not (or only in a not very elegant way) replace the property descriptor, since the property descriptor defines succinctly what structures are correct, while a mechanism for controlling editing actions through operation descriptions can only control what structure changes are correct.

external (e.g. stored) representation together with the internal representation of the objects. Similarly, the applicability of the New and Delete actions can be controlled by defining or omitting a corresponding "New" or "Delete" operation as part of the operations of a (data) object type. In fact, when applied to data objects, New and Delete are completely treated as operations, that is, a request is created and executed.

Each action for changing structure has an indirect object of which the structure is changed, it concerns a property of the indirect object (as indicated by a property descriptor, see chapter 2), and the action has a direct object or value that is being assigned, or removed from the indirect object. The control of the applicability of these actions, as described above, must be viewed to provide additional pre-conditions for these actions. Other pre-conditions will be described below. As a general post-condition, after each of these actions the validity of the indirect object is set, depending on whether it is syntactic well formed or not, to syntactic valid or invalid, respectively.

Actions concerning cross reference properties:
Add a cross reference:
pre-condition:

Direct object must be of the right type (as prescribed by the element type of the property descriptor), and must be in the same context as the indirect object.

post-condition:

Indirect object refers to the direct object via that cross reference property.

Remove a cross reference:
post-condition:

The indirect object no longer references the direct object through the cross reference property. (For the rest self evident).

Concerning component properties:
Add a component:
pre-condition:

The direct object must be of the right type, and must be free in the same context as the indirect object.

post-condition:

The indirect object refers to the direct object via that component property; the direct object is a component of the indirect object.

Remove a component:
post-condition:

The direct object is now a free object in the same context. (For the rest self evident).

Concerning attributes:
Change an attribute value:
pre-condition:

The value must be in the right domain (see chapter 2) i.e. in the domain corresponding to the element type of the property (attribute) descriptor.

post-condition:

One of the elements of the attributes is changed to the new value.

Add an attribute value:
pre-condition:

The value must be in the right domain.

post-condition:

The attribute has a new element of that value.

Remove an attribute value:
(Self evident).

Objects can be added and removed as actual value of an operand or setting. Adding and removing settings and operands is similar to adding and removing cross references; however, no restrictions are made with respect to the context of the objects. Settings may also be attributes, in which case values may be changed, added or removed in the same way as described above.

Deleting an object:
(Self evident).

Creating a new object:
post-condition:

The new object is incorporated in the set of free objects of the workspace.

Executing a request:
pre-condition:

The request must be well formed; if it is, it is set to valid.

post-condition:

This depends on the operation; executing an operation is described in chapter 4. All objects which are newly created as output of the operation (referred to by output operands, sort = out, see chapter 2) are incorporated in the set of free objects.

Definition 8 introduces the general operations StandardStructChange, Add, Remove, Change, New and Delete. New and Delete are treated as any other operation, but have a standard (generic) implementation as described by post_New and post_Delete in the formalization of this chapter and of chapter 2.

Definition 9 - 15 give the pre and post conditions of the various actions, and thus corresponding implementations as methods.

Note, that for example StandardStructChange is not applicable to Folders (definition 5), so that the normal generic mechanisms can not be used. Instead, the operations AddToFolder, RmFromFolder are defined to add and remove objects from a folder respectively. In contrary to the generic mechanism, these changes are transferred directly to the storage.

3.5 A tentative description of the presentation and dialogue

This chapter specifically shows the use of run-time available type information, operation descriptions etc. to drive the user interface. Driving the user interface encompasses driving the dialogue between the user and the system. Therefore, a possible dialogue form will be proposed in this section; subsequently section 3.6 describes how a generic (i.e. schema independent) dialogue manager can conduct this dialogue. The proposed dialogue is based on a presentation space paradigm. This will first be discussed.

The focus of this study is not on human factors, and the proposed dialogue and presentation is thus not based on a study of the literature and/or user psychology. Use has been made of, a.o., ideas which may be observed when using the Macintosh (Naiman, Dunn et al. 1992).

3.5.1 Presentation

The dialogue involves objects, properties, operations, workspaces etc. presented on the *virtual screen*. The physical screen may present only part of the virtual screen; the user may make other parts of the virtual screen visible at the physical screen by using scroll bars and the like. Note however, that parts of the conceptual hierarchy which are not presented in terms of

section 3.4.1.1 point 1, are indeed not presented on the virtual screen. Thus, presenting objects or operations etc. in those terms indeed corresponds to presenting those objects or operations on the virtual screen. In the following when referring to the screen the virtual screen is meant.

The presentation of objects, operations etc. is not discussed in detail here. Fundamental, however, is a space paradigm, where an object (or workspace) may take up a certain part of space on the screen; if another object takes up a certain amount of space **within**, it is assumed to be a component of that first object or workspace.

In general, an object is presented in terms of its properties. This is certainly so, if the user is directly allowed to edit these properties. A component property will correspond to a subspace of the space taken up by the object, in which the component objects may be presented. Furthermore, a button is presented for selecting that property; the purpose of this will be discussed later. An attribute is a subspace in which the corresponding value(s) may be edited. A cross reference property is only presented by a button; if the user selects this property the objects referred to by that property will be highlighted. A request is similarly presented; both operands and settings are basically cross references and therefore presented similarly as cross reference properties. The validity of objects and requests is shown.

A workspace encompasses the space taken up by the context object (which again encompasses the space taken up by all its components etc.) and some empty space in which free objects can be presented. A workspace is presented as a window, to which the menus presenting the operations are linked. The workspace will also give some presentation of the local attributes.

3.5.2 Dialogue

1. Standard dialogue for adding and removing objects to and from relationship properties

The user may first select the property of the indirect object. As a result, the objects that are referred to by that property are highlighted. Also, but differently, all other objects presented on the screen which are of the right type and thus eligible for selection are highlighted. Subsequently, the user may either select an object referred to by the property, as a result of which that object will be removed from the property, or he may also select one of the other objects, which will then be added to the property. In case of component properties, removed objects will immediately be presented as a free objects, and added objects will immediately be presented as part of the property. Note that changes in validity of the indirect object will immediately be apparent.

2. Dialogue for adding and removing objects to and from component properties

Objects may be added or removed to and from component properties by the drag and drop dialogue: the user drags an object (the direct object) across the screen. Moving an object out of a component property corresponds to removing that object from that property (the pre-conditions for this action must of course be satisfied). After removing the object, the object is (officially) free. Free objects may, be assigned to a component property using the drag and drop dialogue. Objects and their properties to which this direct object may be assigned are highlighted. If the user drops the object on top of an object and property which cannot refer to that object the object returns to being free: it is rejected. Otherwise, the object is incorporated in the set of objects contained by that property: it is accepted.

3. Dialogue for migrating objects between workspaces

Objects are migrated between workspaces also through the drag and drop dialogue. By default, if not directly assigned as a component, a dragged object becomes a free object in its new workspace.

4. Dialogue for applying operations

The user may either first select an operation followed by the operands, or first select an operand and then an operation.

If the user first selects an operation, a request is created. The request is presented in the workspace of the selected operation. Next the user may add or remove objects to and from the operands and settings of the request, similarly as objects may be added to and removed from cross reference properties. A user can have any number of requests presented in a workspace, and he is not forced to complete the request.

If the user first selects an operand (an object), as a result all operations on the screen that may be applied to this object are highlighted. Next, the user may select one of these operations. As a result, a request is created and an appropriate operand of the request is made to refer to the previously selected object.

A request may be submitted for execution if it is syntactic valid.

5. Dialogue for opening and closing workspaces

Since modules are similar to operation descriptions (section 3.4.1.2) the user first creates a request by selecting a module just as he creates a request by selecting an operation; next, such a request may be opened, which corresponds to opening the workspace.

Each workspace will have a specific button which may be used to close it. The user may then choose whether he wants to save or discard the contents.

6. User events which are not appropriate in the dialogue

In the previous description, each action, as described in section 3.4.2, corresponds to a (more or less) complete dialogue. If the user generates an event which does not fit in the dialogue which he had started, e.g., if he selects an object which is not appropriate for a certain property or operand, he is given the choice to discontinue that previous dialogue and start a new one or to continue, neglecting this last event. Workspaces, however, may be opened and closed at any moment without disrupting the previous dialogue.

3.6 Dialogue management and syntactical objects

3.6.1 The presentation layer and the dialogue layer - syntactical objects

In the dialogues described in section 3.5, two levels of abstraction may be distinguished. This is especially clear in the drag and drop dialogue: At the lowest abstraction level, moves of the mouse are tracked and the presented object is moved accordingly. At a higher level of abstraction, only the start of the drag, and the identity of the conceptual object which is involved (as represented by the graphical object) in the move is of interest. The reaction of the system at this level involves indicating the objects (representing which conceptual objects) and the properties which may accommodate the moving object - at the lower abstraction level this corresponds to highlighting these objects and properties. Subsequently only the end of the drag movement (the drop) is of interest, end the object (representing which conceptual object) and the property to which the object is assigned.

These two abstraction levels correspond with the presentation layer and the dialogue (or syntactic) layer, respectively as described in section 3.3.1.

At the dialogue layer, a dialogue manager is used which coordinates the dialogue; this will be described in more detail later on. In addition to a dialogue manager, so-called syntactic objects are employed in the dialogue layer. Syntactic objects represent conceptual objects (operations, requests, workspaces ...) in the dialogue. They react to certain events, e.g. the first move of an object generated by the presentation objects (other events such as further movement of the mouse may be treated within the presentation layer itself) and generate the corresponding syntactic events, which are sent to the dialogue manager. They also react to the

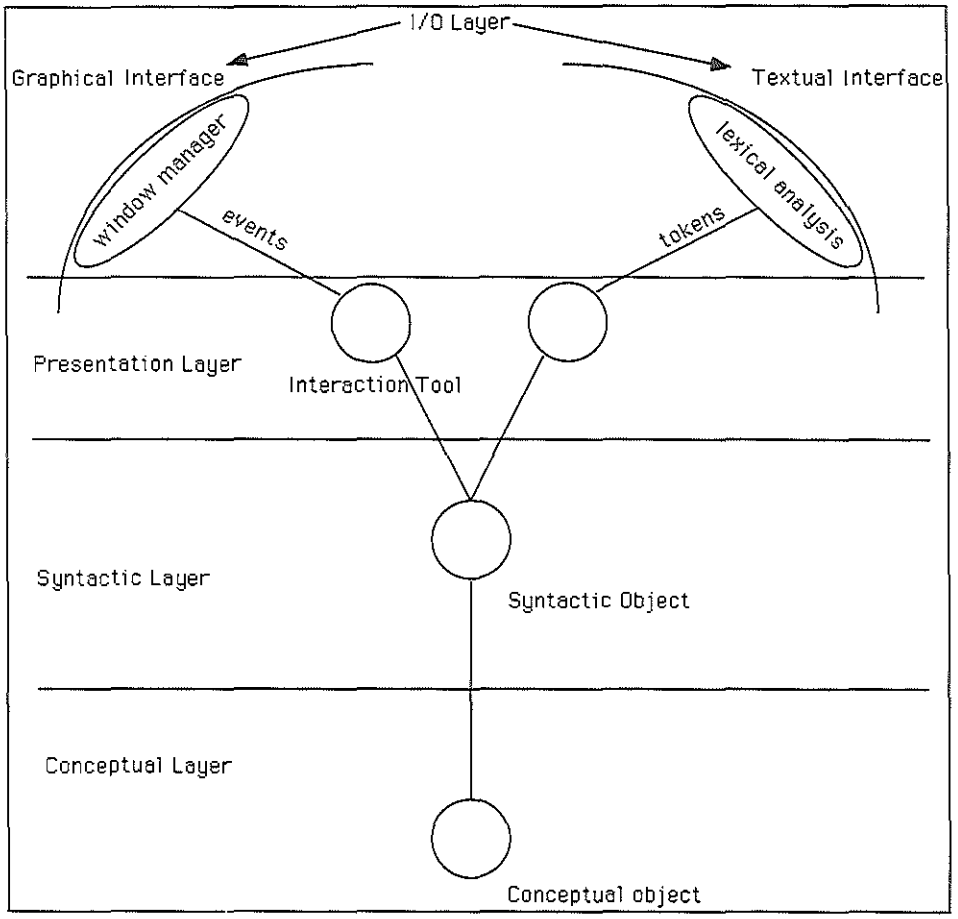

Fig. 3.1

The coupling between the I/O layer and the conceptual layer. A syntactic object represents a certain conceptual object, which may either be a data object an operation description, a workspace or a request. One syntactic object may have several presentation objects, corresponding to different ways of presenting and selecting that object. In the figure, the syntactic object may either be presented graphically and selected by mouse, or presented and selected textually. These different forms of interaction can be used interchangeably in the same dialogue, e.g. selecting a data object by mouse, and the operation by name.

messages from the dialogue manager, e.g. in order to let certain objects and properties indicate that they can accommodate the object that is being moved. These are again translated into lower level messages to the presentation objects, e.g. into a message for a presentation object to highlight itself.

The use of a separate layer of syntactical objects allows two (or more) kinds of presentation and corresponding forms of I/O which can be used interchangeably in the same dialogue, e.g. (fig. 3.1) selecting an object graphically, and the operation textually.

Definition 16 defines the different types of syntactical representatives, each different type representing a different type of object in the conceptual layer. The common subtype is iSyntacticObject_type; it prescribes that a syntactic object refers to a workspace, to the object it represents and has a boolean property indicating whether the syntactic object may be selected. The common supertype for syntactic objects iSyntConceptualObject_type representing conceptual objects (either data objects or requests) prescribes the property Propdescs, which is the set of properties to which objects (or values) may be added, removed, or values changed. The property ToPropsAssignable is used during the dialogue to indicate (and administrate) the properties to which an object that is being moved may be assigned. Other types are types of objects representing:

iSyntDataObject_type:	*data objects,*
iSyntRequest_type:	*request objects (in general),*
iSyntOperationReq_type:	*operation request objects,*
iSyntWorkspaceReq_type:	*workspace request objects,*
iSyntOperation_type:	*operation descriptions,*

3.6.2 The dialogue manager

The dialogue manager is an object which conducts the dialogue by coordinating the syntactic objects. It keeps track of the state of the dialogue: dependent on previous user activities (what events it has received from one of its syntactic objects) it determines what is appropriate for the user to do next. This incorporates both determining what kind of event it expects, and also determining precisely the contents of that event. In order to determine what kind of event is the next appropriate, the dialogue manager is implemented as a kind of finite state machine (Aho, Sethi et al. 1986). If an event does not match the expected event, and if the user wishes to cancel the previous dialogue, the state machine is reset, and the event is applied again. In order to coordinate the various syntactic objects, the dialogue manager keeps an administration of these objects, subdivided into syntactic objects representing data objects, requests,

workspaces and operations. Furthermore, it keeps an administration of syntactic objects that were requested to make themselves selectable: if a dialogue is completed, or reset, these objects must be reset too. Finally, the dialogue manager retains which object, or which property (belonging to which object) was selected previously, so that if the user selects the next expected operation or object, the complete action can be constructed and applied to the conceptual layer.

A short example is the following:
The dialogue manager starts in the state "ANYTHING", which means that anything may happen. If the user selects an object, the corresponding syntactic objects sends the event "ObjectSelected" to the dialogue manager, with the syntactic object which it concerns as a parameter. From this parameter, the dialogue manager derives the identity of the conceptual object that is involved (which is retained for further use) and it determines which syntactic objects represent operations that may be applied to that conceptual object. These syntactic objects are requested to make themselves selectable. The dialogue manager changes its state to "SELECT_OPERATION", meaning that it expects the user to select an operation. If the user subsequently selects an operation, the corresponding syntactic object sends the event "OperationSelected" to the dialogue manager, again giving its own identity as a parameter. The dialogue manager determines whether this is one of the syntactic objects that were selectable. If so, the dialogue manager invokes the action at the conceptual level to create the corresponding request, and to add the previously selected conceptual object as an operand to the request. It also creates a syntactic object to represent this request, and adds this to its administration. Next, it resets the dialogue to "ANYTHING". If an operation was selected that had not been set to be selectable, the dialogue manager asks the user whether he wants to cancel the dialogue which involves the previously selected object. If the user wants to cancel, the dialogue is reset, and the dialogue manager again sends itself the event "OperationSelected", with the same syntactic object as parameter. Now the dialogue manager reacts from state "ANYTHING", and creates the corresponding request as a result.

Definition 18 defines the structural aspects of the dialogue manager. The dialogue manager has an attribute Status, which may either be: ANYTHING, ASSIGN_OBJECT, SELECT_OPERATION, SELECT_OBJECT, giving the state of the dialogue in terms of what is expected next of the user. The dialogue manager keeps an administration of:
all data objects, in the property AllDataObjects
all operation requests, in the property AllOperationReqs
all workspaces, in the property AllWorkspaceReqs
the objects to which the object being moved may be assigned (if appropriate), in the property ToObjectsAssignable

the objects which may be selected at a certain point in the dialogue, in the property ObjectsSelectable
the operations which may be selected at a certain point in the dialogue, in the property OperationsSelectable
the object that was selected as subject for change of one of its properties (if appropriate), in the property IndirObject
the property that was selected, in the property PropDesc
the object that was selected or moved in the property DirObject.

The methods in definition 19 describe some transitions (by the post condition) of the dialogue manager.
post_Reset describes the reset of the dialogue;
post_ObjectMoved describes the reaction to the event object moved. The parameter isdo is the syntactic object representing the object that was moved. If the corresponding conceptual object is not part of the set of free objects, it is being removed. The object and property from which it is removed are found, and the remove action is invoked. Next, the dialogue manager calls itself with the same event, now in a situation where the object is part of the free objects. If the conceptual object is part of the free objects, through the method syntobjassignee (see definition 17), the dialogue manager finds the set of objects to which the moved object may be assigned; the method sets also the property ToPropsAssignable of the corresponding objects. This set of objects is made part of the administration in the property ToObjectsAssignable. The status is set to ASSIGN_OBJECT. The object that was moved is retained in the property DirObject. This is done only if the status was ANYTHING, otherwise cancel is invoked to ask the user whether he wishes to cancel; if he does, the state is reset, and the dialogue manager calls itself with the same event.

post_ObjectAssigned can only be invoked in the state ASSIGN_OBJECT. No other situation is possible. The object to which the assignment is made is represented by the syntactic object isdoindir, the identity of the object itself is in the variable ioindir. This object and the chosen property must be able to accommodate the object being assigned (which was held in the property DirObject). This is tested by the predicate pre_AddCmpinCntxt; If it can be accommodated, the assignment is called. If not the user can cancel, etc.

In the following, the default assumption is, that the user gives a correct event.

post_ObjectMigrated can also only be invoked in the state ASSIGN_OBJECT; if the migration is allowed, it effectuates the migration and leaves the dialogue in the same state, so that the user can (and must) still assign the object.

post_ObjPropSelected is invoked if the user selects an object and its property of which the value will be changed. If the object is a request, the method syntobjassignable is invoked to find out which objects can be assigned to the property (the objects are not limited to any workspace), otherwise -the object is a data object- the method syntobjrelassignable is used, which finds selectable objects within the workspace. Syntactic objects which are selectable are set to be selectable through the corresponding property.

post_ObjectSelected may be invoked in the state ANYTHING or in the state SELECT_OBJECT. If it was selected in the state ANYTHING, the next state is SELECT_OPERATION, and the selectable operations are indicated. If the state was SELECT_OBJECT there may two possibilities: The previously selected property was a component or a cross reference property. This determines the pre and post conditions to use for the property change. In both cases, there may again be two possibilities: The selected object was already in the property plus indirect object that were selected previously - in which case it must be removed - or it was not, in which case (if it can be accommodated) must be added. If the state was SELECT_OBJECT, this piece of dialogue is completed, and the dialogue manager is reset.

post_OperationSelected can either be invoked in the state ANYTHING, or in state SELECT_OPERATION. In both cases, a request must be created (if the invocation was correct). In the second case, an object was selected previously, and this is supposed to be the operand of the newly created request. In both cases, values are propagated from the workspace in which the operation was selected to the request using post_Adopt. In both cases, for the newly created request a syntactic object must be created and this must be added to the administration; this is done using the method AddReqtoSyntLayer.

post_RequestSubmitted can only be invoked in the state ANYTHING. The request must be well-formed; this is tested, and the validity of the request is set accordingly using the method post_set_cond_valid. If the request can indeed be set to valid, (conceptual) objects that will be changed (held by in_out properties) -and specifically their syntactic objects- are removed from the administration, since new objects will (or may) return in their place; a change operation may replace existing objects. The request is executed, and objects that are either newly added because they were replaced, or objects that are newly added because they are held by an out operand are added to the administration. For the changed objects this is done using the AddCompDObjtoSyntLayer, since comp indicates that only components of those objects will be newly added; these are possibly replaced. For newly created objects, this is done using AddRecDObjtoSyntLayer, which adds the object itself and its components recursively to the syntactic administration.

post_OpenRequested can be invoked in any state. It adds the context object in the operand NewInspectObject of the opened workspace and its components recursively to the adminstration. It does the same for the operations referred to (either by ContOps or CRefOps) by the module (remember: this also corresponds to creating the corresponding syntactic objects, and creating the presentation of those operations). Given the fact that new operations and objects are added to the ones that may be selected by the user, and given the fact that the dialogue may be in any state (e.g. SELECT_OBJECT) when the workspace is opened, the following must be done. If the state was, e.g., SELECT_OBJECT, the system must again find out which objects may now be selected, since new objects were added. This is done, and similarly for other possible dialogue states.

3.7 Presentation aspects

Operations or actions that change conceptual objects in the model world -and which are implemented at the conceptual level or below- do not need to take into account the possible existence of a presentation, and therefore do not need to incorporate a possible corresponding change of that presentation.

The dialogue manager is able to determine which actions lead to certain structure changes of conceptual objects. In the case of an operation application, it can infer from the operand descriptors whether an operation application leads to the change of an operand (if the sort of the operand descriptor is in_out, see chapter 2) or to the creation of a new object (if the sort of the operand descriptor is out), and thus whether it requires the change or creation of the presentation of those objects. To effectuate presentation changes, the dialogue manager commands the appropriate syntactic objects to change the presentation; the syntactic objects at their turn invoke the "Present" operation, to be applied to the conceptual object they present.

As will be described in chapter 4, part of the execution mechanism for an operation, such as the operation "Present", involves creating the appropriate representation (or format) for the operands and settings involved, so that these can be used as input by the software or software package that executes the operation. The representation to be generated is specified using the so-called implementation type. In case of the execution of the operation "Present", an implementation type can be specified that corresponds with the data structure or widget that will be generated to present the conceptual object using a certain toolkit.

3.8 Discussion, Conclusion

As outlined in the section on literature (3.3), one of the major shortcomings in the area of tools and systems for developing user interfaces lies in the development of user interfaces that provide strong semantic feedback. User Interface Management systems (UIMS's), due to their focus on specifying the user interface at the dialogue level, provide only limited possibilities in this direction. In user interface toolkits, the dialogue must be implemented by the programmer, which allows him to incorporate semantic feedback but which also (re-)introduces the complexity of this task. In the author's opinion, strong semantic feedback is (one of) the key features of direct manipulation interfaces, and allowing for the simple specification of these kind of interfaces is necessary to satisfy the increasing demand for them.

The approach to user interface modelling and functioning as described in this chapter (which builds on the description of the data model in chapter 2), is specifically directed towards giving semantic feedback. This is done by explicitly modelling the objects in the model world through types and operations, and coupling run-time available type information and operation descriptions to these objects, so that, when the user "grabs" an object appropriate structure changes, and applicable operations can directly be derived. New objects, including results of operations are also coupled in this way to type information and can therefore be manipulated in the same way.

Apart from semantic feedback, there is another important issue which is tackled by this approach, which will become of increasing importance in the future. Referring to the literature (Lu 1992) this is the strive for document oriented interfaces instead of application oriented environments. In document oriented interfaces, not the applications but the documents (or just data) are the key entities in the use of the computer. Document oriented interfaces as described by Lu require cooperation -so called "interoperability"- between various small modules that are used in combination. Each of these modules takes care of its own user interfacing. The approach advocated here is different but aims at the same effect: The assertion is, that it allows for the creation of one "generic" interface, encompassing all objects from larger documents to smaller objects down to the level of strings, numbers, graphical objects etc, and in which such interoperability between modules is not needed. This genericity may be elucidated as follows. One may view the approach as being based on the existence of only a small number of principally different actions a user employs in a (any) user interface: These consist of actions for directly changing the structure of objects and of an action for applying (arbitrary) operations to objects. The generic user interface only needs to support these actions, as guided by the run-time available type information and operation descriptions for specific types of objects and specific operations.

Of course, there are (or seem to be) certain disadvantages, because this approach is restricted by the data model that is chosen. For instance, in the YANUS data model as described in chapter 2, dependencies or constraints between (the values of) different objects cannot be modelled, while in the approach to DOI's described by Lu, each module can implement its own features which are appropriate for the resources that it provides[10]. Such restrictions will therefore also be found when considering user interface approaches that are similar to the YANUS approach, such as described by (Hudson and King 1986; Pemberton 1990). However, specific for YANUS is the use of meta types which makes the data model extensible: For instance, workspaces, which play an important role in the modularization of the use of objects and operations, were introduced in this chapter through a meta type. Basically, this is an extension of the data model already described in chapter 2. Note, however, that the introduction of a meta type is not a trivial matter. In general, this also involves adapting the dialogue manager. At least, meta types offer a framework for introducing these kind of changes. As a result of introducing a new meta type, all new specifications (which are at the type level) can use the modelling power introduced by it.

With respect to the approaches described by (Hudson and King 1986; Pemberton 1990), the following may still be said. These approaches are specifically based on the modelling of interdependencies between objects. This allows deriving output data from certain input data, using a constraint. This may be viewed to correspond in the YANUS model to the derivation of some output using an operation, but where YANUS does not maintain the dependency between output and input. In these other approaches applying an operation to some object which changes that object (in some indirect way) is not allowed. In the author's opinion, there are several reasons why such a construct is useful:

- It is a natural way of manipulating things. In daily life, effectuating compound changes is a normal thing to do. In manipulating data by a computer, it is also very useful: e.g. when effectuating large changes to a text (e.g. search and replace).
- Such a compound change may be viewed as a composition of edit actions. These other approaches do not forbid the use of edit actions; thus forbidding compound edit actions does not seem logical. Note, for example, that opening and closing of a workspace and editing the inspected data in between is modelled very naturally in this chapter as being similar to applying an operation that changes its input.

[10] However, the approach described by Lu, having bigger objects that provide interfacing for the smaller objects within, can still be incorporated in the approach described. This results in certain restrictions for the use of these smaller objects.

As already described in chapter 1, the approach of keeping a type coupled to objects allows optimal combined use of resources. When new modules are added, which reuse existing types as much as possible, or introduce new operations that are applicable to previously defined types, the corresponding resources can be combined: previously defined operations may be applied to newly added objects, as long as the type of the objects is appropriate. Newly defined operations may be applied to objects of those earlier defined types.

Summarizing, the advantages of this user interface approach are:

- easy specification of powerful (direct manipulation) user interfaces - note however, that the specification of the presentation component has not been studied;
- its genericity: It allows for specification of document oriented interfaces containing many different types of objects;
- resources presented by a user interface can be optimally combined.

Literature

Aho, A. V., R. Sethi, et al. (1986). Compilers, Principles, Techniques, and Tools. Reading, Mass, Addison-Wesley.

Akscyn, R. M. (1988). "KMS: A Distributed Hypermedia System for Managing Knowledge In Organizations." Communications of the ACM **31**(7): 820.

Backlund, B., O. Hagsand, et al. (1989). Generation of Graphic Language-Based Environments. Swedish Institute of Computer Science.

Foley, J. (1988). "Interfaces for Advanced Computing." Scientific American, Trends in Computing **1**(1): 62.

Green, M. (1985). "The University of Alberta User Interface Management System." SIGGRAPH Proceedings **19**(3): 205.

Haan, B. J., P. Kahn, et al. (1992). "Iris Hypermedia Services." Communications of the ACM **35**(1): 36.

Hartson, H. R. and D. Hix (1989). "Human-Computer Interface Development: Concepts and Systems for its Management." ACM Computing Surveys **21**(1): 5.

Hudson, S. E. and R. King (1986). "A Generator of Direct Manipulation Office Systems." ACM Transactions on Office Information Systems **4**(2): 132.

Hutchins, E. L., J. D. Hollan, et al. (1985). "Direct Manipulation Interfaces." Human--Computer Interaction **1**: 311.

Jacob, R. J. K. (1986). "A Specification Language for Direct-Manipulation User Interfaces." ACM Transactions on Graphics **5**(4): 283.

Laffra, C., J. Van Den Bos, et al. (1990). DIGIS - More than an Interface Builder. The European X Window System Conference and Exhibition (EX'90), London,

Linton, M., J. Vlissides, et al. (1989). "Composing user interfaces with interviews." IEEE Computer **22**(2): 8.

Lu, C. (1992). "Objects for End Users." BYTE **17**(14): 143.

Marchionini, G. and B. Schneiderman (1988). "Finding Facts vs. Browsing Knowledge in Hypertext Systems." IEEE Computer **21**(1): 70.

Naiman, A., N. E. Dunn, et al. (1992). The Macintosh Bible Fourth Edition. Peachpit Press.

O'Reilly and Associates (1989). X Toolkit Intrinsics Programming Manual. Sebastopol, Calif.,

Pausch, R., M. Conway, et al. (1992). "Lessons Learned from SUIT, the Simple User Interface Toolkit." ACM Transactions on Information Systems **4**(10): 320.

Pemberton, S. (1990). Open User-Interfaces: The Views System. Najaarsconferentie NLUUG, Ede, NLUUG.

Pfaff, G., Ed. (1985). User Interface Management Systems. Berlin, Springer-Verlag.

Shneiderman, B. (1983). "Direct Manipulation: A Step Beyond Programming Languages." IEEE Software : 57.

Teitelbaum, T. and T. Reps (1981). "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." Communications of the ACM **24**(9): 563.

Van Den Bos, J. (1988). "Abstract Interaction Tools: A Language for User Interface Management Systems." Computer Graphics Forum ACM Transactions on Programming Languages and Systems **10**(2): 215.

Van Den Bos, J. and C. Laffra (1990). "Project DIGIS; Building Interactive Applications by Direct Manipulation." Computer Graphics Forum **9**: 181.

Webster, B. F. The NeXT Book. Reading, Addison-Wesley Publishing Company, Inc.

Weinand, A., E. Gamma, et al. (1989). "Design and implementation of ET++, a Seamless Object-Oriented Application Framework." Structured Programming **10**(2): 9.

Wiecha, C., W. Bennett, et al. (1990). "ITS: A Tool for Rapidly Developing Interactive Applications." ACM Transactions on Information Systems **8**(3): 204.

Zeltzer, D. (1985). "Towards an integrated view of 3-D computer animation." The Visual Computer **1**(1?): 249.

Chapter 4

Executing operation requests

4.1 Introduction

As described in previous chapters, YANUS is a frontend system in which a model world is modelled through complex objects of various types. The user manipulates these objects: this can be done using direct generic edit actions, but the user may also apply operations to objects in order to effectuate changes, derive some information, or execute queries that were specifically implemented for a certain type of object.

Operations are applied by first generating a corresponding request, filling in the operands and settings (together the parameters) as described by the corresponding operation description, and by then submitting the request for execution.

Underlying resources are made available in the frontend by means of the execution of operation requests. This encompasses both the delegation of processing tasks, thus the coupling to operation resources, and the coupling to data (storage) resources: By executing a request, data may be read in from an arbitrary storage system into the internal object representation of the model world of YANUS, or (changes to) the data may be saved to the storage system (see chapter 2: opening and closing workspaces). This chapter is concerned with the mechanism for executing requests, as a.o. implemented by the so-called software package interface (see chapter 1, figure 1.1).

The execution of a request is subdivided into several subtasks:

1. As described in chapter 2, the type of the generated request for an operation is always of the highest supertype of that operation, i.e., having the most general description of the types of the operands. For this operation several subtypes may be defined corresponding with more specific types of the operands. The subtype and therefore, (eventually) the implementation, appropriate for the current types of the actual operands must be found. Note, that in this process only operands, not settings are taken into account.

2. For the chosen subtype of the operation, an implementation in an underlying software package or in underlying software must be found. For such an implementation, transformations must be performed to copy the data represented by the conceptual operands (i.e. the operands of the original request) and settings to a representation which can be used as input. For instance, this involves copying and transforming data from a stored representation in a DBMS such as Ingres, to an input file for a software package such as Ispahan. Thus, a conceptual object may have several representations at any time. This means that in the chosen approach the conceptual type of an object and its representation/implementation are decoupled.
3. Assuming that a software **package** is driven (as distinct from software with a procedural interface, which is a more simple case), its *state* may have to be managed: The package may need to be brought from its current state to a state in which it can execute the implementation operation. Since the session with the package will in general continue after the completion of the execution of the request, the resulting state must be determined and stored to enable the execution of the next operation.
4. The software package must be driven by means of a simulated key sequence or by a batch job program, written in some formal language (e.g. SQL statements). This sequence or program must be generated.
5. The output of the package must be processed. This may include recognizing transitions in the state of the package, as well as recognizing the result of the operation and transforming this output to objects in the model world.

As required in chapter 2, subtask 1 should specifically allow the execution of operations with any number of operands (including none), all operands together determining what software is finally executed.

Subtask 2 is called implementation management. It may also be described as follows: The request execution must be such, that the set of operations applicable to a conceptual object of a certain type is the union of what operations may be applied to these same data in the various software packages.

The combination of subtasks 3, 4 and a part of 5 is called software package management. (Returning the output to the level of conceptual objects is part of the implementation management.)

This research is directed at finding general solutions i.e., principles and mechanisms which are independent of the operation to be executed, the package to be used, etc. This renders the system flexible and extensible with respect to the functionality to be integrated. Note, however, that the system was designed with a particular example in mind (its direct

application area): the integration of a DBMS called Ingres and a software package for interactive pattern recognition called ISPAHAN. This means that when integrating other software packages, specific problems, other than the ones stated above, may still be encountered.

Flexibility and extensibility is also required with respect to the data to be integrated. It should be possible to make an ad-hoc connection to some arbitrary database and tap its contents. Only the type of DBMS should be known, or otherwise it should use the (standard) query language SQL. This allows the full integrated functionality of the integrated system to be applied to these new data.

The remainder of this chapter is subdivided as follows: In section 4.2, restrictions that we impose on our problem space are discussed: for example, driving software packages having a purely graphical user interface is outside the scope of this research. In section 4.3, some approaches to request execution as described in the literature are discussed. In section 4.4, an overview of the system that handles the problems stated above is given. Section 4.5 focuses on the search for an appropriate operation subtype. In section 4.6, implementation management is discussed. This not only encompasses the choice of an implementation for an operation and the (external) representation to be generated, but also the creation of an internal representation (i.e. objects) of the data, keeping all representations up to date, etc. In section 4.7, the problem of software package management is discussed. This encompasses state management, creating the input for the package and parsing its output. In section 4.8, a discussion and conclusion is given.

In certain sections, a description of the formalization is given. This is done in *italics*.

4.2 Restriction of the scope

The scope as addressed in this research is restricted by leaving out the following features:

- A programming interface. Implementing this would entail having to map complex composite YANUS operations onto operations in underlying packages.
- Allowing complex conceptual objects to be defined on top of flat files in the underlying storage. E.g. complex objects that are constructed by joining relations in an underlying relational database. By omitting this feature, a.o., problems in updating these kind of structures do not have to be solved. More in general, the structure of the conceptual objects will directly reflect the structure of the external data.

- Allowing the creation of objects which correspond to combinations of data from different databases and DBMS's. This problem is treated by others (see section 4.3). In principle, such other integrated systems may be included in the integration.
- Allowing the integration of data that may be changed beyond the control of YANUS, e.g. data in a DBMS which is shared and updated by others while being in use within YANUS.
- Allowing the integration of software packages that can only be driven via their standard user interface by means of mouse clicks.
- Allowing concurrency: i.e., allowing the user to issue new requests, while the execution of a previous request is still in progress.

Moreover, certain restrictions in the search for an appropriate operation subtype are incorporated. This will be detailed in section 4.5. Error handling and recovery are not modelled as yet.

4.3 Literature

In this chapter, the focus is on general mechanisms and principles which may be used to make external systems execute operations. Some generally used execution mechanisms will therefore be reviewed in this section. Furthermore, systems such as YANUS, which integrate other functionality in addition to the functionality of DBMS's will be reviewed.

Integration of DBMS's is an important topic in computer science. It is achieved by "delegating" a query defined over a global integrated schema to local queries in the underlying DBMS's. An important concept which is often used is that of the "view": the global schema is a view on the underlying local schemas. A query on the global view is translated to queries on the local views and to operations which combine the results of the local queries. In principle a query ranges over the complete data set. Querying, or applying an operation to a specific, designated object is not possible. Thus, this technique is in contradiction with the way in which objects and operations should be usable within YANUS. One of the first systems to achieve an integration based on this concept was Multibase (Landers and Rosenberg 1982). Much research and development is still being done in this area (Wiederhold 1986; Desai and Pollock 1990).

However, for the development of YANUS, techniques for executing operations as applied to specific objects are of interest. Objects as well as operations must hide their implementation from the user. This is called data- and procedure abstraction, respectively.

The use of data abstraction in information system integration has a.o. been described by Bertino (Bertino, Negri et al. 1989) and Eliassen (Eliassen and Veijalainen 1988). Their execution technique is one of polymorphism based on "direct" data abstraction: Objects corresponding to certain underlying data have an interface of operations that drive the underlying software package and read in the result. Two objects may have an interface of similarly named operations and properties and may thus be used similarly, while their internal status and the operations are implemented differently (i.e. polymorphism). Furthermore, which implementation is used is determined at execution time. This is called "dynamic binding". Polymorphism in combination with dynamic binding is a prerequisite for implementation management in YANUS. Eliassen shows that a functional approach to data abstraction (i.e. the operation is central in the execution) allows an extra indirection: a choice can be made which "server" executes the operation. Practically, this may imply choosing a specific software package to execute the operation.

In these approaches objects are still intrinsically coupled to one implementation. Given the need for copying and generating new representations/implementations for one conceptual object as described in section 4.1, these approaches therefore still do not provide a sufficient decoupling between conceptual type and implementation.

Abelson & Sussman (Abelson, Sussman et al. 1991) describe a more indirect form of data abstraction. The operation implementation to be used is determined in a multi-staged way, e.g. the execution of an operation "addition" applied to numbers depends at the first stage on whether the numbers are real or complex (thus, in the terminology of section 4.1, on the type of the operands), and at the second stage on the representation. This multi-staged execution mechanism provides a stronger encapsulation of the representation of an object; it allows varying the representation for one object. Others have also described multi-staged operation execution (Decouchant 1986; McCullough 1987; Purdy, Schuchardt et al. 1987). Still, these approaches do not incorporate a mechanism which allows several representations at a time for one object. A fortiori, none of these approaches encompasses the creation of a new representation when necessary, to allow for the execution of a certain operation in a specific software package.

An important approach in this area is the one described by the Object Management Group OMG (OMG 1992). This approach enables requests for operation execution on objects to be delegated to external systems. The OMG approach strives for a strong decoupling between conceptual type and implementation. For external systems an object-oriented interface may be implemented by encapsulating the underlying non-object-oriented system. Copying of representations between external systems does not seem to be provided for.

In many (programming) environments, the execution mechanism for executing a procedure invocation or message is fixed. Maes (Maes 1987), however, describes how the programmer may be enabled to program his own execution mechanism. A program that takes care of the execution of another program is involved in "meta-computation". The use of this principle, also for the purpose of DBMS integration, is applied extensively in the VODAK datamodel (Schrefl and Neuhold 1988). No mechanism which specifically solves the problems described in this chapter is described. However, it seems natural to implement the execution mechanism for this chapter using a similar approach.

Other systems that integrate existing packages in addition to DBMS's (particularly packages for data analysis) are Rochefort (Hilhorst, Van Romunde et al.) and MW2000 (Van Mulligen, Timmers et al. 1991). Such integration systems specifically require the generation of additional representations in order to enable the analysis of the data. Rochefort seems to have a solution for this problem which is specific for the set of operations that it offers to the user (This is not certain, since its execution mechanism is not revealed). Neither approach uses data abstraction in the sense that operations can directly be applied to specific data objects. MW2000 uses an approach which is closer to that of maintaining a view: There is one key data set with a standard internal representation to which the user applies operations. Dependent on the request, the internal representation is translated to the external representation needed for executing the request, or for starting the module in which the user can apply a specific kind of analysis to the data. MW2000 does encapsulate the execution mechanism for executing certain requests by means of classes. Both Rochefort and MW2000 are mainly directed at driving underlying packages in a batch way. After the execution of each request, the session is terminated, which includes halting the running process of the package. Thus, state management is not needed.

4.4 The layer model

The complete request execution mechanism described in this chapter may be illustrated by a layer model, shown in figure 4.1. This model describes the communication with the underlying software package at different levels of abstraction. It reflects the division in subtasks described in section 4.1. In the rest of this chapter, these different levels will be described.

4.5 Finding the appropriate operation subtype

As described in chapter 2, operations are ordered in subtype/supertype hierarchies: An operation A is a supertype of an operation A', if one or more operand descriptors of operation

A` prescribe more specific types (that is subtypes) than corresponding operand descriptors of operation A. This means, that in the execution of A` a more specific implementation corresponding to more specific representations of the operands is used. However, principally, operation A may still be used instead of A'.

As an example, there may be different representations for complex numbers and integers and corresponding implementations for the addition of complex numbers and the addition of integers. In this hierarchy, addition of complex numbers is a supertype of addition of integer numbers. This corresponds with the fact that integer may be viewed as a subtype of complex number, since each integer may be written as a complex number. See for a detailed explanation (Abelson, Sussman et al.)[1].

Operations A, A', A'' in a subtype/supertype hierarchy have the same name, and the same number of operand descriptors with and same names. There is one *highest supertype* in such a hierarchy. In the example of addition mentioned above this is the addition defined for complex numbers.

Each type for which a certain operation is defined refers in its property "Operations" to the highest supertype of that operation; this is the operation that is instantiated to create a request for that operation. If, for a certain type Tx, a subtype operation has been specifically defined (such as the subtype of addition for integers) Tx refers in its property "OwnOperations" to that corresponding operation description. That operation subtype is said to be *directly applicable* to type Tx.

When the user submits a request for execution, the most specific operation subtype that fits best with the chosen (actual) operands must be found. Note, that the settings are not considered in this process. The corresponding implementation of the operation will be used for the execution of the request.

The search mechanism employed here for finding the operation subtype is relatively simple and does not necessarily find the optimal subtype operation. In this mechanism, the first operand is used as the most important one: It selects the operation subtype of which the required operand type fits best with the type of the object in the first operand, and which is also applicable to all other operands.

[1] The difference between this approach and that described by Abelson and Sussman is, that here a subtype/ supertype hierarchy of operations is used; Abelson and Sussman only use a subtype/ supertype hierarchy of the operands/ parameters.

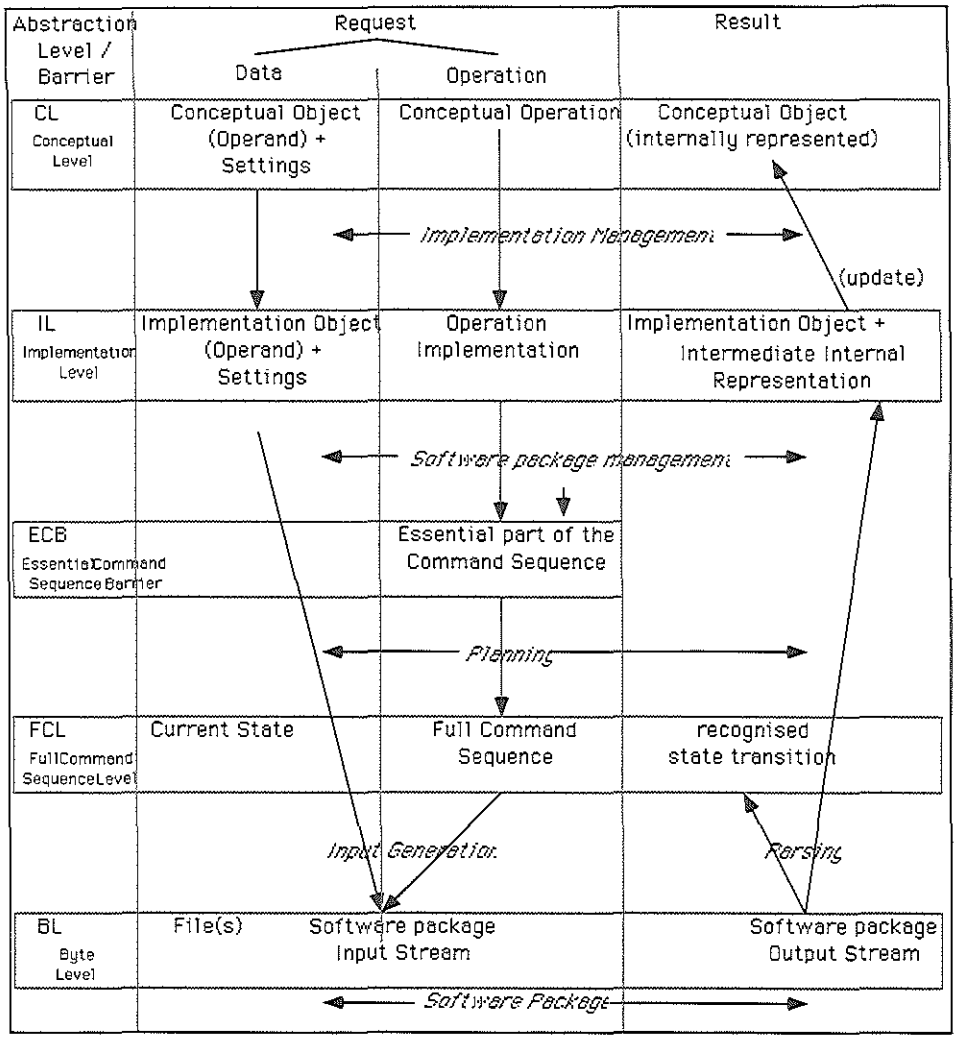

Fig. 4.1

(Caption see next page →)

Where:

- The required operand type fits best with the type of the object if it is that type or a close (as close as possible) supertype.
- An operation is defined to be applicable to an operand if it prescribes an operand type equal to the type of the operand or an arbitrary supertype thereof.

For instance, assuming that operations have been defined to add integers, reals, and complex numbers, and the user wants to add an integer and a real, the first operand being an integer. The operation for addition of integers is directly applicable to integer, but cannot be applied to the second operand. Going up in the hierarchy, the next best operation is found which is the one for adding reals. This operation is also applicable to the second operand. Before using an operation for adding reals, a transformation is necessary for the first operand (called a type coercion by Abelson & Sussman). This is incorporated in the transformation mechanism as described in section 4.6.2 and 4.6.4.

Finally, referring to figure 4.1, The conceptual request with its (actual) operands and settings form the highest so-called *Conceptual* level of abstraction at which communication with the underlying software package takes place.

First, some of the most important aspects of the coupling between the types and operation descriptions which have also been mentioned in the text are recapitulated from chapter 2 in the formalization. Determining the operation subtype of which the implementation will be used is formalized in definition 1 by the predicate best_applicable. This predicate determines

---

Fig. 4.1
Overview of the abstraction levels and barriers used in request execution.
Abbreviations:

CL	Conceptual Level
IL	Implementation Level
FCL	Full Command sequence Level
BL	Byte Level.
ECB	Essential Command sequence Barrier

At each level may be distinguished:

- A request, identifying "what must be done" (an operation or command) and data.
- A result.

At an abstraction barrier no data are identified. In executing transformation operation implementations the same abstraction levels are used.

whether a certain operation satisfies the requirement that it is directly applicable to the type in the type hierarchy closest to the first operand type (the identity of that type is held by the variable frstoprndtype) while being applicable to all other operands. This latter condition is formalized by the predicate appl_toothers. Note that if the set of property descriptors of the request is empty, the chosen operation is directly the type of the request. Other variables in the predicate have the following meaning: itod is the highest operation supertype; ifrstoprnddesc is the operand descriptor for the first operand; maxtype is the most general first operand type, as prescribed by the element type of ifrstoprnddesc. iod is the chosen operation, which is directly applicable to the type iot.

4.6 Implementation management

4.6.1 Schema and objects at the implementation level and their relationship with schema and objects at the conceptual level

The next level of communication with the underlying software package or with underlying software is the so-called *implementation level*, see fig. 4.1. In analogy with the conceptual level, at the implementation level a schema is used to describe (still at a rather high abstraction level) the interfaces of the underlying software (see fig 4.2): At this level, so-called implementation operations can be invoked. An implementation operation represents an operation provided by an underlying software package or otherwise by software, i.e. by some procedure or method implemented in some programming language. An implementation operation description describes the operands of an operation at this level. The required types of the operands (as given by the operand descriptors) must be implementation types, signifying that the corresponding operation can be applied to operands in that representation. Thus, implementation types describe possible data representations/implementations. Special transformation operations can be invoked at this level to create one representation from another.

A conceptual operation is generally executed by executing one or more implementation operations at the implementation level and generating the corresponding data representations. How this must be done is indicated by the relationships between objects, types and operations of the two levels.

A conceptual operation is linked with a sequence of implementation operations, indicating that the conceptual operation can be executed by executing that sequence. Thus, the relationship between a conceptual operation and an implementation operation signifies that the conceptual operation is (to a certain extent) implemented by the implementation operation. Such an

implementation operation has operands with similar names, or a subset of those, as the conceptual operation to which it is coupled. A similar named operand indicates that a specific representation of the same conceptual operand is used as input for the implementation operation. The relationship between a conceptual type and certain implementation types signifies that conceptual objects of that type may be represented in the ways corresponding to those implementation types. An operand descriptor of a certain name of the implementation operation may only prescribe an element type that is one of the implementation types of the conceptual type prescribed by the same named corresponding operand of the conceptual operation. Finally, whether a conceptual object is represented in a certain way, as corresponding to a certain implementation type, is indicated by a relationship between that object and an implementation object which has (i.e. refers to) that implementation type. The implementation object represents that data representation for that conceptual object. It also contains the address or reference to those data in that representation.

Transformation operations, although they are invoked at this level, are not considered to be part of the implementation of a conceptual operation and are thus not linked to the conceptual operation. Instead, all transformation operations that transform between the different implementation types of a certain conceptual type can be found using the so-called transformation table of that conceptual type. In this table, each possibly compound transformation is defined by the sequence of transformation operations which must be invoked to effectuate it. Allowing compound transformations is useful, as shall be seen later. By laying down the possible compositions in a table, an expensive search mechanism to compose the transformation at run-time can be avoided.

Definition 2 extends types used at the kernel of the system (as introduced in chapter 2) to incorporate the implementation level. The property ImplObjects is described as an additional property of all conceptual objects. This property contains all implementation objects of the conceptual object. These are considered to be owned by the conceptual object. The element type prescribed for the property is Object_type, indicating that in principle any object may serve as an implementation object. Each Operation description is prescribed to have the property ImplOperations, which refers the corresponding implementation operations. Each object type is prescribed to have the property ImplTypes, which refers to the corresponding implementation types.
Definition 3 introduces a special kind of operation (descriptions), i.e. those corresponding to transformation operations. These operations have one input operand of the name Input, and one output operand of the name output. Next, the transformtable is introduced. Transformtable is defined as a value type, using the VDM mapping construction. It maps from an input type and an output type (a compound transformation) to a corresponding set of transformation operations. A requirements is, (req_{017}) that the sequence of transformation operations is

continuous, in the sense that the next transformation operation must have an input type equal to the output type of the previous one.

4.6.2 Outline of the execution of a conceptual request using the implementation level

To execute a conceptual request, given that the operation subtype has been found (section 4.5), in outline the following is done: The linked operation implementations are invoked sequentially. For each operation implementation, for each operand, the appropriate data representation of the corresponding conceptual operand is created. Similar as at the conceptual level, for executing a specific implementation operation a corresponding implementation request is created which is to hold the corresponding implementation objects. If a conceptual object which serves as an operand of the conceptual request already owns an implementation object of the appropriate implementation type, that implementation object can directly be used. Otherwise, an implementation object of the appropriate type must be created. To do so, the shortest sequence of transformation operations is executed (which is found using the transformation table), such that from one of the existing implementation objects the implementation object of the appropriate type is generated. This new implementation object is added to the set of implementation objects (and thus representations) owned by the conceptual object and assigned to the operand of the implementation request. Finally, the complete implementation request is submitted for execution. This process is, even more schematically, illustrated in fig. 4.2. It is implemented by a generic method, called "Execute", and underlying methods and functions. See section 4.6.7.

Results of the implementation request submittal are again implementation objects as held by the corresponding out (or in_out, see next section) operands. These are again assigned as implementation objects to the corresponding conceptual objects that serve as result in the conceptual request.

An essential property of this mechanism is that it is "lazy", i.e., a representation is created only when it is needed as input for the execution of an operation implementation.

4.6.3 Implementation management for operations which change objects

A conceptual operation which changes an operand, as indicated by the operand descriptor being of the sort in_out, is implemented by (a) corresponding operation implementation(s) that change an implementation object corresponding to that same operand. The question is here, what the consequences are for other representations of the same conceptual object as they are

no longer up to date. One of the possibilities is to discard these other representations. In general it is better to (try to) update them. E.g., an update should be used for data stored in an external database, or for presentation objects (see chapter 3) on the screen if another representation has changed.

This is done by extending the lazy transformation mechanism of data representations: When a changing-operation is executed on a certain implementation object, all other implementation objects are marked as being no longer up to date. When such an invalidated implementation object is required as an operand for the execution of another implementation operation, it (and thus its underlying representation) will be updated by the transformation operation. Note that this means, that certain transformation operations (dependent on the goal representation, such as a database representation) must always update the underlying representation when available. It also means, that updates are not done immediately. For example, in the case of a database representation, the "Save" operation (chapter 3) must indeed be invoked; it **indirectly** invokes an update by being coupled to a dummy implementation operation that requires the stored database representation as input. Similarly, syntactic objects corresponding to changed conceptual objects must (as commanded by the dialogue manager) apply the "Present" operation to the conceptual object and in that way invoke an update of the corresponding presentation objects.

Definition 4 extends the schema to allow for updates on objects: the boolean attribute Uptodate is introduced which is used to indicate whether a certain representation (as represented by a certain object) is considered to be up to date.

4.6.4 The use of the internal representation

While in some view-oriented integration approaches, as in Multibase (Landers and Rosenberg 1982), data are not explicitly represented in the integrating system itself, in YANUS objects are internally represented and this internal representation is used in various ways. One purpose is interaction with the user (see chapter 3). The use of internally represented conceptual objects which is of relevance in this chapter can be subdivided into two parts:

- Use as objects to which conceptual operations are applied and which represents all underlying representations. This is the view of conceptual objects at the conceptual level.
- Use as yet another representation/implementation of the data. The internal representation may serve as input for internally implemented software, or as an intermediary in the transformation process between external representations. This is the view of conceptual objects at the implementation level.

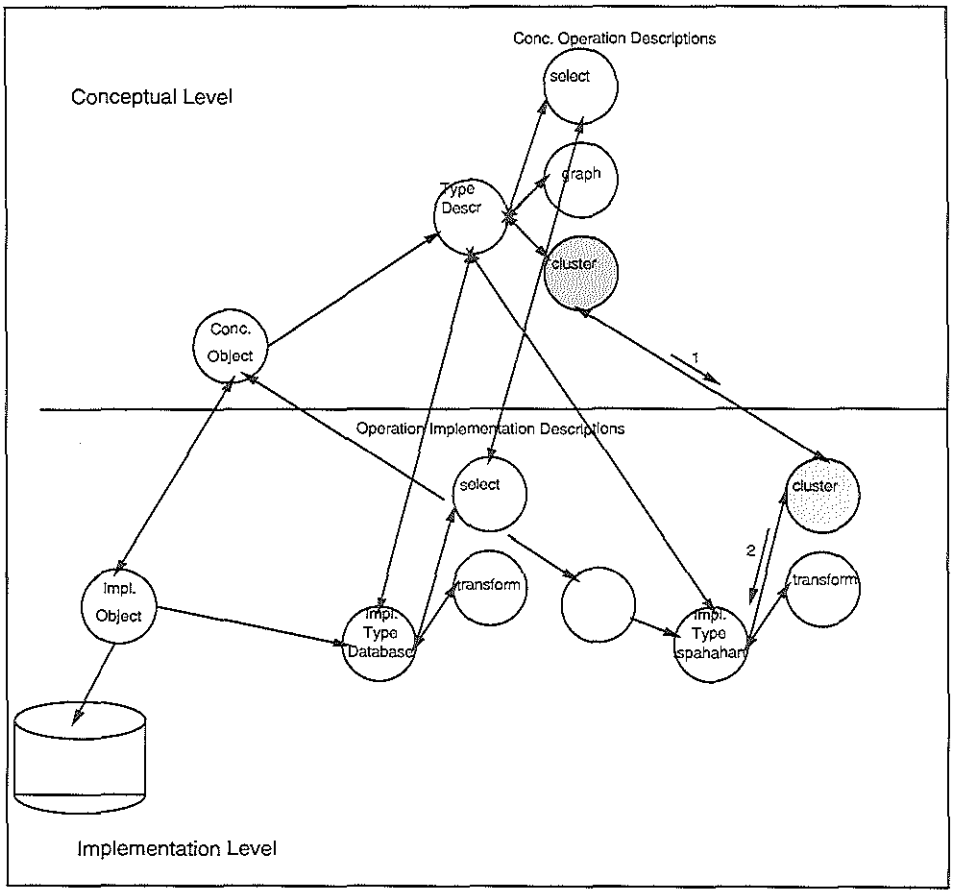

Fig. 4.2

Example of a part of the conceptual- and corresponding implementation schema, and a conceptual and implementation object. For both levels the double link between types and operation descriptions, and the link between an object and its type is given.

(Caption continued on next page→)

Conceptual objects are used as yet another implementation as follows:

1. An implementation operation prescribes an operand as being of the type of the conceptual object. In that case, the conceptual operand itself is used as implementation/representation for the input. This will be the case if the operation implementation is implemented internally. Thus, at the implementation level, the conceptual type is viewed as yet another implementation type which corresponds to the internal representation.
2. Similarly, the internal representation will be used as input or intermediary representation for a transformation if the transformation operation prescribes the corresponding conceptual type as the type of its input or output. It should be noted, that this role as intermediate representation is an important one: If there are n input representations and m output representations, using an intermediate representation n + m transformations need to be implemented, rather than n * m if an intermediate representation is not used. In this context, it is also of relevance that the internal representation is generic: A new conceptual type definition specifies a corresponding internal representation. In chapter 2, the structures that may be defined are described.

From what is described above, it appears that the use of conceptual objects (= internal representation) at the implementation level is similar to the use of other implementations.

(Caption fig 4.2 continued)
The link between a conceptual type and an implementation type represents that a conceptual object of that conceptual type may have an implementation object of that implementation type. An implementation object represents a certain data representation in an underlying software package. It contains the address of those data. One conceptual object may have several implementation objects of different types, as corresponding to different copies in different representations of the same data. In the figure, the conceptual object has two implementation objects, one corresponding to storage in a database, one corresponding to a file which is used by Ispahan.

The link between the conceptual operation and an implementation operation represents that the conceptual operation is (at least partly) implemented by that implementation operation. Thus, via this link and the link between the implementation operation and the required type of its operand (arrows 1 and 2) the implementation type can be inferred that corresponds with the representation required for executing the implementation operation. If an implementation object of that type does not exist for the conceptual object, it is created, which implies creating the corresponding representation. In this case, the representation for Ispahan is needed, and created, as indicated by the dotted circle.

However, there are some differences:

- It is of importance (specifically for the purpose of interaction with the user) to keep the internal representation up to date all the time, i.e., after each change of an underlying (other) representation. Thus, updating of the internal representation is not done lazily.
- In contrast to other implementations, transformations may be necessary and allowed between a conceptual object as viewed at the implementation level of a subtype and a conceptual object of a supertype. This will be explained further in section 4.6.5.

For these reasons, conceptual types in the view of being yet another set of implementation types at the implementation level will still be distinguished from "other", external, implementation types: These other implementation types are instances of a special meta type "ImplObjectType_type".

Updating of the internal representation must be done in such a way that objects are preserved when possible, so that cross references to these objects may remain valid. If objects must be deleted this must be done in the way described in chapter 2, so that cross references to these objects can be removed.

In definition 5 the special meta type ImplObjectType_type is introduced, which is the meta type of all implementation types that represent external representations. Types of conceptual objects, representing internal representations in this context are not an instance of this meta type.

4.6.5 Sub/supertyping and implementation types

Implementation types may have subtype/supertype relationships. For example, an ascii file having a bracket structure (e.g. a textual description of the conceptual schema) "is_a" ascii file. As a result, all (implementation) operations applicable to an ascii file (e.g searching for some pattern) are also applicable to the subtypes, in this case structured ascii files. An implementation type is not the subtype of another implementation type, if a transformation is needed to create one representation, corresponding to one implementation type, from the other.

In this respect, conceptual types, in the view of being a kind of implementation type at the implementation level are treated differently from types of other representations: A conceptual object corresponding to a supertype may have another more general representation than an object corresponding to a subtype, e.g. an object of the subtype may have an attribute of the

value type integer, while the supertype has a similar attribute of the value type real. In that case, a transformation operation is supposed to be implemented and defined in the transformation table (cf. the "raise" type coercion as described by Abelson & Sussman).

As a result, in the application of implementation operations, an object of the subtype may always be used when an object of the supertype is required. In the case of the internal representation (conceptual types) this is only so if no transformation has been defined (which means that no transformation is necessary) between the subtype and the supertype. A result is also, that transformations creating a subtype may be employed when an object of the supertype is required again with the same restriction for conceptual types.

4.6.6 Remarks about the completeness of the transformation table; inheritance of transformations and implementation types

In the usage of a transformation table coupled to a certain conceptual type (viewed at the conceptual level), it is assumed and required that if compound transformations
Tx -> Ty
and
Ty -> Tz exist,
the compound transformation
Tx -> Tz must also exist.
Thus, the transformation table describes compound transformations transitively. In this way, a certain compound transformation can always be found in the transformation table if it exists, and transformation routes need never be composed at run-time.

Given that

1. a transformation from the internal representation of the conceptual subtype to the internal representation of the conceptual supertype (i.e. a raise type coercion) must either always be defined or is not needed at all,

and using

2. all transformations between all external implementation types and the internal representation of the subtype,
3. all transformations from the internal representation of the supertype to all external representations of the supertype,

a result of this transitivity is the following: All transformations will also be defined from all external representations of the subtype to all external representations of the supertype. Therefore, all external representations of the conceptual supertype (in the view of the conceptual level) may also be used as a representation for a conceptual object of the subtype. The implementation types of the conceptual supertype (viewed at the conceptual level) are therefore inherited by the conceptual subtype. Moreover, the transformation table of the conceptual supertype is completely contained in the transformation table of the conceptual subtype.

Definition 6 introduces the extra requirements which describe (or are a result of) the transitivity of the transformation table: req_{ot8} prescribes that all implementation types of the conceptual supertype must be inherited by the conceptual subtype. req_{ot9} prescribes that the transformation table of a conceptual supertype is contained by the transformation table of the conceptual subtype; req_{ot10} states that the transformation table describes transformation sequences transitively, see the text

4.6.7 Further description of the formalization

Definition 7 describes the transformation process, to create for a certain conceptual object an implementation object of a certain required implementation type. The variable allimplobjects is made to contain the initial set of implementation objects. The conceptual object itself is included in this, and is thus in this context viewed as an implementation object. It is first tested, whether an implementation object already exists which has the required type or a subtype of the required type. This object must be up to date. Next, the optimal compound transformation must be determined by the function optimaltransform. Finally, the complete sequence of transformation functions is executed using the function transform route.

Finding the optimal compound transformation is described by the function optimal transform. First, it finds those compound transformations in the transformation table which produce the required result type, or an appropriate subtype thereof using the function findovrltrnsfrms_-onoutputtype. When the required result type is a conceptual type, representing the internal representation, a subtype is appropriate if no transformation exists between that type and a type higher in the hierarchy which is still a subtype of the required result type. This is coded by the function outputtypecompatible. Next, a set of appropriate input implementation objects (as part of allimplobjects) is formed and to that set, corresponding compound transformations are linked, such that these compound transformations are part of the set selected earlier (on producing an appropriate output type) but, in addition, can use the input object to which it is linked as input for the transformation. The link between input objects and compound

transformations is modelled through a mapping as defined by the type BEGSTRANSFORMS. Finally, using the function findshortest_overltrnsfrm, from this mapping between input objects and compound transforms, that input object and the compound transform with the shortest sequence of transformation operations is selected.

The function transformroute describes recursively the execution of the whole sequence: it selects the ith transformation operation from a transformation sequence and executes it, and calls itself to execute the i+1st transformation operation.

Definition 8 describes the complete execution mechanism. Firstly, it introduces the function postcond which defines the relation between the request before the execution and the request after the execution. This function is not further specified so that it can be specified for different types of initial and result requests. post_Execute describes specifically how a request is executed: first, the subtype operation ibestod is found using the function best_applicable (as described in section 4.5). For all operand descriptors describing output (sort is out) of the most general supertype operation (note that this supertype operation certainly describes all operands, see chapter 2), the corresponding output objects are already created. In general, the strategy is to create objects within the execution mechanism, and not to let them be created by the underlying implementation of the operation. This strategy is also used in the transformroute function described earlier. The heart of post_Execute is to call execute_all_implops, which executes the sequence of implementation operations coupled to the operation subtype ibestod.

execute_all_implops, similarly as transformroute, recursively iterates through the sequence of implementation operations. In its body, it executes one implementation operation. This is described by stating subsequent changes (ireq1, ireq2, ...) of the conceptual request and all its operands: In the first step, implementation objects are added to the conceptual operands which are needed as input for the execution of the current implementation operation. These are created using post_Transform. These implementation objects are also assigned to the corresponding operands of the implementation request. Next, the setting values are propagated from the conceptual request to the implementation request. In the following step, output implementation objects are already created (similar as in post_Execute as described above), assigned to the conceptual object that serves as that same output operand in the conceptual request, and assigned to the output operand of the implementation request. All implementation objects of conceptual objects that serve as output or are changed are set to be out of date. Next, the implementation request is (symbolically) executed using postcond. In the following step, the output implementation objects, or changed implementation objects of the resulting implementation request are asserted to be up to date. Due to having the same identity as these same objects before the execution of the implementation request, these objects are already

supposed to be part of the set of implementation objects of the corresponding conceptual object. This is asserted. In the last step, for all output operands or changed operands of the implementation operation, the internal representation of the corresponding conceptual object is updated. This is done by requesting a transformation which has the type of the conceptual object as a result. When all implementation operations have been executed, the request is conditionally set to valid.

Remaining problems are the following:

- *In this set up, operations may freely be applied to stub objects, while these operations may expect the object to be in an opened state. Solution, which has not yet been worked out is, to introduce a special subtype of Request_type of all operations which cannot be applied to stubs. This constraint can be enforced by an extra requirement.*
- *Execute does not seem to be applicable to a workspace. Again, a solution seems to be to define a special subtype of requests on which Execute is defined.*

4.7 Software package management

Execution of an implementation request is managed by the so-called software package manager. The software package manager incorporates the levels below the implementation level, see fig. 4.1.

A software package manager is an object; its state reflects the state of the software package and of the communication with the package. For certain software, specifically internally for implemented software, maintaining such a state is not needed. In that case the software package manager is trivial. Which software package manager must be used for the execution of a certain implementation operation is indicated by the implementation operation.

Major parts of the software package manager have been described and implemented by H.W.B.F. Kuil (Kuil 1993).

4.7.1 Commands - Input generation - Output parsing

An implementation operation is executed using the commands coupled to that operation. A command corresponds to a function in the host programming language (see chapter 6) which can be called under control of the software package manager. It has access to the implementation request, and thus to the operands.

This function can be used in two ways:

1. As corresponding to an executable unit in the underlying package. The function is used to generate the input of the underlying software package;
2. As a direct implementation of the implementation operation;

In case 1, the command will be coupled to a grammar which describes the output of the software package, or, if the package does not directly produce a result, a second command coupled to a grammar will be used to retrieve it. The grammar is used by a parser to parse the output of the package. Functions which incorporate recognized pieces of information into objects are attached to the grammar. These objects are not directly used at the conceptual level, but as an extra implementation for the conceptual object that represents the result. A separate operation is used to update this conceptual object; see also section 4.6.4.

In case 2, no grammar is coupled to the command; the function itself writes its result to the corresponding operand of the implementation request. Case 2 may also be used to control an underlying software package, so that the standard output parsing of case 1 is bypassed. In this case, the function both controls the creation of the input for the software package, and parses the results.

4.7.2 Commands and the state transition diagram

Certain packages, specifically ISPAHAN in the direct application area, have states in which only certain commands can be executed. In that case, specific commands are used to force state transitions. Thus, a difference may be made between essential commands in which some processing takes place, and transition commands. An implementation operation only prescribes the essential commands. This sequence corresponds to the Essential Command sequence Barrier (ECB).

In the following the assumption will be made that the state of the underlying package does not restrict the operations that can possibly be applied to an object, other than given by the schema. This means that the package can always be forced to the appropriate state to execute a command. Forcing the appropriate state transitions is called planning (see fig. 4.1).[2]

[2] In chapter 5, examples will be encountered in the direct application area where restrictions are imposed by the states of the underlying package. Such a restriction can be modelled to the user by introducing appropriate workspaces.

To allow for planning, the following properties are introduced:

- For each (essential) command: the state in which it must be executed, the possible resulting state transitions. For each possible state transition: a grammar rule to recognize this transition from the output of the software package.
- The software package manager has a table describing how a compound state transition may be forced using separate state transition commands.

Using these properties, transition commands are inserted in the command sequence in the same way as transformation operations are inserted in the implementation operation sequence, see section 4.6. The essential commands together with the appropriate transition commands is called the full command sequence (see fig. 4.1).

In the implementation, the simplification has been made to assume that the transition network has a root state, and to let the package always return to this root state after execution of an essential command. As a result planning is only needed to make the transitions from the root node to the node in which the essential command can be executed.

4.8 Discussion, Conclusion

Request execution as described in this chapter builds on commonly used binding (also called dispatching) mechanisms in object-oriented systems (Zdonik and Maier 1990), i.e., the coupling of a request to an implementation. Similar to the binding mechanism as described by Abelson & Susmann, a strict separation is made between two stages of binding: the first is based on the conceptual type of the operands, the second on the implementation. To the best of the author's knowledge, a new aspect of this work is that it allows for the generation and maintenance of more than one representation, including the internal representation. This allows the set of operations applicable to a conceptual object of a certain type to be the union of all operations applicable to all its possible representations in underlying systems.

The described encapsulation principle of underlying software packages is generic and allows for ease of specification and thus addition of other packages. Moreover software packages with an interactive interface, possibly using several states can be encapsulated.

In conclusion, a powerful, generic mechanism for request execution and software encapsulation has been designed. This can easily accommodate the integration of arbitrary, interactive and non-interactive software packages.

Literature

Abelson, H., G. J. Sussman, et al. (1991). Structure and Interpretation of Computer Programs. Cambridge, Massachusetts, The MIT Press.

Bertino, E., M. Negri, et al. (1989). "Integration of Heterogeneous Database Applications through an Object-Oriented Interface." Information Systems **14**(5): 407.

Decouchant, D. (1986). Design of a Distributed Object Manager for the Smalltalk-80 System. Object-Oriented Programming: Systems, Languages and Applications (OOPSLA),

Desai, B. C. and R. Pollock (1990). "MDAS: Multiple Schema Integration Approach." IEEE Transactions on Software Engineering **13**(2): 16.

Eliassen, F. and J. Veijalainen (1988). A Functional Approach to Information System Interoperability. Research into Networks and Distributed Applications, Elsevier Science Publishers BV,
North Holland.

Hilhorst, R. A., L. K. J. Van Romunde, et al. "ROCHEFORT Research on Creating A Human Environment For On-line Research Tools." Statistical Software Newsletter **13**(2): 47.

Kuil, H. W. B. F. (1993). Generalized Encapsulation of Interactive Character Based Applications in YANUS. Dept. of Medical Informatics, Erasmus University. (Report)

Landers, T. and R. L. Rosenberg (1982). An Overview of Multibase. 2nd International Symposium on Distributed Databases, Berlin, North Holland Publishing.

Maes, P. (1987). Computational Reflection. Artificial Intelligence Laboratory, Universiteit Brussel.

McCullough, P. L. (1987). Transparent Forwarding: First Steps. Object-Oriented Programming: Systems, Lanugages and Applications (OOPSLA),

OMG (1992). Object Management Architecture Guide. Object Management Group, Inc. Framingham, Massachusetts.

Purdy, A., B. Schuchardt, et al. (1987). "Integrating an Object Server with Other Worlds." ACM Transactions on Office Information Systems **5**(1): 27.

Schrefl, M. and E. J. Neuhold (1988). Object Class Definition by Generalization Using Upward Inheritance. 4th International Conference on Data Engineering, Los Angelos, IEEE Computer Society Press.

Van Mulligen, E. M., T. Timmers, et al. (1991). A Framework for Uniform Access to Data, Software and Knowledge. Symposium on Computer Applications in Medical Care (SCAMC), Washington D.C., Mc Graw Hill.

Wiederhold, G. (1986). "Views, Objects, and Databases." IEEE Computer : 37.

Zdonik, Z. B. and D. Maier, Ed. (1990). Readings in Object-Oriented Database Systems. The Morgan Kaufmann Series in Data Management Systems. San Mateo, Morgan Kaufmann Publishers, Inc.

Chapter 5

Application of the YANUS model to the reference problem

5.1 Introduction

In the chapters 2 to 5 of this thesis, a model is described for the user-level integration of data and operation resources in YANUS. The described model is supposed to be general, and it must be able to handle a specific non-trivial problem. That problem is referred to as the reference problem (or the direct application area). It consists of offering to the user the integration of data resources, as provided by the DBMS "Ingres", operation resources as provided by a package for interactive statistical pattern recognition "Ispahan", and of simple arithmetic operations. This chapter discusses the application of the model to this reference problem.

The integration model has two major parts: The user interface, which includes the user model, and software package encapsulation. Figure 1.1. presents an overview of the YANUS architecture. YANUS is superimposed on pre-existing DBMS's and other software packages and consists of three components or levels. At the heart of YANUS is the conceptual interface (CI) which provides the user with typed conceptual objects and operations. The user accesses these objects and operations through the user interface (UI). The software package interface (SPI) encapsulates the underlying software packages.

The application of YANUS to the reference problem is meant as a test of the model. Specific questions are:

1. Whether and how the integrated resources can be embedded in the model of typed objects.
2. Whether and how Ispahan, Ingres and the arithmetic functions can be driven by YANUS, given the layered model of the SPI.
3. Whether YANUS offers any added value i.e., extra possibilities which the user of the separate packages and software does not have.
4. Given that the model handles the reference problem well, what may be concluded from that with respect to the generality of the model.

In this chapter (as in other chapters) it is made acceptable that the model has the ability to function correctly. It is not within the scope of this study to give a complete, detailed description, let alone a proof.

In describing and modelling the integration of all resources, two main aspects are of importance:

- The conceptual model: i.e., how are the resources modelled to the user. What are the object types, operations, workspaces, and how can they be combined.
- The request execution: how is the corresponding software be driven.

Certain essential, or representative parts of these aspects will be worked out. Details will be given partly in appendix A5.2, partly in the formalization. As in other chapters, description of the formalization is given in *italics*.

The main subdivision of this chapter reflects the two major aspects: Section 5.2 describes the conceptual model, section 5.3 describes the application of the request execution model. Section 5.2 is further subdivided as follows: First the conceptual models of the individual packages Ingres and Ispahan are discussed. The conceptual model of using arithmetic functions is considered to be trivial. Next, the way in which this functionality can be modeled to the user in YANUS is described. It will appear that the most logical subdivision in workspaces still reflects the separate resources. Special subsections are dedicated to the modelling of data resources as incorporated from Ingres and to the further structuring in sub-workspaces of the pattern recognition operation resources. Section 5.3 is subdivided as follows: In section 5.3.1, the control models of the two packages are discussed. Section 5.3.2 shortly discusses the resulting problems in the application of the execution model. In section 5.4. a discussion and conclusion is given.

5.2 The conceptual model

5.2.1 The conceptual model of Ingres

"Ingres" is a Database Management System (DBMS). It provides to the user facilities to store and retrieve structured data. The most basic way to do so is using queries to be expressed in one of the query languages SQL or QUEL. However, Ingres also provides a whole set of tools to create applications, reports, graphs etc. in order to achieve a more user friendly interaction, and / or user friendly insight in the results. These tools will not be discussed in the context of this chapter, partly because YANUS is directed at replacing the interface (i.e. applications

in this case) of the underlying package, partly because the integration discussed in this chapter is directed at allowing pattern recognition functionality to be applied to data as stored in the database, so that graph and report functionality is not part of the integration. Thus, Ingres will be driven via its query language interface.

The Ingres conceptual model is a relational DBMS. The principles of relational DBMS's and the different existing query languages will not be discussed in detail. See for an extensive treatment (Date). Briefly, in a relational DBMS data are stored in *relations*[1] (which may be compared to tables). Relations consist of *tuples* (the rows in a table) and have a fixed set of *attributes* (a table has a fixed set of columns). For a specific tuple, an attribute has a value (the value of a field in a row).
The following quote of (Date) clarifies some of the terminology used:
"It is frequently the case, that within a given relation there is one attribute with values that are unique within the relation and thus can be used to identify the tuples in the relation. [...] Such an attribute is called the *primary key*. Not every relation will have a single-attribute primary key. However, every relation will have some combination of attributes that, when taken together, have the unique identification property [...]. The existence of such a combination is guaranteed by the fact that a relation is a set: Since sets do not contain duplicate elements, each tuple of a given relation is unique with respect to that relation, and hence at least the combination of all attributes has the unique identification property. In practice it is not usually necessary to involve all the attributes- some lesser combination is normally sufficient [...]. The primary key is a combination in which no redundancy occurs: none of its constituent attributes is superfluous for the purpose of unique identification."

In Ingres, a user may have several databases at his disposal. A database may consist of several (generally related) relations. Queries are executed over one database at a time. Typically a query uses one or more relations from the database as input and produces a new relation as a result. In the terminology of the relational algebra (Date), the user has a collection of operations at his disposal that he can apply to the relations. The most important operations are:

- *Selection.* Tuples are selected for which a certain condition over some of the attributes is fulfilled.
- *Projection.* A relationship can be projected on a subset of its attributes. Those are the attributes of the resulting relationship;
- The *join.* Two relationships are combined in a specific way to form a new relationship. In a *natural join* (which is the default meaning of a join), the requirement is that both

[1] In the description of the user models of both Ispahan and Ingres, the use of words will be in conformity to the normal use of words in that area. In the description of the integrated functionality in section 6.2.2, uniform terminology will be introduced when necessary.

relationship have an attribute in common of which the values have the same domain. The attributes of the new relationship are a combination of the attributes of the two input relationships.

5.2.2 The conceptual model of Ispahan

Ispahan is a software package for interactive pattern recognition (see also (Gelsema 1980; Gelsema and Timmers 1990); from this last reference quotations are used below). In Ispahan (object) *populations* are subjected to analysis. A population consists of objects[2]. An object has certain features. Features are measured values, or otherwise objective (generally quantitative) information about the object. The aim of Ispahan is to be able to classify objects according to the values of their features. There are two main sorts of classification algorithms:

- The supervised classification algorithms. These can only be used if certain objects have been "labelled" with an a-priori classification. These algorithms use the feature values of the objects in the different a-priori classes to be able to classify objects for which the classification is unknown. For example, first the statistical distribution of the feature values for each a-priori class is derived, and subsequently classification can be performed on the basis of these distributions.
- The unsupervised classification algorithms. No a-priori classification is available. Objects are "clustered" into classes on the basis of some distance measure in the feature space.

Ispahan provides other tools such as histogram and scatter plot generation, feature evaluation - , mapping and transformation algorithms to provide insight in the structure of a population in the high-dimensional feature space. Mappings map the objects from the full dimensional feature space to a plane. Transformations transform the feature space. An example is a principal components transformation to decorrelate the features.

Important data structures are trees. Two main types of trees are the population tree and the decision tree.

- A *population tree* describes a subdivision of a population into sub-populations. The top node represents the complete population; descendant nodes represent subpopulations.

[2] Not in the object-oriented sense. See section 5.2.3

These subpopulations have empty cross sections. A descendant node may again have descendants, etc.

- A *decision tree* is a population tree, which has a decision function attached to at least one of the branch points (non-terminal nodes). This describes a classification rule which has been designed on the basis of the (sub)population represented by that node. Such a decision function can be used to classify another comparable (sub)population as represented by a ("test")node in another (population) tree, as a result of which descendant nodes are created representing the subpopulations in which the objects in the testnode are subdivided.

Each node of a population tree has a so-called *feature mask*. This mask specifies the status of each feature. The status may be on or off. [.] Any operation requested at any node in a population tree will operate on the object feature vectors according to the feature mask, i.e. only those features which according to the mask are switched on are taken into account. Thus, decision functions, mappings, etc. may be designed using different subsets of features at different nodes in the tree. [.] The feature evaluation algorithms manipulate this feature mask.

In addition to population and decision trees, so-called selection trees are used. [.] Instead of subpopulations being attached to the nodes [of such a tree], each node, except for the root node, specifies criteria in terms of one or more feature values, which when applied to an object (sub)population will result in a population tree. [.]

5.2.3 Conceptual model of the integrated resources in YANUS

To integrate the resources of Ingres and Ispahan as described in 5.2.1 and in 5.2.2, the conceptual models must be amalgamated. First note that a useful integration of both packages may be achieved when the DBMS contains (a) relation(s) which may be regarded to describe objects in a population, the attributes corresponding to features.

For the sake of removing ambiguity in the following discussion, the terminology of both conceptual models will be somewhat adjusted to harmonize with each other and with the terminology used in the YANUS general conceptual model. Instead of "relation" (which should not be confused with a "relationship" in YANUS) the less formal, but commonly used term *table* will be utilized. Instead of a "tuple" the word *record* will be used. The values in a record may in principle be stored as attributes in the YANUS object model and terminology, but in general each value will be stored in the attribute of an *atomic object*, i.e., an object having a single attribute. We will say that a record consists of *fields*. Thus, fields will generally be atomic objects, component objects of a record. Instead of objects, a population

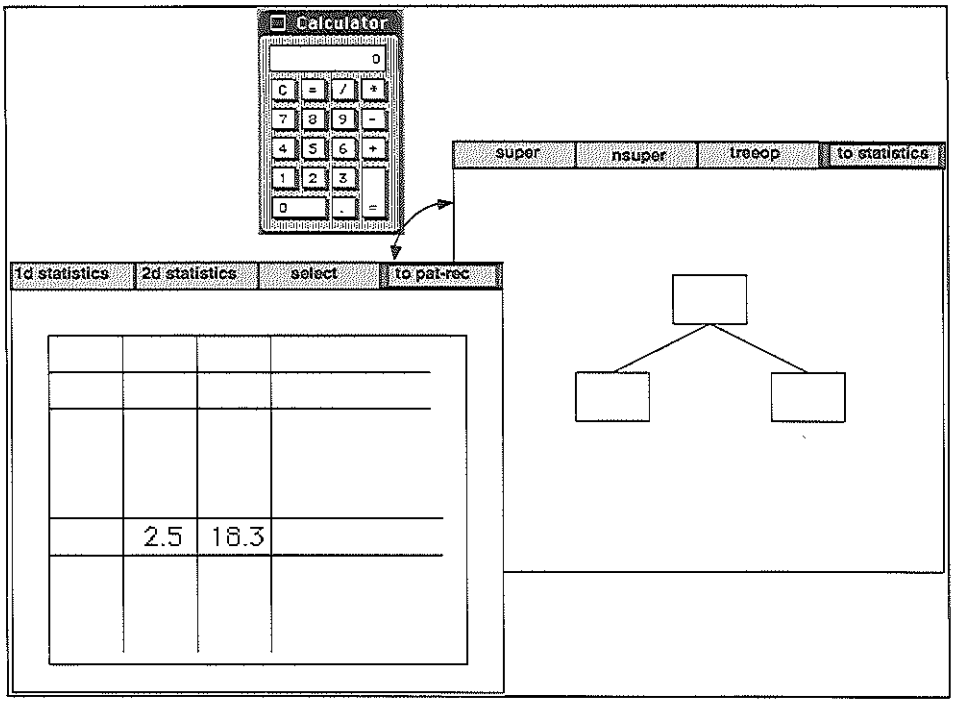

Fig. 5.1

Possible presentation on the screen of the workspaces modelling the database, statistical pattern recognition and arithmetics resources. Using the buttons "to pat-rec" and "to statistics" the user can switch between the two workspaces. The two workspaces cannot be open at the same time. Note that the switch operation as such has not been modelled as a separate operation: it consists of closing the first workspace and opening the other. The calculator can always be open; the arithmetic operations it presents can be applied to the numbers presented in the table, since these are modelled as atomic objects. The statistical operations have not been modelled formally.

will be said to exist of *individuals*. A population is a table of which each record represents an individual. In a population table, there is always one field that contains the a-priori classification label of each individual even if all labels are identical for all individuals. All other terminology remains the same.

In broad terms, the integration of database data resources, pattern recognition and arithmetic operation resources may be modelled as follows: The user is allowed to perform projections and selections on population tables -for simplicity the join is not included, see section 5.2.1- in order to derive a final *personal copy* of the table. This personal copy may be analyzed by applying pattern recognition functionality. The atomic objects containing numbers corresponding to the fields of the records may be used as input for arithmetic operations.

In terms of workspaces, a separation is made between workspaces in which tables may be manipulated (see section 5.2.1) a workspace in which pattern recognition operations may be applied to the population, and a workspace which (only) gives access to arithmetic operations. The user may switch between a workspace presenting a (valid personal copy of a) population and a workspace in which pattern recognition is applied to the same population. Fig. 5.1 gives a possible presentation of this scenario. Appendix A5.2 the main lines of the complete conceptual model is given, this includes the model of the arithmetic operations.

5.2.3.1 The conceptual model of DBMS functionality within YANUS

In the integration, the focus is on population tables. All features which may be used to classify individuals (and which are thus "descriptive") will generally be stored as in one table. Queries ranging over more than one table (joins) will not be considered.

It is assumed, that a certain population table (or otherwise table) may be reached via the complete component hierarchy which starts at the root object (see chapter 3). This means that the population table is represented by a conceptual object; the object's type declares the object to be a table. The user is allowed to apply queries to such a table. A query involves the combination of a projection and selection, similar as in the query language SQL. The result of the query is again a table object.

Formally three types of population tables are used:

- The *original* population table corresponding directly to the database. This table may possibly be shared with other users. Since YANUS keeps a copy of this population table, and changes are only saved when the whole population is saved (see chapter 3), such changes may conflict with changes made by others at the same time. Such conflicts will be avoided. The assumption is, that the user wants to make a snapshot

of the data to subject them to analysis. He is not allowed to change data in the original population table[3]. This is modelled by letting the population table and its records have non-changeable properties, while the population table is set to valid and thus frozen directly after reading (chapter 2). The user may query (i.e. perform a projection or selection on) this population, and thus obtain a "derived population". The user may also directly create a personal copy.

- A derived population table. This is similar to a "view" on the original population table in database terminology. The user is not allowed to change the data. Removing records is allowed, but is only viewed as a further restriction of the selection (not as a delete). A further query may be made, as a result of which a newly derived population table is obtained. Records that were removed are also removed in this newly derived population table. The user may also create a personal copy.
- The user's personal copy of a population table. By creating this copy the user prevents other users from changing the data on which he is performing analysis. The personal copy will be stored as a separate database. Initially the personal copy will be changeable: updating and deleting records is allowed. However, the user may require to set the table to valid. As a result the table "freezes"; it can no longer be changed. To this valid/ frozen population table the user may apply pattern recognition functionality, i.e., he may switch to the corresponding workspace. By requiring the table to freeze before pattern recognition operations are applied to it, invalidation of the population trees corresponding to the population can be prevented. Further querying may also be performed on this personal copy, in which case a derived population table is again obtained (from which again another copy may be made, etc.).

A query operation must be modelled such that a projection can only be made on a subset of the set of fields of the current population table. This can be done by defining this as an extra requirement for testing the validity of the request (see chapter 2). However, such an extra requirement cannot be taken into account in user guidance. In other words, the data model does not support the explicit modelling of such a constraint. User guidance in this situation may be provided as an ad-hoc solution at the presentation level: The set of appropriate field descriptors is presented as part of the interaction tool through which field descriptors may be selected in the formulation of the query. Other forms of guidance in correctly formulating queries are not treated in this thesis.

[3] Mechanisms exist, such as e.g. described by (Hornick, Morrison et al. 1991), which avoid these kind of conflicts by detecting them: A time stamp is used in the database, which indicates the last time an update has occurred. A user X is not allowed to update a table if another user Y has updated the table after X had read it. This however requires a time stamp in the underlying database, which will generally not be available.

Correct type information for each table is a prerequisite for both user guidance and checking with respect to the formulation of the projection request. Moreover, such type information is needed to enable parsing of the query result (see section 5.3.2.1). When a query involving projection on a subset of the fields is executed, the type of the output table must therefore be derived. This will be described in section 5.3.

The global conceptual model as given in appendix A5.2 refers to meta types, primal types and operations as are introduced in the formalization:

The first step in modelling tables and records is modelling the corresponding meta types, i.e. the types of all table and record types. This is done as follows, and for the following reasons:

- *Each table type prescribes the component property "Records". To model this, first a meta type "WholeObjectType_type" is introduced. This is the type of types prescribing only component properties; next a meta type "SimpleWholeObjectType_type" is introduced. This is the type of types prescribing one and only one component property, such as the property "Records". The meta type "TableObjectType" is a subtype of "SimpleWholeObjectType_type. Note that the use of an adapted operation "DeriveProperties" as described by post_DeriveProperties plays an important role in this; see also chapter 2.*
- *Since a record has a set of primary keys and other fields, a record type is characterized by corresponding sets of key descriptors and field descriptors. This is modelled by the meta type "RecordType_type".*
- *Key descriptors as well as field descriptors are a special kind of property descriptors having as elementtypes only so-called simple atomic object types, i.e. object types of objects having only one attribute. To model this, first the type "AtomicObjectType_type", of all atomic object types is introduced, instances of which prescribe only attributes; a subtype of that is "SimpleAtomicObjectType_type" instantiations of which prescribe only one attribute. The modelling of the special kind of key and field descriptors will be explained in the following.*
- *Derived and original tables are not modelled internally in the same way: records in derived tables do not own the atomic objects containing the field values, but have a cross reference to the corresponding fields in the original table. In this way, changes in the fields of the derived table will also appear in the original table, and the other way around. Therefore, for both key descriptors and field descriptors a distinction is made between two groups: field (or key) descriptors which are component descriptors for the record type of an original table, and field (or key) descriptors which are cross reference descriptors for the record type of a derived table. Field descriptors and key descriptors which are component descriptors and are only allowed to prescribe simple*

atomic element types (see previous point) are instances of "SimpleAtomicComponent-Descr_type"; The other group of Field and key descriptors are instances of "SimpleAtomicCrossRefDescr_type".

- *Now the subdivision at meta type level between the type of all original record types "OriginalRecordType_type" and the type of all derived record types "DerivedRecord-Type_type", both subtypes of "RecordType_type" can be understood: "OriginalRecord-Type_type" prescribes record types to have key and field descriptors of the type "SimpleAtomicComponentDescr_type"; "DerivedRecordType_type" similarly prescribes the type "SimpleAtomicCrossRefDescr_type" for key and field descriptors.*

"AtomicObjectType_type", "SimpleAtomicObjectType_type", "WholeObjectType_type", "SimpleWholeObjectType_type", "SimpleAtomicCrossRefDescr_type", "SimpleAtomic-ComponentDescr_type" and the corresponding post conditions of "DeriveProperties" are defined in definition 1. "TableObjectType_type", "RecordType_type", "OriginalRecordType-_type" and "DerivedRecordType_type" are defined in definition 2.

At the type level, the following primal types are modelled in definition 3:

- *The supertype of all table types "TableObject_type", which prescribes that each table has the property "Records" in which all records are held.*
- *The supertype of all original table types, "OriginalTableObject_type" which prescribes that records must be of the type:*
- *"OriginalRecordObject_type", the type of records in original tables, the primal type of "OriginalRecordObjectType_type".*
- *The supertype of all derived table types, "DerivedTableObject_type which prescribes that records must be of the type:*
- *"DerivedRecordObject_type", the type of records in derived tables, the primal type of "DerivedRecordObjectType_type".*

At this level, the operation "Query" is also modelled. It has an input operand, the table to which it is applied and an output operand, the resulting table, which is always a derived table. Through the setting "Fields" it models explicitly the fields on which the projection is performed. Through the setting "Predicate" it models the selection predicate. This has only been worked out as far as a boolean function using a set of fields as input. req_{odqj} defines the extra requirement that a projection can only be made on a subset of the set of fields of the input table. Since Query is implemented differently for original tables and derived tables, see section 5.3.2.1, iQuery has two subtypes with the same name, iQuery' and iQuery'', applicable to the corresponding types.

5.2.3.2 The conceptual model of Pattern recognition functionality within YANUS

A formal definition of types and (a representative set of) operations used to model pattern recognition operation resources, and the corresponding population trees is given in appendix A5.2. This basically reflects data structures and operations as used in Ispahan. Furthermore, a subdivision in workspaces is given.

In the pattern recognition workspace, the object of interest is the user's personal copy of a population table. This workspace presents all trees and their constituent nodes corresponding to this population. Furthermore, the workspace offers the major part of all pattern recognition operations. Certain operations are offered in separate workspaces. This will be discussed further on. Note, that the given subdivision in workspaces allows a further subdivision of the presentation on the physical screen, e.g., presenting trees by means of icons which may be opened for inspection, or operations by means of pull-down menus. These are, however, not modelled at the conceptual level[4].

For certain (groups of) operations within the pattern recognition functionality, separate workspaces are needed. One such workspace is used to create and edit a scatter plot, see appendix A5.2. The scatter plot is created when the workspace is opened. Applicable operations are e.g.: create a piecewise linear discriminant ("Cleave"), zoom-in ("Scale"), and "ChangeFeatures". As described in chapter 3, the workspace provides an environment in which local parameters can be held, which can also be propagated to the requests which are created within that workspace. For instance, the features which are to be plotted along the axes are initialized as settings of the workspace and propagated to the requests created within the workspace.

A special feature not provided by the data model is needed in this workspace. This is a result of the fact that the scatter plot is not an independent, addressable object in Ispahan. The scatter plot exists, and certain operations may be applied to it, only in a specific state of Ispahan. When the state is changed in order to execute other operations, the scatter plot disappears. Thus, in this case it is not possible to hide this usage of states in Ispahan from the user. This means that other operations should not be requested during the time that the scatter plot workspace is open. For this purpose, the boolean property "Exclusive_Use" of a workspace is introduced; if this property is true, all other workspaces are locked as long as the workspace which requires exclusive usage is open. This is not modelled formally.

[4] Note that opening and closing of workspaces corresponds to explicit operations executed by the underlying software; this while opening and closing of icons is purely concerned with the presentation of data on the physical screen.

The workspace clgrow is a similar case: This workspace corresponds to an iteration in the underlying program. Only operations controlling the iteration are allowed. This workspace must therefore also require exclusive usage.

Note that StandardStructChange (see chapter 3) is defined to be "applicable" to tables and thus to populations, but not to population trees. Changes in tree structure can only occur due to the application of specific operations such as expand.

5.2.3.3 Conceptual model of arithmetic functions

Arithmetic functions are simply modelled as being applicable to atomic objects containing real numbers. See appendix A5.3. Atomic objects containing integer numbers are modelled as subtypes, so that these can also be used as input. A separate workspace, called "Numbers" is modelled to present these operations. Results are supposed to be free objects of the workspace.

5.3 Driving the underlying packages and retrieving the result

5.3.1.1 The control model of Ingres

As mentioned, Ingres will be controlled using a query language. At any moment within a session of the query language, access is given to the tables of one particular database only. Access to another database is gained by starting up another session. (This can be done by another software package manager, so that more than one session may be running at any one time.) Results of a query are directly returned.

5.3.1.2 The control model of Ispahan

Ispahan will be driven by the YANUS SPI via its user interface. This is possible since input for Ispahan is mainly ascii. Furthermore, relevant output, i.e., output concerning changes in trees and/ or newly created trees, can also be obtained in ascii form using the "help" option.

Ispahan has a menu driven user interface. An option in a menu may either provide direct access to an algorithm, or gives access to a submenu. Thus, all menus and their submenus form a tree (the menu tree). Each node in the tree corresponds to a state of Ispahan, in which it can execute certain commands. If an algorithm is completed, or if the use of a submenu is finished, the interface automatically returns to the menu from where the algorithm was started, or from which the submenu was started, respectively. Appendix A5.1 gives an overview of the menus and their options. A command to start an algorithm may need certain parameters. These parameters are requested in a fixed sequence. A command to go to a submenu may also

involve entering parameters e.g. when choosing the option scatter in the plots menu, a population tree, a node in that tree, and the class definition[5] will be prompted for.

Special options (so-called "manager" options) are invoked to make certain data accessible within Ispahan: In order to create an object population a file containing all data concerning the population must be read-in. Existing object populations and trees must be opened to make them accessible.

Ispahan has certain "hidden" options which may be executed from any state. The above mentioned "help" option is one of these.

5.3.2 Executing requests

5.3.2.1 Executing query requests in Ingres

Creating a query and sending it to Ingres underlies all operations on Ingres and is relatively simple: A query request has a setting "Fields" prescribing the fields on which a projection is done and a setting "Predicate" prescribing a selection criterion. A special operation implementation generates the query text. Another sends the text to Ingres and parses the results, see chapter 4. Ingres has only one state, so that no commands need to be inserted in order to change the state.

Specific tasks involved in driving Ingres concern manipulating type information:

1. When executing a projection, the type information of the new table must be derived given the type information of the old table and the list of fields on which the projection is done.
2. Before executing the query, the grammar of the output must be generated so that it can be parsed.

Moreover, when a table is examined for the first time, type information about this table must be retrieved. Incorporating a new database (of which the database manager is of a "known" type) is possible in principle in the integration model, but has not been studied in detail.

[5] Class definition is either priori or posteriori, describing whether the objects in the scatter plot must be labelled according to their a priori classification, or to their a posteriori classification, in which case their classification as given by the descendants of the node is used as labelling.

Thus, as part of the execution of a query operation extra type information, or information related to type information (i.e. the output grammar) regarding the result of that operation must first be derived. This can also be done using extra operation implementations coupled to the conceptual operation, see also the formalization description.

Definition 4 gives the definition of implementation types of tables and records, including those of population tables. Original tables and Derived tables have not only a different internal representation, they also have a different way of representing the underlying data through implementation objects. Implementation objects of original tables directly refer to the database and table within that database through the corresponding names, as modelled by the type "ImplOriginalTable_type", and the modelling of addresses in implementation objects through the type "DatabaseAddress_type". The address object may also contain a query text; this will be used for simplifying execution of requests and will be discussed at a later stage. Implementation objects of derived table objects contain address objects, modelled by the type "ViewDescr_type". These address objects not only contain the database and table name (and a query text), but also contain a query which is an instantiation of the operation description Query, see formalization of section 5.2.3.1. This represents the fact that the implementation object represents a result of a query. It could be said that the implementation object describes a view. Since records may be removed from a derived table, the address object of a derived table objects also contains the keys of removed objects; this is also part of the view description. Both implementation objects of original tables and implementation objects of derived tables have an attribute "Grammar", which is used to contain a derived grammar (see text, and definition 5). One other implementation type, specific for PersonalCopyPopulations (the corresponding type "PersonalCopyPopulation_type" is a subtype of "OriginalTable_type") is "ImplIspahan_type". The address of the corresponding implementation objects only contains a file name. The implementation object for records as modelled by ImplRecord_type keeps the corresponding keys in the corresponding address object. Finally, implementation types for fields are (e.g.) IngresReal4_type and IngresReal8_type, which correspond to different possible storage representations for reals in Ingres.

Definition 5 gives some details with respect to executing a query request. This is done for the application of a query request to an original table; some remarks will also be given about the case of applying a query to a derived table. The implementation operations corresponding to the query operation are: "DeriveType", "SetOutputImplObject", "DeriveQueryText", "DeriveGrammar", and "DatabaseQuery". "DeriveType" works on the internal representation of the input table and the output table (which already exists due to the request execution mechanism, see chapter 4); it derives the type of the output table, given the type of the input object and the fields on which it is projected. This is similar for application of a query to a derived object. "SetOutputImplObject" fills in the implementation object of the output of the

query so that it describes the view as mentioned earlier. Specifically, the address is set to contain a copy of the query request. For a query applied to a derived table object, specifically this task is implemented differently: in that case the query as contained by the address object of the implementation object of the output table is set to be a composition of the query held by the address of the input implementation object and the query requested by the user, so that the new view is described relative to the original table, not with respect to the view from which it was derived. "DeriveQueryText" derives the query text which must be sent to the underlying DBMS, as corresponding to the view description held by the output implementation object. Note this use of the view description is the most important reason for keeping it. "DeriveGrammar" derives the grammar of the output. The result of this is stored in the attribute Grammar of the output implementation object. No further details are given. "DatabaseProject" is the implementation operation which finally sends the query to the database and parses the result.

Definition 6 describes some transformations at the implementation level as defined for tables and populations, and the way these transformations are incorporated in the corresponding transformation table. Specifically the following compound transformations are defined as part of the transformation table of "OriginalObject_type":

- *From the database representation of the original table to the internal representation; this corresponds to one transformation operation called "CreateOriginalInternTable-fromDB".*
- *From the internal representation of the original table to an ispahan file; this corresponds to one transformation operation called "CreateFilefromOriginalIntern-Table".*

Due to the transitive description of compound transformations in the transformation table, the transformation table therefore also contains the compound transformation from the database representation of the original table to an ispahan file; this corresponds to the two transformation operations "CreateOriginalInternTablefromDB" and "CreateFilefrom-OriginalInternTable" in sequence. This allows the user to apply the pattern recognition module directly to a stub personal copy population.

5.3.2.2 Driving Ispahan

More than in driving Ingres, driving Ispahan involves managing the correspondence between the underlying objects and the objects in YANUS. For instance, the following happens when the user tries to execute the operation "Test" (see appendices A5.1, A5.2 and the formalization), in which a new tree is created. In order to initialize the request, the user must select the two nodes "Pnode" and "Dnode". The correspondence of these nodes, and the nodes and trees

in Ispahan is given by the address in each of the implementation objects, which contains both the tree name and the node name. The user must also provide the name ("Newtrenam") of the tree to be created, in a setting. All these names will be transferred at invocation of the Test operation in Ispahan. After execution of the operation in Ispahan, information about the newly created tree which has the name as given by the setting "Newtrenam", is obtained from Ispahan using the "help" option. On the basis of the result of that command invocation, the parser creates the corresponding tree in YANUS.

Another major aspect of driving Ispahan is the state management. The general model for state management ("planning") has been discussed in chapter 4.

Another aspect of driving Ispahan is, that Ispahan sometimes provides graphical feedback and even offers graphical interaction -e.g. for creating a piecewise linear discriminant in a scatter diagram-. Since replacing this functionality in the YANUS interface is not very useful and time consuming, YANUS must be transparent for this interaction. To allow YANUS to be transparent for pre-existing graphical feedback and dialogue, one of the possible actions of the parser (which parses the output of the underlying package) is to create a separate window and to pass on the output of the package (selectively) to the window and reversely, to pass on user's actions in that window to the package.

Definition 7 defines the implementation type of an implementation object representing a tree in Ispahan. Definition 8 describes the implementation operations that correspond to the execution of the conceptual operation "Test" (as defined in appendix A5.2). One implementation operation is called "SetOutputObject". Note that the output object is automatically created as part of the execution mechanism before the implementation operations are invoked. "SetOutputObject" stores the name of the new tree (Newtrenam) in the output object. "SetOutputImplObject" does the same for the implementation object of the output tree. "TestInvoke" invokes the Test command and subsequently invokes the "help" command to retrieve the information about the new tree. Definition 9 defines the transformation operation which reads the internal representation of a tree from the external, Ispahan representation. In (Kuil 1993) is described how the general software package manager and parsing software as implemented in this project, can be used to drive Ispahan. Note however, that the creation of objects (internal representation of trees in Ispahan) has not been incorporated in the implementation.

5.3.3 Driving Arithmetic software

Arithmetic operations are directly implemented on the internal representation. Commands coupled to the implementation operations directly correspond to function calls. No states, state transitions nor parsing is needed.

Definition 10 shows how the conceptual operation "Add" can be coupled to the implementation operation "ImplAdd" which works directly on the internal representation. The corresponding post condition describes the result of the application of ImplAdd.

5.4 Discussion, Conclusion

In this section, answers to the questions posed in the introduction (section 5.1) of this chapter will be discussed.

1. It has been shown in this chapter that the principle of typed objects and workspaces can be used to model the integrated resources of Ingres, Ispahan and arithmetic operations. Specifically, there is no need for modelling forced sequences of operations in the conceptual interface, something which is outside the scope of the YANUS model.
   Note, however, that the reference problem served as feedback in the development of the model. The final form of the model has clearly been influenced by the reference problem.
   Certain extensions not described in chapter 2 - 4 were still needed as described in this chapter, such as the introduction of exclusive usage of a workspace.
2. Both Ingres and Ispahan may be driven using the request execution model as described in chapter 4. Error handling is still an open issue.
3. The added value of the integration of Ingres and Ispahan is the possibility to do pattern recognition directly on the data selected from a database. In principle, other possibilities exist: for example, showing the contents of a node in a population tree as a table, and allowing further selection to be applied to it. This has not been worked out any further.
4. The model being applicable to the integration of Ingres and Ispahan cannot be viewed as a proof for its generality. This is especially so, since it has been developed keeping this application in mind as a concrete example and test. Still, this result may be generalized somewhat in the following ways:

Ingres is a relational DBMS, providing a query language interface (even a standard query language, SQL). Thus, the integration of Ingres seems exemplary for many such DBMS's with respect to both the conceptual model and the control model. Note, however, that only a direct mapping has been modelled between the (structure of) the database data and the conceptual objects as provided by the user, and that the join operation has not (yet) been modelled.

The control model of Ispahan seems to be a relatively complex case. This suggests (and it appears to be that way for Ingres) that for the integration of other packages not all features of the execution model as described in chapter 4 are relevant. Furthermore, the control model of a menu tree, where each menu corresponds to a state of the package is used more often; for example the package "Harvard Graphics" has such a control model.

Literature

Date, C. J. (1981). An Introduction to Database Systems. Reading, Massachusetts, Addison-Wesley Publishing Company.

Gelsema, E. S. (1980). ISPAHAN: an Interactive System for Pattern Analysis: Structure and Capabilities. Pattern Recognition in Practice, Amsterdam, North-Holland Publishing Company.

Gelsema, E. S. and T. Timmers (1990). ISPAHAN/IPACS User's Manual. Erasmus University, Department of Medical Informatics.

Hornick, M. F., J. D. Morrison, et al. (1991). Integrating Heterogenous, Autonomous, Distributed Applications using the DOM Prototype. GTE Laboratories.

Kuil, H. W. B. F. (1993). Generalized Encapsulation of Interactive Character Based Applications in YANUS. Dept. of Medical Informatics, Erasmus University. (Report)

Appendix A5.1

Overview of the Ispahan menu structure. At the top of each menu a comment is given, preceded by **. Next, the name of the menu is given. Subsequently, all options in the menu are given, and for each option its parameters and whether it corresponds to a submenu call. Finally, for each parameter the type is given: name (NM), node (ND), integer (IN), Yes/No answer (YN), character (CH), vector point in the plot (VP), real (RE) or octal (OC).

```
**      MASTER MENU
        MASTER
MANAGR                                                  submenu
TREEOP      TREE                                        submenu
SCATTR      POPTRE NODE IDENT                           submenu
SUPER                                                   submenu
NSUPER                                                  submenu
MAPPNG      POPTRE NODE                                 submenu
TRANSF                                                  submenu

POPTRE      NM
NODE        ND
IDENT       NM
TREE        NM


**      ISPAHA NMANAGER
        MANAGR
CREATE                                                  submenu
OPEN        NAME
LIST
SPLIT       POPTRE NEWTR1 NEWTR2 RATIO
COPY        TREE NODE NEWTRE OBJPOP
CLOSE       NAME
KILL        NAME
DIFF        TREE1 NODE TREE2 NODE

NAME        NM
POPTRE      NM
NEWTR1      NM
NEWTR2      NM
RATIO       IN
TREE        NM
NODE        ND
NEWTRE      NM
OBJPOP      NM
TREE1       NM
```

```
NODE        ND
TREE2       NM
NODE        ND


**      TREE OPERATIONS
        TREEOP
DESCND      NODE
ASCEND
DELNOD      NODE
DELSUB      NODE
EXPAND      NODE NSONS
MASK        NODE                                          submenu
STAT        NODE COVARCOREL
MAP         NODE LIST?
CLUCON      NODE
HDCOPY

NODE        ND
NSONS       IN
NODE        ND
COVAR       YN
COREL       YN
NODE        ND
LIST?       YN
NODE        ND


**      BUILD/VIEW SCATTER PLOTS
        SCATTR
LSTNMS
SETCOD      CLASS LABEL
FEATS       FTNAM1 FTNAM2
SCALE       F1MIN F1MAX F2MIN F2MAX
VIEW        CLASS
ZUMIN       X1 Y1 X2 Y2 LIST?
ZUMOUT
CLEAVE      SON1 SON2 NEWTRE
HISTOG      AXIS NBINS                                    submenu
HDCOPY

CLASS       NM
LABEL       CH
FTNAM1      NM
FTNAM2      NM
X1Y1        VP
```

```
X2Y2        VP
LIST?       YN
F1MIN       RE
F1MAX       RE
F2MIN       RE
F2MAX       RE
AXIS        CH
NBINS       IN
SON1        ND
SON2        ND
NEWTRE      NM


**      VIEW HISTOGRAMS
        HISTOG
VIEWHS      CLASS
MARK        MARK
HDCOPY

CLASS       NM
MARK        VP


**      SUPERVISED CLASSIFICATION
        SUPER
FISHER      POPTRE NODE
LINMAX      POPTRE NODE
QUAMAX      POPTRE NODE
BAYES       POPTRE NODE
(QM)NN      POPTRE NODE
READDF      POPTRE NODE
CLRDF       POPTRE NODE
DISCRE
TEST        POPTRE PNODE DECTRE DNODE NEWTRE PRIOR? VALMIS

POPTRE      NM
NODE        ND
POPTRE      NM
PNODE       ND
DECTRE      NM
DNODE       ND
NEWTRE      NM
PRIOR?      YN
VALMIS      RE
```

```
**      UNSUPERVISED PROCEDURES
        NSUPER
CLGROW  POPTRE NODE
KMEANS  POPTRE NODE NCLASS NEWTRE NORM MODE NSTEPS
AGGLOM  POPTRE NODE NEWTRE
KNEAR   POPTRE NODE KNEAR MODE
HIERCL  POPTRE NODE

POPTRE  NM
NODE    ND
NCLASS  IN
NEWTRE  NM
NORM    IN
MODE    NM
NSTEPS  IN
POPTRE  NM
NODE    ND
NEWTRE  NM
POPTRE  NM
NODE    ND
KNEAR   IN
MODE    NM


**      FEATURE MASK MANIPULATION
        MASK
VWMASK
FEATON  FTNAME
FEATOF  FTNAME
SETOCT  MASK1 MASK2 MASK3
AUTMSK  NFEATS NSTEPS FTOENT FTOREM
TTEST   CNFLIM

FTNAME  NM
NFEATS  IN
NSTEPS  IN
FTOENT  RE
FTOREM  RE
CNFLIM  RE
MASK1   OC
MASK2   OC
MASK3   OC


**      CREATION OF NEW POPULATIONS AND TREES
        CREATE
```

```
READFL    DATAFL OBJPOP POPTRE NOBS NCLASS NFTTOT NFTIS PHEADFL
SELECT    OBJPOP SELTRE NODES
FILTER    POPUL NODE SELTRE NEWTRE
CONFUS    POPTRE
DISCRE
MISSNG

DATAFL    NM
OBJPOP    NM
POPTRE    NM
NOBS      IN
NCLASS    IN
NFTTOT    IN
NFTISP    IN
HEADFL    NM
OBJPOP    NM
SELTRE    NM
NODES     IN
POPUL     NM
NODE      ND
SELTRE    NM
NEWTRE    NM


**    MAPPING ALGORITHMS
      MAPPNG
NLMAP     FTNAM1 FTNAM2 MAGFAC NITER                    submenu
TRIANG    REF.PT                                        submenu
OPTPLN    IDENT
DECLUS    IDENT ALPHA NODE
FISDEC    IDENT ALPHA NODE
DIS2MN    IDENT
OLDMAP    POPTRE NODE IDENT

FTNAM1    NM
FTNAM2    NM
MAGFAC    RE
NITER     IN
REF.PT    NM
IDENT     NM
ALPHA     RE
NODE      ND
POPTRE    NM
NODE      ND
IDENT     NM
```

```
**      SAMMONS NON LINEAR MAPPING
        NLMAP
CONTIN      MAGFAC NITER
HDCOPY
SCATTR      IDENT

MAGFAC      RE
NITER       IN
IDENT       NM


**      TRIANGULATION MAPPING
        TRIANG
TRIANG      REF.PT
HDCOPY
SCATTR      IDENT

REF.PT      VP
IDENT       NM


**      TRANSFORMATIONS
        TRANSF
PRINCO      POPTRE NODE NEWPOP NFEATSSCALE
SCALE       POPTRE NODE NEWPOP NORM
ODV         POPTRE NODE NEWPOP NFEATS
FISMDV      POPTRE NODE NEWPOP NFEATS
POCAL       POPTRE NODE NEWPOP NOPER
PRIORS      POPTRE NODE NEWPOP
DISCRE

POPTRE      NM
NODE        ND
NEWPOP      NM
NFEATS      IN
SCALE?      YN
POPTRE      NM
NODE        ND
NEWPOP      NM
NORM        IN
POPTRE      NM
NODE        ND
NEWPOP      NM
NOPER       IN
```

Appendix A5.2

Parts of the schema used to describe the integration of Ingres and Ispahan.
Syntax description (informal):

Each type (each instance of Type_type), including operation descriptions, modules etc. is defined using the following template:

Define Type-name **inst_of** name-of-Metatype
[
    Property-name: value

    or

    Property-name:
    [
        compound value
    ]
]

An important property of a module -of an instance of CreatableModule_type- is Conttypes. The types that are contained through this property are defined in the value block of the property Conttypes; thus:

    Conttypes:
    [
        Define Type-name ...
        ...
    ]

Property descriptors are defined as part of the value block of (e.g.) OwnPropDescriptors. The definition of a property is as follows:

    Property-name: Elementtype-name;

(MinElt = MaxElt = 1 of the property descriptor) or

    Property-name: { Elementtype-name };

(MinElt = 0, MaxElt = ∞) or
    Property-name: { Elementtype-name }+;

(MinElt = 1, MaxElt = ∞) or

    Property-name: { Elementtype-name }[MinElt:MaxElt];

By default: changeable = true, unique = true; if otherwise these are set using the corresponding keywords in bold between brackets, e.g. (**unchangeable**).

For operands the keywords **in out in_out** and (**invalid**) are used. By default the required validity of an operand is valid. Whether an operand is an input, output or both must always be given. For output operands specifically, minelt is always 0 (see chapter 2).

A parser for this syntax has not been implemented.

In module PopulationBrowse_module three types of population tables are defined: OriginalPopulationTable, DerivedPopulationTable, and PersonalCopyPopulationTable (see text). Note that in the definitions of these types, the metatypes TableObjectType_type, OriginalRecordObjectType_type and DerivedRecordType_type and the corresponding primaltypes are used, see the formalization. The module PopulationBrowse_module further references the operations Query and MkPrsnlCpy; these are also defined in the formalization.

In the module PatternRec_module the type definitions of a Tree and its Nodes can be found. Furthermore, (pattern recognition) operations applicable to trees and nodes are defined.

Finally, in the module Number_module, the type of a real object is defined and the corresponding operations.

```
Define PopulationBrowse_module inst_of CreatableModule_type
[
        Supertype:    WorkspaceReq_type
        OperandDescriptors:
        [
                Population:    in (synt_valid) OriginalPopulation_type;
        ]

        Conttypes:
        [
        Define OriginalPopulation_type inst_of TableObjectType_type
        [
                Supertype:    OriginalTableObject_type
                OwnPropDescriptors:
                [
                        Records: { OriginalIndividual_type };
                ]
        ]

        Define OriginalIndividual_type inst_of OriginalRecordType_type
        [
                Supertype:    OriginalRecordObject_type
                                /* primal type! */
```

```
        OwnPropDescriptors:
        [
                priorclass: StringObject_type (unchangeable);
        ]
]

Define DerivedPopulation_type inst_of TableObjectType_type
[
        Supertype:    DerivedTableObject_type
        OwnPropDescriptors:
        [
                Records: { DerivedIndividual_type };
        ]
]

Define DerivedIndividual_type inst_of DerivedRecordType_type
[
        Supertype:    DerivedRecordObject_type
        OwnPropDescriptors:
        [
                priorclass: StringObject_type (unchangeable);
        ]
]

Define PersonalCopyPopulation_type inst_of TableObjectType_type
[
        Supertype:    OriginalTableObject_type
        OwnPropDescriptors:
        [
                Records: { OriginalIndividual_type };
                Trees: { Tree_type };
        ]
]

]

Contops:
[
]

CRefops:
[
        Query; MkPrsnlCpy;
]

]
```

```
Define PatternRec_module inst_of CreatableModule_type
[
        Supertype:    WorkspaceReq_type
        OperandDescriptors:
        [
                Population:    in_out (valid) PersonalCopyPopulation_type;
        ]

        Conttypes:
        [
        Define Tree_type inst_of ObjectType_type
        [
                Supertype:    DataObject_type
                OwnPropDescriptors:
                [
                        Name: String;
                        Node: Node_type;
                ]
        ]

        Define Node_type inst_of ObjectType_type
        [
                Supertype:    DataObject_type
                OwnPropDescriptors:
                [
                        Name: String;
                        PopAttached: {^ OriginalIndividual_type };
                        FeatureMask: { Bool_type }[10:10];
                ]
        ]

        Define NonTermNode_type inst_of ObjectType_type
        [
                Supertype:    Node_type
                OwnPropDescriptors:
                [
                        Nodes: { Node_type };
                ]
        ]

        Define DecNode_type inst_of ObjectType_type
        [
                Supertype:    NonTermNode_type
                OwnPropDescriptors:
                [
                        DecFunc: String;
```

```
        ]
]

]

Contops:
[
Define Split inst_of OperationDescr_type
[
        Supertype:    Request_type
        OperandDescriptors:
        [
                Intree: in Tree_type
                Population: in_out (valid) OriginalPopulation_type;
                Outtree: out Tree_type;
                Outtree2: out Tree_type;
        ]
        SettingDescriptors:
        [
                Outtreenam: String;
                Outtreenam2: String;
        ]

]

Define Fisher inst_of OperationDescr_type
                        /* attaches a dec. function */
                        /* note: type of node changes! */
[
        Supertype:    Request_type
        OperandDescriptors:
        [
                Node: in_out Node_type;
        ]
]

Define Test inst_of OperationDescr_type
                        /* test the decision function */
[
        Supertype:    Request_type
        OperandDescriptors:
        [
                Pnode: in Node_type;
                Dnode: in DecNode_type;
                Newtre: out Tree_type;
        ]
```

```
        SettingDescriptors:
        [
                Newtrenam: String;
                Prior?: Bool;
        ]
]

Define: Scatter_module inst_of CreatableModule_type
[
        Supertype:    WorkspaceReq_type
        OperandDescriptors:
        [
                Node: in Node_type;
                Population: in PersonalCopyPopulation_type;
        ]
        SettingDescriptors:
        [
                Feat1: String;
                Feat2: String;
        ]
        OtherPropDescriptors:
        [
                Scatter: Scatterplot_type;
        ]
        Conttypes:
        [
        Define: Scatterplot_type inst_of ObjectType_type
        [
                Supertype:    DataObject_type
                OwnPropDescriptors:
                [
                ]
        ]

        Contops:
        [
        Define Cleave inst_of OperationDescr_type
        [
                Supertype:    Request_type
                OperandDescriptors:
                [
                        Scatter: in Scatterplot_type;
                        Newtre: out Tree_type;
                ]
                SettingDescriptors:
                [
```

```
                Son1: String;
                Son2: String;
                Newtrenam: String;
        ]
    ]

    Define Scale inst_of OperationDescr_type
    [
        Supertype:    Request_type
        OperandDescriptors:
        [
            Scatter: in_out Scatterplot_type;
        ]
        SettingDescriptors:
        [
            F1min: Real;
            F1max: Real;
            F2min: Real;
            F2max: Real;
        ]
    ]

    Define Setcod inst_of OperationDescr_type
    [
        Supertype:    Request_type
        OperandDescriptors:
        [
            Scatter: in_out Scatterplot_type;
        ]
        SettingDescriptors:
        [
            Class: String;
            Label: Char;
        ]
    ]
    ]

]

Define: Clgrow_module inst_of CreatableModule_type
[
    Supertype:    WorkspaceReq_type
    OperandDescriptors:
    [

        Node: in_out Node_type;
```

```
            ]
            OtherPropDescriptors:
            [
                Status: Status_type;
            ]

            Conttypes:
            [
            Define: Status_type inst_of ObjectType_type
            [
                Supertype:    DataObject_type
                OwnPropDescriptors:
                [
                ]
            ]

            ]

            Contops:
            [
            Define Iterate inst_of OperationDescr_type
                                /* one more iteration */
            [
                Supertype:    Request_type
                OperandDescriptors:
                [
                    Status: in_out Status_type;
                ]
            ]
            ]
        ]
        ]
]

Define Number_module inst_of CreatableModule_type
[
        Supertype:    WorkspaceReq_type
        OperandDescriptors:
        [
        ]

        Conttypes:
        [
        Define RealObject_type inst_of: SimpleAtomicObjectType_type
        [
```

```
        Supertype:    AtomicObject_type
        OwnPropDescriptors:
        [
                Value: Real;
        ]

]
]

Contops:
[
Define Add inst_of iOperationDescr_type
[
        Supertype:    Request_type
        OperandDescriptors:
        [
                Op1: in RealObject_type
                Op2: in RealObject_type
                Otp: out RealObject_type
        ]
]

Define Subtract inst_of OperationDescr_type
[
        Supertype:    Request_type
        OperandDescriptors:
        [
                Op1: in RealObject_type
                Op2: in RealObject_type
                Otp: out RealObject_type
        ]
]

Define Multiply inst_of OperationDescr_type
[
        Supertype:    Request_type
        OperandDescriptors:
        [
                Op1: in RealObject_type
                Op2: in RealObject_type
                Otp: out RealObject_type
        ]
]
```

```
        Define Divide inst_of OperationDescr_type
        [
                Supertype:    Request_type
                OperandDescriptors:
                [
                        Op1: in RealObject_type
                        Op2: in RealObject_type
                        Otp: out RealObject_type
                ]
        ]
        ]

]
```

Chapter 6

Implementation of the YANUS Object Model

In chapter 2 the YANUS object model has been described. The object model is used throughout the thesis. This chapter describes how objects, functions, methods or implementations of operations (see chapter 4) can be mapped in a relatively simple way to objects, methods and overloaded functions in the *host programming language* C++. This description is based on experience with a first implementation effort and on the formalization.

This chapter is subdivided as follows: In section 6.1, a short discussion will be given concerning the choice of C++ as the host language. In section 6.2, the implementation of generic access, which is needed in order to be able to create YANUS types at run-time, is described. In section 6.3, the correspondence between YANUS types and C++ classes is discussed. As YANUS types are created at run-time, YANUS objects cannot be created using the standard C++ mechanism, which is coupled to (compiled) classes. Therefore, the creation of YANUS objects must be generic. This is discussed in section 6.4. The generic creation of and access to objects is based on the availability of their types. However, this does not apply to meta types, since metatypes are self-describing (i.e. are their own types). Therefore, a bootstrapping mechanism is needed to create metatypes in a non-generic sense. This is described in section 6.5.

6.1 Using C++ as the host-language

The host-language must provide a sufficiently high abstraction level to enable easy implementation of objects and methods. However, the abstraction must not be at a level that the flexibility is not sufficient and/ or the data model and computational model which is provided interferes with the YANUS model.

The choice has been made for C++. Useful features of C++ are:

- C++ provides a (simple) object model offering polymorphism and inheritance.
- C++ is one of the most widely used object-oriented programming languages, programs written in C++ may therefore be easily ported.

- High levels tools for C++ programmers exist. Among these are toolkits needed to implement the User Interface (see chapter 3), but also OODBMS's, so that YANUS objects may be made persistent[1].
- Since C++ is an extension of C (it is translated into C), C software may be embedded without problems. C is suitable for programming the low level interfacing to underlying packages.

Examples of problems that may be encountered when using other programming languages are e.g.:

- Smalltalk only provides a computational model of method invocation on objects. Instead, pure function invocation (of overloaded functions) is sometimes also needed, or at least useful.
- Pascal does not provide function variables; i.e., where the function to be executed is laid down in a variable. Such a feature is very useful, specifically for executing commands.

CLOS is a language which may also qualify as a useful host programming language for YANUS. However, CLOS was not available at the time of inception of this project.

6.2 Modelling YANUS objects and properties

In C++, two standard mechanisms are used to access an object's state:

1. Using a "public" member variable which may be directly read and set, just like members of structures may be accessed in the underlying programming language C. This mechanism is in general too primitive to provide access to YANUS objects. It does, for example, not allow the incorporation of additional actions in the access (e.g. doing run-time type checking).
2. Using hidden member variables together with public methods (methods correspond to member functions in C++) that hide the way the state is changed or read. Access on a certain part of the state, either for reading or writing, must specifically be defined and implemented as a method on the C++ class. These methods will be referred to as *specific access* methods.

[1] No OODBMS has been employed yet though, neither has the use of a OODBMS been specifically studied.

In modelling the access to a YANUS object, a distinction is made between the access to the various properties[2] and to other structural elements, such as its type and validity. The access to these other structural elements can be modelled using specific access.

Read and write access to YANUS properties is implicitly defined as a result of the definition of a property in a YANUS type. Furthermore, YANUS types are definable at run-time, i.e., no recompilation of the system is needed as a result of a type definition.

Access[3] to properties is therefore implemented generically, i.e., type and property independent. This is done in the following way:
An object's value, which gives access to the properties (see chapter 2) is modelled as an array. Each position in the array holds a property[4] (or more formally: a property value). To access a certain property, the corresponding position or *offset* must be known. This offset is a specifically accessible member variable of the corresponding property descriptor. Two forms of generic access can be used:

- Via the type of the object and the name of the property, access is given to the property descriptor and thus to the offset. Some form of optimization is used (e.g. a hash table contained by the type) to map the name of the property to the property descriptor.
- As described in chapter 3, in many cases the property descriptor is given and the offset can be accessed through it.

In the bootstrapping phase, objects are created for which no types exist as yet. In that case fixed and pre-defined offsets are used. This will be detailed in section 6.5.

6.3 The correspondence between YANUS types and C++ classes

1. For certain types of YANUS objects, specific access methods are applicable, such as:
   - for all objects: access to the type, validity and value
   - for all object types: access to the hash table
   - for all property descriptors: access to the offset.

---

[2] This includes all types of properties such as relationship properties, reference properties, attributes, properties which hold operands in a request etc.

[3] "Access" encompasses both retrieval and storage. On top of this access the setting of back references etc. (see chapter 2) is implemented.

[4] This is a sequence. Access to the elements in such a sequence is not further discussed here.

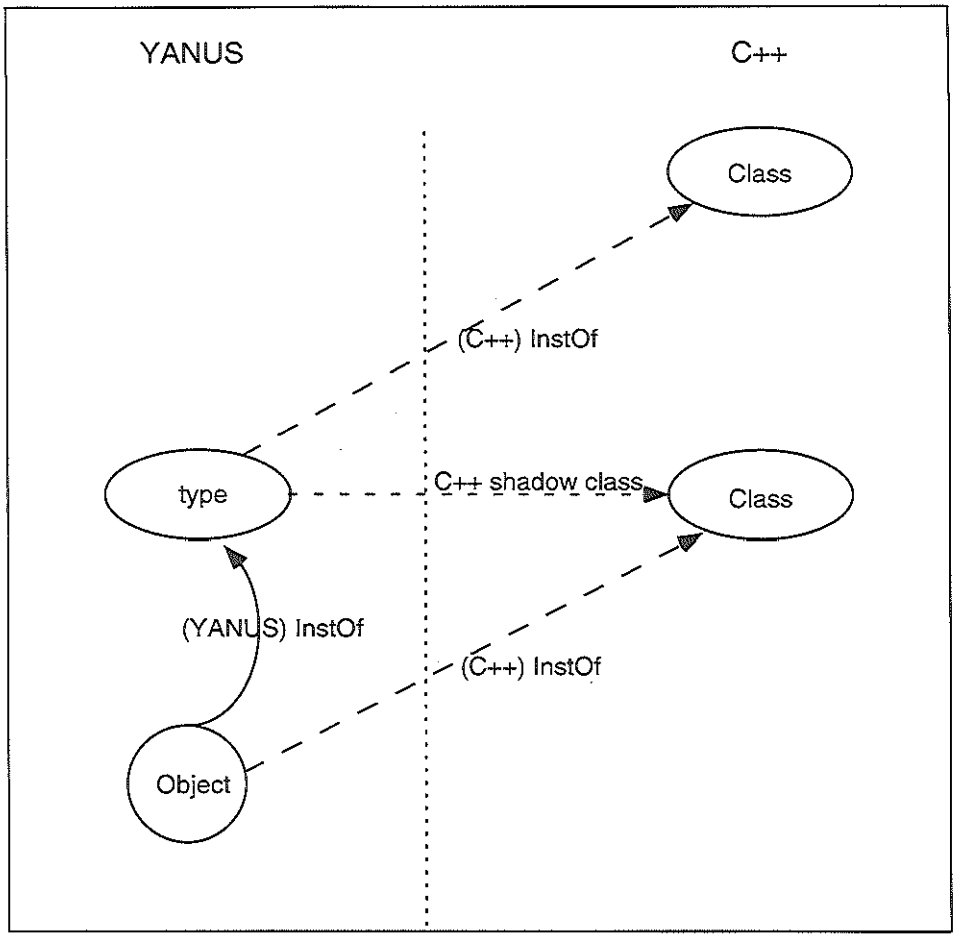

Fig. 6.1

Relationship between YANUS objects, YANUS type objects and C++ classes. Each YANUS type corresponds directly or indirectly to a shadowclass, so that all YANUS instances of the type are C++ instances of the shadowclass.

2. In chapters 2, 3, and 4 different methods such as New, Execute etc were defined. These were defined formally by their signature on the basis of sets. E.g. Execute is applicable to objects in the set IRO. Although such a set is defined as the set of all instances of a certain YANUS type -for IRO this is the YANUS type "Request_type"-, no mechanism was described indicating how such a method could be implemented and linked to an object of that type, or how such a method could be invoked.

In this chapter, the mechanism is introduced to equate the set of all instances of a certain YANUS type to a class, **so that all instances of a certain YANUS type are also instance of that class** (see fig. 6.1). This class is called the shadow class of the YANUS type. On this class all specific (access) methods as mentioned above are defined and implemented. On these classes methods and functions may also be implemented that serve as implementations of conceptual operations (see chapter 4). It should be noted that in the implementation of the object model, the C++ class completely takes the role of the VDM set as used in the formalization. This is also the case in the bootstrapping phase, see section 6.5.

The following naming convention is used: If a YANUS type of the name XXX_type has a shadowclass, this shadowclass is called XXXSC. Thus, objects which are YANUS instances of the type XXX_type, are in C++ instances of XXXSC. See figures 6.1 and 6.2.

The majority of YANUS types do not correspond directly to a C++ class, but have a supertype (directly or indirectly) which corresponds directly to a shadowclass. All such YANUS types are also said to correspond to that shadowclass. YANUS instances of all types that correspond to a certain shadowclass are all C++ instances of that same class.

As a result, there is also a uniformity between the YANUS type- and C++ hierarchy of shadow classes. If a YANUS type A is (directly or indirectly) a subtype of a YANUS type B, it corresponds to the class SC to which B corresponds, or to a class SC' which is a subclass (see fig 6.2).

6.4 (Generic) Creation of YANUS objects

In C++, class specific constructor methods are used to generate instantiations of that class. Such a constructor method allocates memory for the member variables and also adds a class specific "table" of applicable methods.

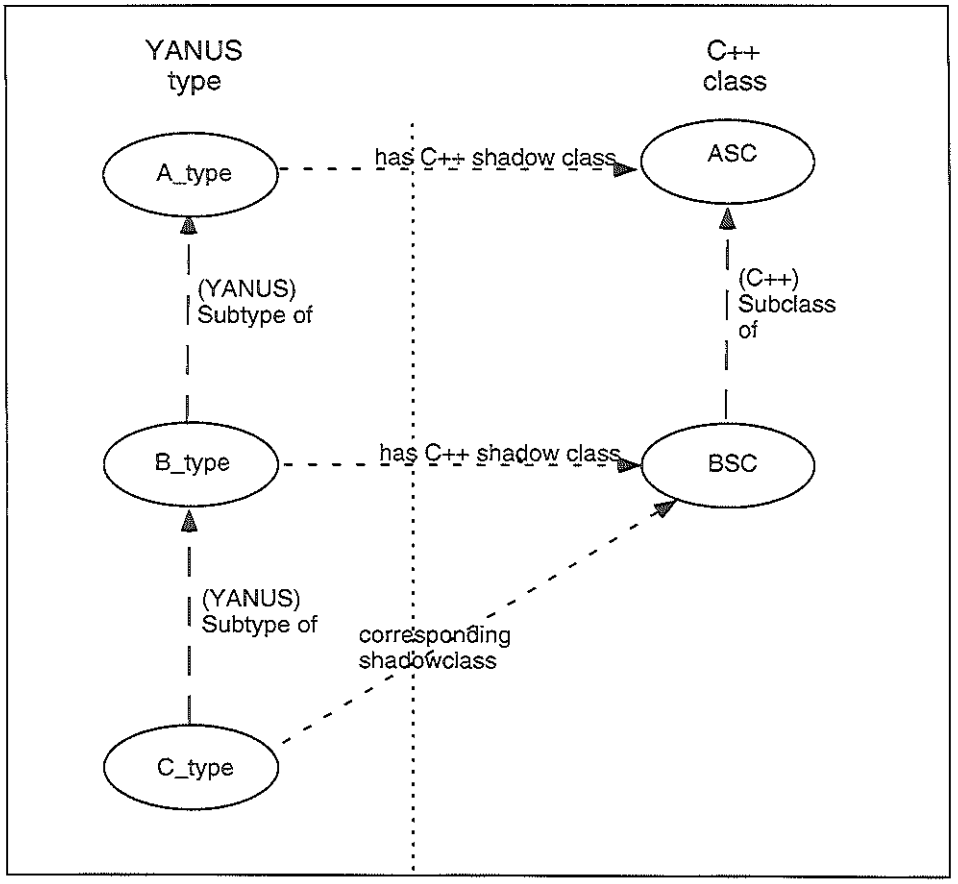

Fig. 6.2

Relationship between the YANUS type hierarchy and the C++ class hierarchy. If YANUS type B_type is a subtype of type A_type, and B_type and A_type correspond to shadowclasses BSC and ASC respectively, than BSC must be a subclass of ASC (or BSC and ASC are identical). C_type does not have a shadowclass itself. C_type is still said to correspond with the shadowclass BSC.

For the construction of YANUS objects, an array of the appropriate size to hold the properties (the value of the object) must also be allocated. Thus, the construction of a YANUS object may be viewed to be subdivided in the construction of the C++ part and in the construction of the value part. However, since that object construction must be generic, for each (YANUS) type an object of that type must be constructed using one single function. Moreover, the corresponding C++ class and thus the specific constructor which must be used cannot be derived from the YANUS type.

Therefore, the "New" operation as used is different from the one formally described in chapter 2: Linked to a type (using either a property or a C++ member variable) is a template for the C++ part of its instance. This template has been constructed using a specific constructor during the bootstrap phase. Each time a new object of that type is created, the C++ template is copied. In addition, the object value array is created generically, using the information in the type describing the complete set of property descriptors. Note, that a template and copy mechanism is not used to create the object value array. This allows the complete set of properties to be different than during the boot phase, see section 6.5.

6.5 Bootstrapping the YANUS kernel

When the YANUS system is started, no meta type hierarchy exists. In that situation types cannot be created using the mechanism described above, neither can meta types be created, since meta types are their own instances.

In the bootstrap phase, the C++ shadow classes ObjectTypeSC, StructPropDescrSC and BasicValueTypeSC play a central role, similar as corresponding VDM sets play a central role in the beginning of the formalization. The basic idea is, that specific methods for creation and access of the value part of the objects may be used during the bootstrap phase. These are implemented on the shadow classes mentioned above: For each such shadow class the assumption is made that the corresponding YANUS object has a specific initial set of properties. In general this will be a subset of the set of properties which objects of the corresponding YANUS type will really have, to minimize the amount of specific software that must be written. In case of StructPropDescrSC this may for example be "Name", "Supertype", "EltType", "MinElt", "MaxElt". The corresponding specific constructor used during the bootstrap phase creates a value array with the corresponding size. In the case of StructPropDescrSC, given the initial set of the five properties described above, this size is five times the memory unit to hold a property value. Specific access methods are defined on the class through which these properties can be read and written using a pre-defined constant offset.

Using these methods defined on the classes, the first version of the meta type hierarchy may be created and filled in. Creating a complete meta type hierarchy, as for example shown in diagram 2.1., involves among others that for each meta type the complete set of property descriptors is derived, that each type refers to its C++ template (as mentioned in section 6.4) and that each meta type (including "MetaType_type") refers to its type (which is "MetaType_type"). Using such a complete hierarchy, generic creation and access of instances is now possible. However, due to the fact that the initial set of properties as supported by the shadow class may (and will in general[5]) be smaller than the set of properties which are prescribed by this meta type hierarchy itself, a similar hierarchy must be generated using generic object creation and access, so that the objects in the hierarchy do have all the properties that the hierarchy itself prescribes. For example, the structural property descriptors of the first version of the meta type hierarchy will not have the properties "Changeable" and "Unique", although these properties are prescribed by the meta type "StructPropDescr_type". When creating a next version of the hierarchy, using generic mechanisms, the property descriptors will have these properties, and these properties must also be filled in.

6.6 Discussion, Conclusion

It may be concluded that it is relatively simple to map YANUS objects and functions onto C++ objects and methods, respectively. In this way, YANUS can "tap" the power of C++; YANUS objects are also C++ objects and thus incorporation of C++ tools such as toolkits and databases is possible.

C++ also imposes certain restrictions, such as single inheritance and a fixed method invocation mechanism. It may well be possible to either build YANUS on top of a more flexible object model such as CLOS, or on top of a language such as C, so that YANUS must provide its own method inheritance and invocation mechanism. The latter alternative does not seem to be attractive (although it may be a theoretically interesting venture) due to the extra effort this requires, and because the high level tools provided for C++ which cannot be used in that setting. The former alternative seems to be an interesting possibility, which warrants further study.

[5] It is both impossible and not desirable to make the set of implemented properties in the boot phase complete. It must be possible to introduce properties not foreseen in the implementation of the bootstrapping phase on one of the YANUS types which correspond to classes that play a role in that phase.

Chapter 7

Discussion, Conclusion

This thesis considers the integration of data and operation resources with the aim of creating a better information processing environment for the user. Such an environment could only be achieved by providing the appropriate tools to some implementor to create a specific integration and to extend it. Providing those tools is another major purpose of this work. In fact, YANUS is a development environment for resource integration. It proved even to be useful to allow explicit modelling through meta types of the kind of tools the integration implementor is allowed to use. Meta types are powerful tools for the *system designer*.

In this chapter the system will be discussed from three points of view:

1. The user
2. The integration implementor
3. The system designer.

This chapter also revisits some of the most important literature, and compares the approach described in this thesis to existing approaches.

7.1 The viewpoint of the user

This thesis proposes an encompassing environment which can accommodate many different resources, so that they can be optimally combined by the user. Combining resources is based on typing, i.e. classification by type (see also chapter 3). The type of a data object symbolizes what it represents. It incorporates a description of the structure of the data and the applicable operations. Operations are characterized by the type of data objects which can be used as parameters.

The use of resources thus combined is uniform. Changing structure, or applying operations is achieved through generic (type independent) dialogues.

The flexibility of the integration, as achieved by the appropriate development tools, is useful for the user: It makes it possible to add new resources to the integration. Due to the principle of combining resources on the basis of typing, newly introduced resources are compatible with older ones when the types match: Previously defined operations may be applied to newly added objects, as long as the type of objects is appropriate. Similarly, newly defined operations may be applied, when appropriate, to objects of earlier defined types.

One of the issues in the approach of combining resources on the basis of typing is, that the user may want an object to represent concepts at various abstraction levels, and/or he may want to vary the way of looking at a certain object. For example, a graph may be manipulated as a graph -e.g. it may be smoothed, the scaling may be varied etc.-, but the user may also want to use a graph as a graphical figure, to apply colouring, create shadows, depth etc. Our approach does not allow an object to be treated as two different types at the same time, but the user may be provided with an operation to for instance, create a graphical figure from the graph, not by changing the object's type (which is not allowed) but by creating a new object with a different type.

Looking at the same object in different ways is related but not quite similar: When designing a house, one designer may want to consider mainly insulation, another designer may want to consider the electrical connections. Two such different ways of looking at the same object may correspond with a different structural decomposition of the same object (Gielingh 1987). This may be partly incorporated in the present approach by setting up different types of workspaces through which different parts of the same object may be inspected. An example of this is encountered in the direct application area, where a population may be viewed as a table, but also as a set of population trees (chapter 6). The fact that the system provides a snapshot (a kind of copy) of the data resources to the user allows him, in principle, to let the snapshot be a transformation of the data, and thus makes it possible to use various such transformations in different types of workspaces. Offering such facilities may be an interesting subject of further study and extension of the model.

In chapter 3 it has been described, how the user may be supported by indicating applicable operations given an object, or indicating objects that may be assigned given a property, or objects that may be used as input given a certain parameter of an operation etc. Thus support is given to use and combine the resources conform the typing scheme. Clearly, certain application areas will require the use of more extensive support, i.e. on a semantically higher level. The best examples are games, where a move may be correct, but not necessarily smart. Building such support on top of the standard support described above, e.g. by using smart agents that can be asked for advice, will be an interesting subject for further study.

7.2 The viewpoint of the integration implementor; YANUS as a "development environment"

To the integration implementor the genericity of the YANUS model is of importance, i.e., the fact that the dynamics of the system adapts itself to the various specifications including conceptual and implementation schemas. As a result, extending the integration only involves extending those specifications, and in principle no new software needs to be written. One of the dangers of this approach is inflexibility, due to the limitations of what can possibly be specified. This may result in restricted application possibilities. This danger has been tackled in three ways:

- Specifications of interfaces are split up into various levels of abstraction, so that at the lowest levels certain details may be specified, e.g. the output syntax of the software package. Note, however, that in this thesis not much attention has been given to such lower level specifications.
- A lower level specification such as the driving of a package through commands and the parsing of the output can be bypassed using software in the host-language C++. Whether the same sort of bypass may be used in user interface specification/ implementation has not been studied.
- The use of meta types allows the adaption and extension of the specification languages.

In chapter 1 it was indicated that the main goal of this research is to enable the application of data analysis to data resources as available from DBMS's. In chapter 2 and 3, the manipulation and editing of schema information was added to this. As a result, quite complex types of data and corresponding operations may be modelled. In retrospect, the application of the YANUS model does not seem restricted to the focus mentioned in chapter 1. Moreover, the area of application is principally extensible due to the flexibility given by the meta type level. However, YANUS is not an appropriate tool for modelling scenario's where the user has to perform certain operations in sequence, i.e., in which the system has to maintain a state other than the one embedded in the data presented.

It is certainly of relevance and interest to acquire experience in the application of the system to various forms of integration. This is especially so since the direct application area, as described in chapter 5, has mainly been a theoretical exercise.

7.3 The viewpoint of the system designer; meta types

As mentioned, the modelling "power" of the system designer is based on meta types, and specifically on the use of primal types. Through the primal type, coupled to each meta type, the system designer can define behaviour (functions and operations) and properties, which are inherited by all types which are instances of that meta type and therefore subtypes of the primal type. Specifically, workspaces were added at a late stage to the model, which proved the importance of the meta type concept.

Flexibility for the system designer is not the only advantage of the use of meta types. A recapitulation:

- The use of meta types allows types (or other parts of the schema, such as operation descriptions) to be accessed as objects. Thus, schema information can be used at run-time to determine the system's behaviour. This is the basis for the genericity of the model. It is employed in supporting the user to use resources conform to the typing scheme (see chapter 3). It is also employed in executing operation requests (see chapter 4).
- In the same way as a schema (or: information at the type level) is used to specify and control the creation, editing and manipulation through operations of objects, information at the meta type level may be used to specify and control the creation and manipulation of schemas.

Genericity may not always be an optimal solution, since it uses type information interpretatively, and may thus be slow. For certain aspects of the YANUS model, specifically for executing requests, it would be interesting to investigate the possibility to generate specific methods for executing each conceptual operation. Such specific methods may again be generated by means of generic methods within the context of the described data model.

It is of interest to further study the application of meta types for computational reflection (Maes 1987). This would for instance allow one to incorporate both merging of operation with data resources and merging of data with data resources in order to combine overlapping data as described by Schrefl (Schrefl and Neuhold 1988).

7.4 Other approaches

MW2000 is another approach which provides integration of data and operation resources. It also has the possibility to drive underlying existing software packages and to provide for data

translations between those packages. While the YANUS approach is directed at offering to the user many different kinds of resources, which may be used in optimal combination (see chapter 1), MW2000 is focused on providing an integration of (relational) database data- and statistical software analysis operation resources which has been worked out and implemented in depth. In combination with its aim at providing a multitude of resources, YANUS is more strongly directed at extensibility, due to the principle of genericity. Also, YANUS has a stronger focus on a highly interactive "exploratory" use of its data, which is both incorporated in the user interface and in the driving of the underlying package, the latter allowing for maintaining a kind of dialogue with the underlying package. A strong feature of MW2000 is its support for client server architectures, which has not been specifically studied in this project.

Apart from MW2000, many different other integration approaches exist, but in those other approaches the concept of "integration" may have a different interpretation.

In certain projects integration is interpreted as integration of resources to the (application) programmer (Nilsson, Nordhagen et al. 1990; Hornick, Morrison et al. 1991; OMG 1991). This leads to an application oriented model of information processing to the user, even if applications are better tuned to the tasks of the user (Nilsson, Nordhagen et al. 1990). However, an important subject which is accomplished by these approaches, and not modelled here is cooperation between distributed systems and thus potentially between users. This does not mean that this kind of interoperability cannot be incorporated in principle in the YANUS model. This aspect, however, is outside the scope of this thesis.

A future development of so-called "Document Oriented Interfaces" (Lu 1992) provides an integrated view of resources to the user, and thus abolishes the application oriented model. Such an integrated view, however, requires cooperation between all tools and all objects (which may be viewed as small applications in their own right), while YANUS is the pivot in driving underlying resources. This difference may be compared with the difference of needing translations between all of n different languages or using one intermediate language. The YANUS approach does not require standards, although the availability of standards would simplify the implementation of software package managers.

Integration of heterogeneous databases is another important topic in literature (Schrefl and Neuhold 1988; Bertino, Negri et al. 1989; Cannata 1991). Such integration is not directed at abolishing the application oriented model. Still there are ways to compare those approaches and the YANUS approach, and here similarities may be found. Both forms of integration are directed at integration of resources at a high conceptual level. Database integration is focused on integrating data resources while YANUS focuses on the integration of data resources with

operation resources. In database integration, underlying databases are united into one. In YANUS separate data resources will remain separate at the conceptual level, although they are used -also in combination- within one environment. Both forms of integration employ data modelling as the way to express the integration. Even here, similarities may be found: while an object as described by (Schrefl and Neuhold 1988) is a generalization of underlying objects corresponding to data in underlying databases, a conceptual object in YANUS is a kind of union of all the implementation objects it may possibly have. Note that in YANUS the use of data modelling is extended to control the user interface.

Literature

Bertino, E., M. Negri, et al. (1989). "Integration of Heterogeneous Database Applications through an Object-Oriented Interface." Information Systems **14**(5): 407.

Cannata, P. E. (1991). The Irresistible Move towards Interoperable Database Systems. First International Workshop on Interoperability in Multidatabase Systems, Kyoto, Japan, IEEE Computer Society Press.

Gielingh, W. F. (1987). General Reference Model for AEC Product Definition Data (version 3). IBBC-TNO.

Hornick, M. F., J. D. Morrison, et al. (1991). Integrating Heterogenous, Autonomous, Distributed Applications using the DOM Prototype. GTE Laboratories.

Lu, C. (1992). "Objects for End Users." BYTE **17**(14): 143.

Maes, P. (1987). Concepts and Experiments in Computation Reflection. Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), Addison Wesley.

Nilsson, E. G., E. K. Nordhagen, et al. (1990). Aspects of System Integration. The First International Conference on Systems Integration (ICSI), Morristown, New Jersey, IEEE Computer Society Press.

OMG (1991). The Common Object Request Broker: Architecture an Specification (CORBA). Object Management Group, Inc.
Framingham, Massachusetts.

Schrefl, M. and E. J. Neuhold (1988). Object Class Definition by Generalization Using Upward Inheritance. 4th International Conference on Data Engineering, Los Angelos, IEEE Computer Society Press.

Summary

1. Towards an integrated user environment

The computer may be said to provide certain "resources" to the user. In general, these resources can be subdivided into data resources and operation resources[1]. At the moment, an increasing number of resources is becoming available. This is mainly due to the growing of networks, and to the fact that many companies are starting to offer their services through such networks, which will be further stimulated by new technological advances such as the introduction of ISDN.

Resources are generally offered to the user by means of applications. An application provides access to certain data and to specific operations applicable to those data. A normal computer environment is thus a so-called "Application oriented" environment. Unfortunately, this architecture restricts the possibilities to use all resources available on the computer in combination with each other.

At the moment, much attention is given to architectures that diminish these shortcomings by allowing for communication between applications. By these means, the user gets the possibility to "nest" data from different applications -called "object embedding"- for example a drawing from a drawing application may be nested within a text from a text processing application. This is done in such a way, that the combination may be presented on the screen, and both parts may directly be edited. Also, the user may link data from different applications, such that they depend on each other dynamically - this is called "object linking" -.

There is, however, a third form of combination of resources, namely the combination of operation and data resources. This form of combination becomes specifically of interest, if either data describe the same phenomena in the real world, or similar kinds of data are used within different applications. For example, the user would like to use the *name* of a person with whom he made an appointment in the electronic agenda to search the *address* in a database; he would like to use the *address* to look it up in a map/routing application. He would also like to use *time*, *day* and *address* to plan his trip by public transport, and possibly also to order some flowers which should be delivered then and there.

[1] In certain situations the programmed code for a certain operation may be viewed as "data". In that situation other operations are again provided to manipulate those data.

To allow for such combined use of data and operation resources in a user-friendly fashion, it is not sufficient to have standards which support the copying of data between applications. Firstly, the application cannot help the user in copying the appropriate data for a certain purpose, since it cannot "know" that purpose. Secondly, the user does not want to manage several copies of the same data. The latter becomes specifically a burden when the copied data may be changed or stored by the other application. Database data are for example often stored in separate files for the purpose of statistical analysis. These separately stored copies may lead to inconsistencies.

The combined use of data and operation resources has been a major focus of the study treated in this thesis. A general[2] model[3] of a system (called YANUS[4]) for presenting resources to the user is described. Resources are offered in such a way that operations or modules - corresponding to groups of operations - can always be combined with data resources (modelled as data "objects") by either directly **applying** them to data objects, and/or by merely **selecting** data objects as input. Copying is never necessary. The system can help the user in indicating which combinations make sense.

YANUS is a graphic interactive front-end; it integrates resources by serving as a hub for transport of data from its source (and generally: its storage) to all different operation implementations (see figure 1.1). YANUS is both an environment in which **many different** types of data and operation resources may be presented to the user, and it allows for **an easy extension with new types of resources**. This includes ease of extension of the user interface. In other words: YANUS is principally application independent and extensible. YANUS may **encompass many different resources within one environment**. From this point of view, YANUS could be compared to a graphical shell on top of the operating system, where YANUS does not only encompass the directory and file level but also the levels which are normally separately managed in different applications.

Special attention has been given to integrating ("encapsulating") existing data and operation resources, again focusing on application independency and extensibility. The encapsulation by YANUS of underlying software encompasses copying data between resources and bringing them in the right representation, and managing these different copies. This is hidden from the user: to him, there is only one data object. Furthermore, the underlying software is driven, if

---

[2] The scope restrictions are outlined in the thesis

[3] The model has been formally described in VDM. For certain parts of the model experience in the implementation has been acquired.

[4] YANUS stands for Yet ANother Unified System

necessary using key strokes, or through commands. Its output data are again incorporated as manipulatable objects in the environment. An example has been worked out in the thesis which concerns the integration of a relational DBMS ("Ingres") and a software package for interactive statistical pattern recognition ("ISPAHAN"), allowing the pattern recognition operations to be applied directly to the database data. A special problem encountered in this example is the fact that ISPAHAN cannot be driven by means of a script in some programming language; it is an interactive menu driven package in which the user explores his data, making direct use of the results presented to him. This interactivity is also provided by the interface of YANUS, which must thus maintain a dialogue with both the user and the underlying package.

2. Data model technology as basis for YANUS

The technological basis for YANUS is its data model. The data model has both aspects of object-oriented and semantic data models.

First, we shall view the data model from its structural (semantic) aspects.

Objects are the principal entities in the data model. Objects have properties. These properties may either refer to other objects or contain values such as numbers, strings etc. Which properties an object has, and the values or references which may be contained by each of the properties is prescribed by the objects' type. The object is said to be an instance of its type.

The user of YANUS may apply operations to objects. To this end, so-called operation descriptions are modelled in the data model. Operation descriptions are a special kind of types. An instance of an operation description is a request (for that operation); the properties of such a request correspond to (are) the parameters of the operation. The user may submit a request for execution, when all properties have been filled in correctly.

Essential for the model is that types (and thus also operation descriptions) are themselves represented as objects, so that all information about the types is run- time accessible. For example, what properties instances of that type may have, the values or type of objects that may be referenced by each of these properties is explicitly represented by property descriptors. Explicitly represented is also what operations may be applied to the instances of a certain type, by means of references to the corresponding operation descriptions.

This explicit representation of types as objects plays an important role in the implementation of the dynamics of the system. A major part of the dynamics of the system can namely be implemented generically, i.e. as type independent software. Type dependencies in the

dynamics are taken care of by interpreting the type information. For example, objects can be generated generically: the memory allocated for a new object is dependent on the set of property descriptors of the type. Thus, one routine (in fact: one "method", as we shall see) creates instances in this way for all different types. This is of course a well known example[5]; the principle is much more widely applied in YANUS, as shall be seen.

The dynamics in YANUS is implemented in the form of methods. Methods are defined on types and applicable to their instances. Note that in contrast to operations, the user is not informed about the existence of these methods. As in "classical" object-oriented systems, methods are inherited through a subtype/supertype type hierarchy, so that all methods applicable to instances of the supertype are also applicable to instances of the subtype.

To allow for the implementation of **generic** methods, two important concepts are introduced: meta types and primal types.

Meta types are types of types, that is, they describe the structure of types in the same way as a type describes the structure of its instances. Thus, through the meta type, the structure of a type (as being an object) is determined.

Methods can be generic, if they use the information in the type, and thus the type's structure. Since the structure of the type is determined by the meta type, all types with the same meta type can have the same generic methods. This is realized by letting all types which are instances of the same meta type be subtype of one same "primal" type, which is associated with the meta type. In other words: the primal type is the root of the subtype/supertype hierarchy of all types which are instances of the same meta type, and all these types inherit generic methods from the primal type.

By means of meta types and their corresponding primal types, a differentiation between groups of types (instances of different meta types) is introduced. For example, all operation descriptions have a specific meta type, different from the meta type of all other types. All operation descriptions inherit the generic method "Execute" from the primal type associated with this meta type; this method is thus used to execute a request for any operation of the user.

[5] And it can be done at compile time. In fact, instead of using generic methods to implement the dynamics, another possibility would be to use generic methods that generate type specific methods that implement the dynamics. This has not yet been studied.

Genericity is used by a generic piece of software called the "dialogue manager" to conduct the user interface dialogue of YANUS. For example, the links between operation descriptions and (normal) types are used by the dialogue manager to guide the user in correctly applying operations to objects. Also, editing of objects is supported generically by using the information in the property descriptors of the type. A "Workspace" is another example of a generically supported user interface feature. It is a special context (similar to a window) in which the user can edit certain data and apply operations to them. The workspace concept is introduced by means of a special meta type the instances of which are so-called modules. Workspaces are instances of modules. Workspaces can be opened and closed using generic methods.

Thus as a result of the explicit representation of types, each of the presented resources has its own knowledge linked to it, describing its own use. The dialogue manager interprets this knowledge. This makes it possible to support the use, at a semantic level, of a theoretically unbounded and dynamic set of differently typed objects, and of different operations and modules. In standard dialogue management where the knowledge about types and operations etc. is basically "hardcoded" into the dialogue manager, this would be practically impossible, specifically given the requirement that new resources should be easily added, so that it may immediately be used (in conformance with its type) in combination with all other resources.

This extensibility is thus an essential aspect of the user interface approach described in the thesis: New resources can simply be plugged in, by describing them in terms of types, operations, etc. The dynamics is completely pre-defined and does not need to be extended. Next to describing the resources in this way, a description of the presentation must also be given. How this should be done has not been investigated.

In the generic user interface mechanism described above, types are used to specify the use in the user interface of their instances. Meta types can be used in this same way, that is, to specify the manipulation and editing of their instances in the user interface. As a result, the whole type structure (including operation descriptions, modules etc.), also called the schema, can be extended through the system's own user interface.

Genericity is also used by the method "Execute" to execute requests using underlying resources. Similar as types of objects and operation descriptions used at the user or conceptual level are described in a so-called conceptual schema, types of objects and operations are described, which correspond to specific data representations and specific operation implementations in underlying software. This is done in so-called implementation schemas. The structural coupling between the conceptual schema and implementation schemas indicates the appropriate implementations for a conceptual operation. Furthermore, it indicates the representations which must be generated for the various parameters etc. in order to use a

certain operation implementation. The "Execute" method interprets this information. Again, this approach allows for ease of specification and extension of the integrated resources due to pre-defined dynamics.

Finally, it should be mentioned, that meta types allow for the introduction of completely new features in the system at a fundamental level. Modules are an example of this possibility. The introduction of new meta types is again supported by an even higher level type, the type of all meta types. This metatype-type is its own instance. The data model is thus self-descriptive.

Samenvatting

1. Op weg naar een geïntegreerde gebruikersomgeving

Men zou kunnen zeggen dat de computer bepaalde "resources" (Nederlands: hulpbronnen) aan de gebruiker aanbiedt. Deze resources kunnen worden onderverdeeld in data-, en operatie-resources[1]. Op het ogenblik neemt het aantal aangeboden resources sterk toe. Dit komt met name door de proliferatie van netwerken, en ook doordat steeds meer bedrijven hun diensten via zulke netwerken aanbieden. Dit laatste zal verder gestimuleerd worden door nieuwe technologische ontwikkelingen zoals ISDN.

Resources worden in het algemeen door middel van applicaties aan de gebruiker aangeboden. Een applicatie biedt toegang tot bepaalde data en tot specifieke operaties die op die data toepasbaar zijn. De meeste computeromgevingen zijn derhalve "applicatie-geörienteerd". Helaas beperkt deze architectuur de mogelijkheden om alle resources die door de computer worden aangeboden met elkaar te combineren.

Op het moment wordt veel aandacht geschonken aan architecturen die dit nadeel (gedeeltelijk) opheffen, door communicatie tussen applicaties technisch mogelijk te maken. Ten gevolge hiervan kan het voor de gebruiker mogelijk worden gemaakt om data uit verschillende applicaties ten opzichte van elkaar te nesten. Een tekening uit een tekenapplicatie kan bijvoorbeeld genest worden in een tekst van een tekstverwerkingsapplicatie. Dit kan op een zodanige manier dat de combinatie van tekst en tekening op het scherm als geheel gepresenteerd kan worden en beide nog apart te wijzigen zijn met behulp van de applicatie. Dit wordt in het vakjargon ook wel "object embedding" genoemd. Ook kan de gebruiker data uit verschillende applicaties met elkaar koppelen, zodanig dat deze verschillende data dynamisch van elkaar kunnen afhangen. Dit wordt ook wel "object linking" genoemd.

Er is echter nog een derde vorm van combinatie van resources, namelijk de combinatie van operatie- en dataresources. Deze vorm van combinatie wordt dán interessant, indien data uit verschillende applicaties dezelfde fenomenen in de buitenwereld beschrijven, of indien verschillende soorten applicaties gelijksoortige data gebruiken. De gebruiker zou bijvoorbeeld een *naam*, verkregen bij het maken van een afspraak tussen twee elektronische agenda's, willen gebruiken voor het opzoeken van een *adres* in een database; het *adres* willen gebruiken voor het zoeken in een kaartenapplicate; het *tijdstip*, de *dag* en het *adres* willen gebruiken

[1] In bepaalde situaties kan de geprogrammeerde code behorende bij een bepaalde operatie gezien worden als "data". In die situatie worden echter weer andere operaties aangeboden om die data te manipuleren.

voor het plannen van de route per openbaar vervoer, en misschien ook voor het bestellen van een bos bloemen die dan en daar bezorgd moet worden.

Om een dergelijke combinatie van data en operaties op een gebruikersvriendelijke wijze (technisch) mogelijk te maken is .het niet voldoende om standaarden te hebben die het kopiëren van data tussen applicaties ondersteunen. Ten eerste kan de applicatie de gebruiker niet helpen bij het kopiëren van de juiste data voor een bepaald doel, aangezien de applicatie dat doel niet kan "kennen". Ten tweede wil de gebruiker niet verschillende kopieën van de zelfde data hoeven te onderhouden. Dit laatste wordt met name lastig als kopieën gewijzigd en/of opgeslagen kunnen worden door verschillende applicaties. Data uit databases worden bijvoorbeeld vaak opgeslagen in aparte files ten behoeve van statistische analyse. Dit kan tot inconsistenties leiden.

Het gecombineerde gebruik van data- en operatie-resources is een belangrijk doel van dit proefschrift. Een algemeen[2] model[3] van een systeem, YANUS[4] geheten, dat resources aan de gebruiker aanbiedt wordt beschreven. Operaties of modules[5] kunnen in YANUS gecombineerd worden met data-resources (gemodelleerd als data-objecten) door ze direct toe te passen op data-objecten, of door data-objecten direct te selecteren als input voor een operatie of module. Het kopiëren van data is, voor de gebruiker, nooit nodig. Het systeem kan de gebruiker helpen door aan te geven welke combinaties zinnig zijn.

YANUS is een grafisch interactief "frontend"; het integreert resources door als een overslag-station te fungeren: het transporteert data van z'n bron (in het algemeen: de opslag) naar de verschillende operatie-implementaties (zie fig. 1.1). YANUS is zowel een omgeving waarin **verschillende types** van data en verschillende operatie-resources aan de gebruiker gepresenteerd mogen worden, als een omgeving waarin een **nieuw type resource eenvoudig kan worden toegevoegd**. Dit omvat de bijbehorende uitbreiding van de user interface. Dus, in andere woorden: YANUS is principeel toepassing-onafhankelijk en uitbreidbaar. YANUS **biedt alle resources binnen één omgeving aan**. Het kan vergeleken worden met een grafische shell bovenop het operating system, waarbij YANUS echter niet alleen de directories en files

---

[2] Beperkingen worden in het proefschrift aangegeven.

[3] Het model is formeel gespecificeerd in VDM. Voor bepaalde delen van het model is ervaring in de implementatie verkregen.

[4] YANUS staat voor Yet ANother Unified System

[5] Een module is een gegeneraliseerde operatie, die bij toepassing op een data-object, toegang geeft tot de inhoud van dat object en tot een groep van operaties die specifiek bruikbaar zijn voor het manipuleren van die inhoud.

en de bijbehorende programma's (d.w.z. de operaties op dat niveau) kan opstarten, maar ook de lagere datastructuren en operaties aanbiedt die normaliter in aparte applicaties worden aangeboden.

Speciale aandacht in het proefschrift wordt gegeven aan het integreren ("encapsuleren") van **bestaande** data- en operatie-resources, opnieuw gericht op toepassing-onafhankelijkheid en uibreidbaarheid. Om bestaande applicaties te encapsuleren binnen de eerder beschreven frontend moet YANUS het kopiëren van data verzorgen en het daarbij omzetten naar de juiste datarepresentatie, zodat een bepaald pakket de invoer-data in de goede vorm krijgt aangeboden. YANUS moet ook de verschillende kopieën beheren. Dit alles onzichtbaar voor de gebruiker: voor hem corresponderen de verschillende kopieën met één conceptueel data object. YANUS moet tevens de onderliggende pakketten aansturen, zonodig via gesimuleerde toetsaanslagen of via commando's. De uitvoer van een pakket wordt weer geïncorporeerd als door de gebruiker manipuleerbare objecten. Een voorbeeld is uitgewerkt in het proefschrift betreffende de integratie van een relationele DBMS ("Ingres") en een software-pakket voor interactieve statistische patroonherkenning ("ISPAHAN"). De frontend moet de directe toepassing van patroonherkenning op database data mogelijk maken. Een speciaal probleem dat wordt tegengekomen in dit voorbeeld is de interactieve menugestuurde aansturing van ISPAHAN. ISPAHAN wordt exploratief gebruikt: de gebruiker wil telkens aan de hand van tot dan toe verkregen resultaten z'n volgende stappen bepalen etc. Dit betekent dat YANUS een dergelijke interactieve dialoog met de gebruiker moet kunnen houden, en tevens met het onderliggende pakket.

2. Datamodel technologie als basis voor YANUS

De technische basis voor YANUS is het datamodel. Het datamodel heeft zowel aspecten van object-geörienteerde en semantische datamodellen.

Eerst zullen we het datamodel van z`n structurele (semantische) kant beschouwen.

Objecten zijn de centrale entiteiten in het datamodel. Objecten zijn informatiepakketjes die een identificatie, ofwel: identiteit hebben die onafhankelijk is van hun inhoud of geheugen-positie. Objecten bevatten zgn. (structurele) properties. Deze properties kunnen verwijzen naar andere objecten of kunnen waardes bevatten zoals getallen, strings etc. Welke properties een object heeft, en welke waardes of referenties elk van de properties mag bevatten is voorgeschreven door het type van het object. Het object wordt ook wel een instantiatie van z`n type genoemd.

De gebruiker van YANUS kan operaties op objecten toepassen. Operaties zijn expliciet gedefinieerd binnen het systeem door zgn. operatie-(signatuur-)beschrijvingen. Operatie-

beschrijvingen zijn te beschouwen als (object-)types. Een instantiatie van een operatie-beschrijving is een speciaal soort object die dient als aanvraag voor die operatie; de properties van dat object zijn de parameters van die operatie. De gebruiker kan een aanvraag "submitten" voor uitvoering, indien alle properties correct ingevoerd zijn.

Essentieel voor het model is dat types (en dus ook operatie-beschrijvingen) zelf ook weer als objecten worden gerepresenteerd, zodat informatie over het type run-time toegankelijk is. Welke properties een instantiatie van een type mag hebben, en welke waardes of referenties door die properties mogen worden bevat, is vastgelegd in de zgn. property-beschrijvende objecten, die gekoppeld zijn met het type object. Expliciet gerepresenteerd is ook welke operaties op objecten van een bepaald type mogen worden losgelaten, namelijk door middel van referenties van het type naar de overeenkomende operatie-beschrijvingen.

Deze expliciete representatie van types als objecten speelt een belangrijke rol in de implementatie van de dynamiek van het systeem. Een belangrijk deel van de dynamiek van het systeem kan namelijk generiek geïmplementeerd worden, d.w.z. als type-onafhankelijke software. Type-afhankelijkheden in de dynamiek worden verzorgd door de type-informatie te interpreteren. Objecten kuunen bijvoorbeeld generiek gegenereerd worden: het geheugen dat wordt gealloceerd voor een nieuw object is afhankelijk van de set property-beschrijvende-objecten dat aan het type verbonden is. Dat betekent dus, dat één routine (in feite, zoals we zullen zien: één "methode") instantiaties kan creëren voor alle diverse types. Dit is natuurlijk een bekend voorbeeld[6]; het principe wordt veel uitgebreider toegepast in YANUS zoals nog gezien zal worden.

De dynamiek in YANUS is geïmplementeerd in de vorm van methoden. Methoden worden gedefinieerd op het type en zijn toepasbaar op de bijbehorende instantiaties. In tegenstelling tot operaties heeft de gebruiker geen kennis van het bestaan van methoden. Methoden doen onzichtbaar hun werk. Methoden erven (overigens net zoals operatie-beschrijvingen) over in een supertype/subtype-relatie. Dat betekent dat alle methoden die toepasbaar zijn op instantiaties van een bepaald type ook toepasbaar zijn op instantiaties van een subtype van dat type.

Om generieke methodes (zie boven) te kunnen gebruiken worden twee belangrijke concepten geïntroduceerd: metatypes en primaltypes.

---

[6] Het is ook mogelijk, zoals b.v. in C++ om een creatie methode (een constructor) tijdens compile time te genereren. Vaak geldt dat genericiteit vervangen kan worden door software generatie tijdens compile tijd. Dit is ook mogelijk binnen het gegeven datamodel, maar is niet verder bestudeerd.

Metatypes zijn types van types, dat wil zeggen ze beschrijven de structuur van types op dezelfde wijze als types de structuur van hun instantiaties beschrijven. Door middel van het metatype is dus de structur van een type (als zijnde een object) bepaald.

Methodes kunnen generiek zijn, als zij de informatie in het type gebruiken, d.w.z. als ze de structuur van het type gebruiken. Aangezien de structuur van een type bepaald is door het metatype kunnen alle types met hetzelfde metatype dezelfde generieke methodes hebben. Dit wordt gerealiseerd door alle types die instantiaties zijn van het zelfde metatype subtype te laten zijn (eventueel indirect) van éénzelfde type, het zgn. primaltype. Van dat primaltype worden de bijbehorende generieke methoden geërfd. In andere woorden: het primaltype is de root van de subtype/supertype-hiërarchie van alle types die instantiatie zijn van hetzelfde metatype.

Operatie-beschrijvingen hebben een specifiek metatype en zijn een voorbeeld van een mogelijke differentiatie tussen types die kan worden geïntroduceerd met behulp van metatypes. Alle operatie-beschrijvingen erven de methode "Execute" van de primaltype die geassocieerd is met dat metatype. Deze methode is dus toepasbaar op instantiaties van een willekeurige operatie-beschrijving, d.w.z. op een willekeurige aanvraag.

Genericiteit wordt ook gebruikt om de user interface dialoog tussen YANUS en de gebruiker te voeren. De verbindingen tussen operatie beschrijvingen en andere (normale) object types wordt bijvoorbeeld gebruikt om de gebruiker te begeleiden in het correct toepassen van operaties op objecten. Ook het wijzigen van structuur, "editing", van een object wordt met behulp van de kennis in de aan het type gekoppelde property beschrijvende objecten begeleid. Een ander voorbeeld is het gebruik van zgn. "workspaces", d.w.z. specifieke contexten vergelijkbaar met windows, waarin de gebruiker specifieke data kan manipuleren en bijbehorende operaties kan selecteren. Een speciaal metatype is daartoe gedefinieerd; de instantiaties daarvan zijn modules. Workspaces zijn instantiaties van modules. Generieke methoden gedefinieerd op alle modules, en dus toepasbaar op alle workspaces, zijn methoden om een workspace te openen en te sluiten.

In de user interface heeft dus ieder van de gepresenteerde resources (data-objecten, workspaces, operaties) de kennis over z'n eigen gebruik in de vorm van type-informatie aan zich gekoppeld. De generieke software die de dialoog coördineert (de dialoogmanager) interpreteert deze kennis. Dit maakt het mogelijk om het gebruik van een principieel onbegrensd en dynamisch aantal data-objecten, workspaces, operaties, modules semantisch te ondersteunen. In de standaard implementatie van dialoogmanagement, waarbij de kennis over de objecttypen etc. in feite "hard" in de dialoogmanager is ingeprogrammeerd is iets dergelijks praktisch onmogelijk. Dit te meer gezien de eis dat nieuwe soorten resources eenvoudig

toegevoegd moeten kunnen worden aan de integratie, en meteen volledig bruikbaar moeten zijn in combinatie met alle andere resources indien dit zinnig is op grond van de typering.

Dit laatste is dus een essentieel aspect van de beschreven user interface aanpak: nieuwe resources kunnen eenvoudig "ingeplugd" worden door de resources in termen van types operaties etc. te beschrijven. De dynamiek is volledig voorgedefinieerd, en hoeft dus niet te worden uitgebreid. Naast de beschrijving op deze wijze van resources, moet ook een beschrijving van de presentatie gegeven worden. Hoe dit gedaan zou kunnen worden is echter niet bestudeerd.

In het generieke user interface-mechanisme zoals hierboven beschreven worden types gebruikt om het gebruik van hun instantiaties te specificeren. Metatypes kunnen op dezelfde manier gebruikt worden om de manipulatie en editing van hun instantiaties te specificeren. Bijgevolg kan de volledige type-structuur (inclusief operatie beschrijvingen, modules etc.), ook wel "schema" genoemd, met behulp van ditzelfde generieke user interface-mechanisme worden uitgebreid.

Genericiteit wordt ook gebruikt door de methode "Execute" om aanvragen uit te voeren met behulp van de onderliggende resources. Net zoals types van objecten en beschrijvingen van operaties gebruikt kunnen worden om te beschrijven wat de gebruiker moet worden aangeboden, kunnen ook types van objecten en operaties beschreven worden die respectievelijk corresponderen met specifieke datarepresentaties en specifieke operatie-implementaties die de onderliggende software biedt. De types, operatie-beschrijvingen en modules op het gebruikers-nivo worden hierbij ook wel tesamen het conceptuele schema genoemd, terwijl types en operatie-beschrijvingen op het implementatieniveau tesamen het implementatieschema genoemd worden. De structurele koppeling tussen het conceptuele schema en het implementatieschema geeft onder andere de juiste implementaties voor een conceptuele operatie aan. Aangezien verder een operatie-implementatiebeschrijving (net zoals een beschrijving van een conceptuele operatie) aangeeft wat de correcte implementatietypes voor de parameters van die operatie zijn, is dus bekend welke datarepresentatie gegenereerd dient te worden om een bepaalde operatie-implementatie te gebruiken. De methode "Execute" interpreteert al deze kennis. Ook hier maakt de generieke aanpak, met voorgedefinieerde dynamiek, de uitbreiding van de integratie eenvoudig.

Uit het voorgaande mag blijken dat met behulp van metatypes fundamenteel nieuwe faciliteiten in het systeem kunnen worden geïntroduceerd. Modules zijn een voorbeeld van deze mogelijkheid. Net zoals de introductie van nieuwe types ondersteund wordt door metatypes, wordt de introductie van nieuwe metatypes ondersteund door het type aller metatypes. Dit metatype-type is z'n eigen instantiatie. Het datamodel is dus zelf-beschrijvend.

Curriculum vitae

Theo Dirk Meijler was born on August 28, 1960 in Amsterdam, The Netherlands. He completed grammar school in 1978 in Amsterdam. In the fall of the same year he started with Technical Physics at the "Technische Hogeschool" (Now Technical University) in Delft. After his bachelors degree in 1984 he specialized in the application of computer science in experimental physics. In february 1986 he received his masters. In the following autumn he started his obligatory military service, in which he studied noise pollution of military exercise grounds. Two reports were published. In August 1987 he started as a Ph.D. student at the Department of Medical Informatics at the Erasmus University of Rotterdam. He now works at BSO - MediaLab.

User-Level Integration of Data and Operation Resources by means of a Self-Descriptive Data Model

Part II

Formalization

Contents

Chapter 2

The YANUS datamodel

Formalization

Definition 1: (Primitive Domains)
Let DOMAINS = { D_1, ..., D_n }
The set of *Primitive Domains*
We note D the union of all primitive domains:
$D = \bigcup_{Di \in DOMAINS} D_i$

$\forall D_i, D_j \in DOMAINS \mid D_i \neq D_j \wedge D_i \cap D_j \neq \{\} \bullet D_i \subset D_j \vee D_j \subset D_i$
$\{Z, \mathbb{R}, B\} \subset DOMAINS$,
where B the set of boolean values
B = { true, false }

--The domains D_i are either disjoint or one of the domains is a subdomain of the other.
■

Definition 1a: (Other Useful Primitive Domains)
CHAR, the set of characters
STRING, the set of strings: STRING = CHAR*
NAMES, countably infinite set of symbols
TNAMES countably infinite set of (YANUS) *Type Names*
TNAMES $\subset$ NAMES,
BVTNAMES set of names for basic value types
BVTNAMES $\subset$ VTNAMES
BVTNAMES is finite:
BVTNAMES $\in$ TNAMES-set
PNAMES countably infinite set of *property names*
PNAMES $\subset$ TNAMES
PNAMES $\cap$ BVTNAMES = {}
{ NAMES, TNAMES, BVTNAMES, PNAMES } $\subset$ Domains

VALIDITY, finite set of symbols
VALIDITY = { invalid, synt_valid, valid }
< : VALIDITY × VALIDITY
invalid < synt_valid < valid
min : VALIDITY-set → VALIDITY

SORT, finite set of symbols
SORT = { in, out, in_out }
SORT ∈ Domains

--Basic values are drawn from any of the domains. Thus a basic value bv ∈ D.
▪

Definition 1b: (Basic value type names and primitive domains)
BVTNAMES is the set of *basic value type names*

Specifically:

{ Basic_val, Integer, Real, Bool, Name, Tname, Pname, BVTname } ⊂ BVTNAMES

A mapping from basic value type names to the corresponding primitive domain:

domain ∈ BVTNAMES $\xrightarrow{m}$ DOMAINS ∪ D

domain Basic_val = D
domain Integer = Z
domain Real = ℝ
domain Bool = B
domain Char = CHAR
domain String = STRING
domain Name = NAMES
domain Tname = TNAMES
domain Pname = PNAMES
domain BVTname = BVTNAMES
▪

Definition 2: (Identifiers)
Let ID countably infinite set of symbols called *identifiers*.
ID ∩ NAMES = {}
ID ∉ Domains
▪

Definition 3: (Element Values and NIL)
NIL = { Nil }
Nil ∉ D
Nil ∉ ID

EV is the set of element values:
EV = D ∪ ID ∪ NIL
▪
An *element value* is either a basic value or an identifier.

Definition 4: (Property Values)
$PV = ID^* \cup D^*$

--We distinguish between sequences of basic values and sequences of identifiers.
■

Definition 5: (Object Values)
$OV = PNAMES \xrightarrow{m} PV$

$< Pn_1 : v_1, ..., Pn_k : v_k >$ is the partial function t defined on $\{ Pn_1, ..., Pn_k \} \subset PNAMES$, such that $t(Pn_i) = v_i$ for all i, $i \geq 1$, $i \leq k$, where $v_i \in PV$.

Two object values $ov_1, ov_2 \in OV$ are equal, denoted by '=':
$ov_1 = ov_2 <=> dom\ ov_1 = dom\ ov_2 \wedge \forall\ Pn \in PNAMES \mid Pn \in dom\ ov_1 \bullet$
$ov_1(Pn) = ov_2(Pn)$
(using equality between sequences)
■

Definition 6: (Objects)
$O = OV \times ID \times VALIDITY$

An *object* $o \in O$ is a triple:

o = (value, type, validity)

where value $\in$ OV; type $\in$ ID; validity $\in$ VALIDITY
■

Definition 7: (Object identifiers, mapping from identifiers to objects, consistent set of objects)
$IDOBJ = ID \xrightarrow{m} O$
$\gamma \in IDOBJ$
γ is defined globally
$i\gamma = dom\ \gamma$ is the set of identifiers in use

$id_1, id_2 \in dom\ \gamma$
Two objects $\gamma(id_1), \gamma(id_2)$ are identical iff:
$id_1 = id_2$
■

Definition 8: (methods on identifiers)
value : $i\gamma \rightarrow OV$
value(id) $\triangleq \gamma(id)[1]$;
type : $i\gamma \rightarrow i\gamma$
type(id) $\triangleq \gamma(id)[2]$;

validity : iγ → VALIDITY
validity(id) ≙ γ(id)[3];
valid : iγ → B
valid(id) ≙ validity(id) = valid
synt_valid : iγ → B
synt_valid(id) ≙
validity(id) = valid ∨ validity(id) = synt_valid
■

Definition 9: (Get methods)
get : iγ × PNAMES → EV-set
get(iod,Pn) ≙ elems value(iod)(Pn)

singet: iγ × PNAMES → EV
singet(iod,Pn) ≙ if (len(value(iod)(Pn)) ≥ 1)
then value(iod)(Pn) [1]
else Nil
■

Provisional Definition 1: (Ad-hoc introduction of general sets: first level)
IO'' ∈ iγ-set
∀ io ∈ IO'' • type(io) ∈ IOT --See provisional definition 3
IT'' ∈ iγ-set
IVT'' ∈ iγ-set
IBVT'' ∈ iγ-set
IOT'' ∈ iγ-set
IOD'' ∈ iγ-set
ISPD'' ∈ iγ-set
■

Lemma p1: (Subset relationships between general sets, first level)
IBVT'' ⊆ IVT'' ⊆ IT'' ⊆ IO''
ISPD'' ⊆ IVT''
IOD'' ⊆ IOT'' ⊆ IT''
■

Lemma p2: (Properties defined for objects in the general sets)
∀ io ∈ IT'' • { Name, Supertype } ⊆ dom value(io)

∀ io ∈ IOT'' • { Name, Supertype, PropDescriptors, Operations } ⊆ dom value(io)

∀ io ∈ ISPD'' •
{ Supertype, Name, EltType, Minelt, Maxelt, Changeable, Unique } ⊆ dom value(io)
■

Provisional Definition 2: (Ad-hoc introduction of general sets: second level)

IO' ∈ iγ-set
IT' ∈ iγ-set
IVT' ∈ iγ-set
IBVT' ∈ iγ-set
IOT' ∈ iγ-set
IOD' ∈ iγ-set
ISPD' ∈ iγ-set

Let iϕ ∈ IT', iϕ globally defined
iϕ is the *empty type*
■

Lemma p3: (Subset relationships between general sets, second level)

IO' ⊆ IO''
IT' ⊆ IT''
IVT' ⊆ IVT''
IBVT' ⊆ IBVT''
IOT' ⊆ IOT''
IOD' ⊆ IOD''
ISPD' ⊆ ISPD''

IBVT' ⊆ IVT' ⊆ IT' ⊆ IO'
ISPD' ⊆ IVT'
IOD' ⊆ IOT' ⊆ IT'
■

Lemma p4: (Postulating the type of the elements contained in the properties of objects in the general sets, second level)

∀ it ∈ IT' • ∀ v ∈ get(it, Supertype) • v ∈ IT'
∀ it ∈ IT' • 0 ≤ len value(it)(Supertype) ≤ 1
∀ it ∈ IT' • ∀ v ∈ get(it, Name) • v ∈ TNAMES
∀ it ∈ IT' • len value(it)(Name) = 1
∀ ibvt ∈ IBVT' • ∀ v ∈ get(ibvt, Name) • v ∈ BVTNAMES
∀ iot ∈ IOT' • ∀ v ∈ get(iot, PropDescriptors) • v ∈ ISPD'
∀ iot ∈ IOT' • ∀ v ∈ get(iot, Operations) • v ∈ IOD'
∀ ispd ∈ ISPD' • ∀ v ∈ get(ispd, Name) • v ∈ PNAMES
∀ ispd ∈ ISPD' • ∀ v ∈ get(ispd, EltType) • v ∈ IT'
∀ it ∈ ISPD' • len value(it)(EltType) = 1
∀ ispd ∈ ISPD' • ∀ v ∈ get(ispd, Minelt) • v ∈ ℕ
∀ ispd ∈ ISPD' • len value(ispd)(Minelt) = 1
∀ ispd ∈ ISPD' • ∀ v ∈ get(ispd, Maxelt) • v ∈ ℕ
∀ ispd ∈ ISPD' • len value(ispd)(Maxelt) = 1
∀ ispd ∈ ISPD' • ∀ v ∈ get(ispd, Changeable) • v ∈ B
∀ ispd ∈ ISPD' • len value(ispd)(Changeable) = 1
∀ ispd ∈ ISPD' • ∀ v ∈ get(ispd, Unique) • v ∈ B

$\forall$ ispd $\in$ ISPD' • len value(ispd)(Unique) = 1
■

Definition 10: (Methods on IT')
singet(iϕ,Name) = Nil
singet(iϕ,Supertype) = Nil

supertype : IT' $\rightarrow$ IT'
supertype(it) $\triangleq$ if(singet(it,Supertype) = Nil) then iϕ
else singet(it,Supertype)

The relationship subtype_of
subtype_of : IT' $\times$ IT'
it_1 subtype_of $it_2 \triangleq it_2$ = supertype(it_1)

name : IT' $\rightarrow$ TNAMES
name(it) $\triangleq$ singet(it,Name)
■

Definition 11: (Methods on IOT')
propdescriptors : IOT' $\rightarrow$ ISPD'-set
propdescriptors(iot) $\triangleq$ get(iot,PropDescriptors)
operations : IOT' $\rightarrow$ IOD'-set
operations(iot) $\triangleq$ get(iot,Operations)
■

Definition 12: (Methods on ISPD')
elttype : ISPD' $\rightarrow$ IT'
elttype(ispd) $\triangleq$ singet(ispd,EltType)
minelt : ISPD' $\rightarrow$ $\mathbb{N}$
minelt(ispd) $\triangleq$ singet(ispd,Minelt)
maxelt : ISPD' $\rightarrow$ $\mathbb{N}$
maxelt(ispd) $\triangleq$ singet(ispd,Maxelt)
changeable : ISPD' $\rightarrow$ B
changeable(ispd) $\triangleq$ singet(ispd,Changeable)
unique : ISPD' $\rightarrow$ B
unique(ispd) $\triangleq$ singet(ispd,Unique)
■

Provisional Definition 3: (ad-hoc introduction of general sets, third level)
IT $\in$ IT'-set
IBVT $\in$ IBVT'-set
IOT $\in$ IOT'-set
IOD $\in$ IOD'-set
ISPD $\in$ ISPD'-set
■

Lemma p4: (subset relationships)

$IT \subseteq IT'$
$IVT \subseteq IVT'$
$IBVT \subseteq IBVT'$
$IOT \subseteq IOT'$
$IOD \subseteq IOD'$
$ISPD \subseteq ISPD'$

$IBVT \subseteq IVT \subseteq IT$
$ISPD \subseteq IVT$
$IOD \subseteq IOT \subseteq IT$
■

Lemma p5: (values of properties)

$\forall$ it $\in$ IT • supertype(it) $\in$ IT
$\forall$ iot $\in$ IOT • propdescriptors(iot) $\in$ ISPD
$\forall$ iot $\in$ IOT •
  $\forall$ $ipd_1 \in$ propdescriptors(iot) •
    $\neg(\exists$ $ipd_2 \in$ propdescriptors(iot) •
      $ipd_1 \neq ipd_2 \wedge name(ipd_1) = name(ipd_2))$
$\forall$ iot $\in$ IOT • operations(iot) $\in$ IOD
$\forall$ iot $\in$ IOT •
  $\forall$ $iod_1 \in$ operations(iot) •
    $\neg(\exists$ $iod_2 \in$ operations(iot) •
      $iod_1 \neq iod_2 \wedge name(iod_1) = name(iod_2))$
$\forall$ ispd $\in$ ISPD • elttype(ispd) $\in$ IOT $\vee$ elttype(ispd) $\in$ IBVT
$\forall$ ispd $\in$ ISPD • maxelt(ispd) $\geq$ minelt(ispd)
■

Definition 13: (New, preliminary)

New(iot : IOT) inewo : IO''
ext wr : $\gamma \in$ IDOBJ
pre
post post_New(ot, γ, γ, inewo)

post_New : IOT $\times$ IDOBJ $\times$ IDOBJ $\times$ IO'' $\rightarrow$ B
post_New(iot, γ, γ', inewo') $\triangleq$
  $\exists$ inewo' $\in$ ID | inewo' $\notin$ dom γ •
    $\exists$ newo' $\in$ O | ($\forall$ ispd $\in$ propdescriptors(iot) •
    value'(newo')(name(ispd)) = []) $\wedge$ $\gamma' = \gamma \oplus$ (inewo' $\rightarrowtail$ newo') $\wedge$
    if(synt_wf(inewo')) then validity(inewo') = synt_valid
      else validity(inewo') = invalid
    $\wedge$ type(inewo') = iot
■

Lemma 1:
$\forall$ io $\in$ IO'' • let iot = type(io) in
$\forall$ ispd $\in$ propdescriptors(iot) • name(ispd) $\in$ dom value(io)

Proof: By the definition of New
■

Definition 14:
templ_inst_of : IO'' × IOT
io templ_inst_of iot ≙ type(io) subtype_of* iot

synt_inst_of : IO'' × IOT
io synt_inst_of iot ≙ type(io) subtype_of* iot ∧ synt_valid(io)
■

Definition 15: (Syntactic well formedness and related functionality, preliminary definition 1)
synt_wf : IO'' → B
synt_wf(io) ≙ let (validity(io) = synt_valid), iot = type(io) in
iot $\in$ IOT ∧ synt_wf_wrt(io,iot)

synt_wf_wrt : IO'' × IOT → B
synt_wf_wr(io,iot) ≙ io templ_inst_of iot ∧
$\forall$ ipd $\in$ propdescriptors(iot) • propreq(io,ipd)

propreq : IO'' × ISPD → B
propreq(io,ipd) ≙ let Pn = name(ipd) ∧ ieot = elttype(ipd) in
if(ieot $\in$ IBVT) then
($\forall$ v $\in$ get(io,Pn) • v $\in$ domain name(ieot))
elseif(ieot $\in$ IOT) then
($\forall$ id $\in$ get(io,Pn) • id synt_inst_of ieot)
else false ∧
minelt(ipd) ≤ len value(io)(Pn) ≤ maxelt(ipd) ∧ if(unique(ipd) = true) then
unqelt(value(io)(Pn)) else true
■

Lemma 2: (syntactic validity and the predicate synt_wf)
$\forall$ io $\in$ IO'' • synt_wf(io) <=> synt_valid(io)

To be certified. See def. 42 and further
■

Definition 16: (inst_of)
inst_of : IO'' × IOT
io inst_of iot ≙ type(io) subtype_of* iot ∧ valid(io) ■

Definition 17: (Well formedness and related functionality, preliminary definition 1)

17a:

(Extra requirements)

requirement : IO' → B

erequirements ∈ IOT ⇸ requirement

allerequirements ∈ IOT ⇸ requirement

allerequirements(it) ≙ $\bigcup_{itd \in IT \mid it\ subtype\_of^*\ itd}$ erequirements(itd)

17b:

wf : IO'' → B

wf(io) ≙ let (validity(io) = valid), iot = type(io) in
    iot inst_of iObjectType_type ∧ wf_wrt(io,iot)

wf_wrt : IO'' × IOT → B

wf_wrt(io,iot) ≙ io templ_inst_of iot ∧
    synt_valid(io) ∧ (∀ req ∈ allerequirements(iot) • req(io)) ∧ (
        ∀ ipd ∈ propdescriptors(iot) • wfprop(io,ipd)
)

wfprop : IO' × ISPD → B

wfprop(io,ipd) ≙ let Pn = name(ipd) ∧ ieot = elttype(ipd) in
    if(ieot inst_of iBasicValueType_type) then
        true
            --extra requirements are only implemented on objects!
    elseif(ieot inst_of iObjectType_type) then
        (∀ id ∈ get(io,Pn) • id inst_of ieot)
    else false

■

Lemma 3: (validity and the predicate wf)

∀ io ∈ IO'' • valid(io) => wf(io)

To be certified. See def. 36

■

Subtyping

Definition 18: (Interpretations)

M : IOT → IO''

M(iot) ≙ { io ∈ IO | io inst_of iot }

M(iot) is the *model* of iot

similarly:
M`` : IOT → IO``
M``(iot) ≙ { io ∈ IO | io templ_inst_of iot }
M` : IOT → IO``
M`(iot) ≙ { io ∈ IO | io synt_inst_of iot }
∎

Definition 19:
(Structural subtyping requirement on basis of the types` Model)
$\leq_{st}^{I} \in$ IOT × IOT

$iot_1 \leq_{st}^{I} iot_2 \triangleq$
$M(iot_1) \subseteq M(iot_2)$
∎

Lemma 4: (instance of a type than also instance of the supertype)

4a:
io templ_inst_of it => io templ_inst_of sypertype(it)

4b:
io synt_inst_of it => io synt_inst_of sypertype(it)

4c:
io inst_of it => io inst_of supertype(it)

Proof: (4a)
from io templ_inst_of it
type(io) subtype_of* it (Def. 14)
it subtype_of supertype(it) (Def. 10)
type(io) subtype_of* supertype(it)
infer io templ_inst_of supertype(it)

similar for 4b and 4c.
∎

Lemma 5: (Subtyping axiom conforms Structural subtyping requirement on basis of the Model)
Let $iot_1, iot_2 \in$ IOT
iot_1 subtype_of* iot_2 => $iot_1 \leq_{st}^{I} iot_2$

Proof:
first:

from io inst_of iot => io inst_of supertype(iot) (Lem. 4)
$\forall$ io $\in$ IO'' • io inst_of iot => io inst_of supertype(iot)
$\forall$ io $\in$ IO'' • io $\in$ M(iot) => io $\in$ M(supertype(iot))
infer
M(iot) $\subset$ M(supertype(iot))

from iot_1 subtype_of* iot_2
$M(iot_1) \subset M(iot_2)$ (repeated application of the previous)
infer
$iot_1 \leq_{st}^{1} iot_2$
■

Definition 20: (Structural subtyping requirement on basis of property definitions and extra requirements, definition 1 preliminary)
$\leq_{st}^{2}$: (IBVT × IBVT) $\cup$ (IOT × IOT) $\cup$ (ISPD × ISPD)

Let $ibvt_1$, $ibvt_2 \in$ IBVT
$ibvt_1 \leq_{st}^{2} ibvt_2$ => domain name($ibvt_1$) $\subseteq$ domain name($ibvt_2$)

Let iot_1, $iot_2 \in$ IOT
$iot_1 \leq_{st}^{2} iot_2 \triangleq$
$iot_1 = iot_2$
$\vee$
($\forall$ $ipd_2 \in$ propdescriptors(iot_2) •
$\exists$ $ipd_1 \in$ propdescriptors(iot_1) •
(ipd_1 subtype_of* ipd_2
$\vee$
$ipd_1 \leq_{st}^{2} ipd_2$)
$\wedge$ allerequirements(iot_1) $\subseteq$ allerequirements(iot_1))

Let ipd_1, $ipd_2 \in$ ISPD
$ipd_1 \leq_{st}^{2} ipd_2 \triangleq$
$ipd_1 = ipd_2$
$\vee$
(name(ipd_2) = name(ipd_1) $\wedge$
elttype(ipd_1) $\leq_{st}^{2}$ elttype(ipd_2) $\wedge$
minelt(ipd_1) $\geq$ minelt(ipd_2) $\wedge$
maxelt(ipd_1) $\leq$ maxelt(ipd_2) $\wedge$
unique(ipd_2) => unique(ipd_1)
)
■

Definition 21: (Subtyping requirement on basis of applicable operations)
$\leq_{st}^{3}$: IOT × IOT
$iot_1 \leq_{st}^{3} iot_2 \triangleq$
$\forall$ iopd $\in$ operations(iot_2) • iopd $\in$ operations(iot_1) ■

Definition 22: (General subtyping requirement)
$\leq_{st}$: IOT × IOT
$iot_1 \leq_{st} iot_2 \triangleq iot_1 \leq_{st}^2 iot_2 \wedge iot_1 \leq_{st}^3 iot_2$
■

Lemma 6: (subtyping relationship implies subtyping)
$\forall\ iot_1, iot_2 \in IOT \mid iot_1$ subtype_of$^+$ $iot_2 \bullet iot_1 \leq_{st} iot_2$

To be certified. See def. 23c.
■

Lemma 7: (properties of an object)
$\forall\ io \in IO'' \bullet \forall\ iot \in IOT \mid$ io templ_inst_of iot •
$\forall\ ispd \in$ propdescriptors(iot) • name(ispd) $\in$ dom value(io)

Proof:

from	io templ_inst_of iot	
	let iot_0 = type(io)	
	iot_0 subtype_of* iot	(Def. 14)
	$iot_0 \leq_{st} iot$	(Def. 14)
	$\forall\ ispd \in$ propdescriptors(iot) • $\exists\ ispd_0 \in$ propdescriptors(iot_0) • name($ispd_0$) = name(ispd)	(Def. 20)
	$\forall\ ispd_0 \in$ propdescriptors(iot_0) • name($ispd_0$) $\in$ dom value(io)	(Lemma 1)
infer	$\forall\ ispd \in$ propdescriptors(iot) • name(ispd) $\in$ dom value(io)	

■

Lemma 8: (If an object is well formed following the subtype, it is well formed following the supertype)
Let io $\in$ IO; iot $\in$ IOT
wf_wrt(io,iot) => wf_wrt(io,supertype(iot))

Proof:
To prove this, we first prove:
synt_wf_wrt(io,iot) => synt_wf_wrt(io, supertype(iot))

from	$iot_1, iot_2 \in$ IOT; iot_1 subtype_of iot_2	
infer		
	$iot_1 \leq_{st}^2 iot_2$	(Lemma 6)

To be proved is now:

from	$iot_1 \leq_{st}^2 iot_2$
infer:	
	$\forall\ io \in IO \mid$ type(io) subtype_of(iot_1) • synt_wf(io,iot_1) => synt_wf(io,iot_2)

Let io ∈ IO'; synt_wf(io,iot_1); io further arbitrary.
(Thus, for the following, ∀ io ∈ IO | synt_wf(io,iot_1) •, may be used as prefix)
{ p_1, ..., p_i } = propdescriptors(iot_2);
{ p'_1, ..., p'_l } = propdescriptors(iot_1);
Lemma:
$l \geq i$
This can be proved from:
$\forall j \in \mathbb{N}+ \mid j \leq i \bullet (\exists n \in \mathbb{N}+ \mid n \leq l \bullet name(p'_n) = name(p_j))$
(definition of $\leq_{st}^2$)
and from req_{otl} which states that names of properties must be different.
Thus we can write:
{ p'_1, ..., p'_i, p'_{i+1}, ..., p'_{i+k} } = propdescriptors(iot_1)
where
$\forall j \in \mathbb{N}+ \mid j \leq i \bullet name(p'_j) = name(p_j)$

1. First assume that $\forall j \in \mathbb{N}+ \mid j \leq i \bullet p'_j = p_j$
Then
from $\forall j \in \mathbb{N}+ \mid j \leq i+k \bullet propreq(io,p'_j)$
$\forall j \in \mathbb{N}+ \mid j \leq i \bullet propreq(io,p'_j)$
$\forall j \in \mathbb{N}+ \mid j \leq i \bullet propreq(io,p_j)$ (Assumption)
infer
synt_wf(io,iot_2)

2. Now we leave out the assumption. We have to prove that from: $\forall j \in \mathbb{N}+ \mid j \leq i \bullet$ propreq(io,p'_j)
we can infer:
$\forall j \in \mathbb{N}+ \mid j \leq i \bullet propreq(io,p_j)$
that is, if we can infer:
$\forall j \in \mathbb{N}+ \mid j \leq i \wedge p'_j \neq p_j \bullet propreq(io,p_j)$

In this we use that from:
$name(p'_j) = name(p_j)$
we can infer:
$p'_j \leq_{st}^2 p_j$ (Definition 20)

We enumerate, for arbitrary p'_j the subrequirements of propreq showing that if $p'_j \leq_{st}^2 p_j$ and propreq(io,p'_j), that than also propreq(io,p_j).

from $name(p'_j) \in$ dom value(io)
infer $name(p_j) \in$ dom value(io) ($name(p'_j) = name(p_j)$)

Let $name(p'_j) = name(p_j) = P_j$

from if(elttype(p'$_j$) inst_of iBasicValueType_type) then
$\forall$ v $\in$ get(io,P$_j$) • v $\in$ domain name(elttype(p'$_j$))
infer if(elttype(p'$_j$) inst_of iBasicValueType_type) then
$\forall$ v $\in$ get(io,P$_j$) • v $\in$ domain name(elttype(p$_j$))
elttype(p'$_j$) $\leq_{st}^2$ elttype(p$_j$) => domain name(elttype(p'$_j$)) $\subseteq$ domain name(elttype(p$_j$))

from if(elttype(p'$_j$) inst_of iObjectType_type) then
($\forall$ id $\in$ get(io,P$_j$) • id synt_inst_of elttype(p'$_j$))
infer if(elttype(p'$_j$) inst_of iObjectType_type) then
($\forall$ id $\in$ get(io,P$_j$) • id synt_inst_of elttype(p$_j$))
elttype(p'$_j$) $\leq_{st}^2$ elttype(p$_j$); this proof

from minelt(p'$_j$) $\leq$ len value(io)(P$_j$) $\leq$ maxelt(p'$_j$)
infer minelt(p$_j$) $\leq$ len value(io)(P$_j$) $\leq$ maxelt(p$_j$)
minelt(p'$_j$) $\geq$ minelt(p$_j$); maxelt(p'$_j$) $\leq$ maxelt(p$_j$)

For uniqueness we have the following cases:
I. unique(p$_j$) then also: unique(p'$_j$) (from p'$_j$ $\leq_{st}$ p$_j$)
In that case we must have: unqelt(value(io)(P$_j$)) for both p'$_j$ and p$_j$.
II. $\neg$unique(p$_j$) then either: unique(p'$_j$) or: $\neg$unique(p'$_j$)
(from p'$_j$ $\leq_{st}^2$ p$_j$). For p$_j$ no extra requirement is imposed, thus either values for which unqelt(value(io)(P$_j$)) holds and values for which this doesn't hold are allowed.
--And therefore: all property values which are valid for the subtype are also valid for the supertype--

Now we have to proof that:

wf_wrt(io,iot$_1$) => wf_wrt(io,iot$_2$)

Finally we have that:
from iot$_1$ subtype_of iot$_2$
infer
allerequirements(iot$_1$) =
allerequirements(iot$_2$) $\cup$ erequirements(iot$_1$)

and thus:
from wf(io,iot$_1$)
$\forall$ req $\in$ allerequirements(iot$_1$) • req(io)
infer
$\forall$ req $\in$ allerequirements(iot$_2$) • req(io)

and also:
from $\forall$ j $\in$ $\mathbb{N}$+ | j $\leq$ i • p'$_j$ $\leq_{st}^2$ p$_j$
$\forall$ j $\in$ $\mathbb{N}$+ | j $\leq$ i • ($\forall$ id $\in$ get(io,P$_j$) • id inst_of elttype(p'$_j$))
infer
$\forall$ j $\in$ $\mathbb{N}$+ | j $\leq$ i • ($\forall$ id $\in$ get(io,P$_j$) • id inst_of elttype(p$_j$)) (Lemma 4)■

Definition 23:
InitTypeSet ∈ IOT-set

InitTypeSet =
{ iObject_type, iType_type, iValueType_type, iBasicValueType_type,
iStructPropDescr_type, iObjectType_type, iOperationDescr_type,
iMetaType_type }

InitPropDescriptors ∈ ISPD-set

InitPropDescriptors =
{ iName, iName', iName'', iSupertype, iEltType, iMinelt, iMaxelt,
iChangeable, iUnique, iPropDescriptors, iOwnPropDescriptors, iOperations,
iOwnOperations, iPrimalType, iPrimalProp_descr }

InitValueTypes ∈ IBVT-set

InitValueTypes =
{ iBasicValue_type, iBool_type, iPosInteger_type, iTname_type,
iBVTname_type, iPname_type }

iObject_type inst_of iObjectType_type
name(iObject_type) = Object_type
iObject_type subtype_of iϕ

iType_type inst_of iMetaType_type
name(iType_type) = Type_type
iType_type subtype_of* iObject_type
propdescriptors(iType_type) =
{ iName, iSupertype }
ownpropdescriptors(iType_type) =
{ iName, iSupertype }
operations(iType_type) =
{ }
ownoperations(iType_type) =
{ }
primaltype(iType_type) = iϕ

iName inst_of iStructPropDescr_type
name(iName) = Name
iName subtype_of* iPrimalProp_descr
elttype(iName) =
iTname_type
minelt(iName) = 1
maxelt(iName) = 1

```
                    changeable(iName) =
                                        false
                    unique(iName) =     true
          iSupertype inst_of iStructPropDescr_type
                    name(iSupertype) =  Supertype
                    iSupertype subtype_of* iPrimalProp_descr
                    elttype(iSupertype) =
                                        iType_type
                    minelt(iSupertype) =
                                        0
                    maxelt(iSupertype) =
                                        1
                    changeable(iSupertype) =
                                        false
                    unique(iSupertype) =
                                        true

iValueType_type inst_of iMetaType_type
          name(iValueType_type =        ValueType_type
          supertype(iValueType_type) =  iType_type
          propdescriptors(iValueType_type) =
                                        { iName, iSupertype }
          ownpropdescriptors(iValueType_type) =
                                        { }
          operations(iValueType_type) =
                                        { }
          ownoperations(iValueType_type) =
                                        { }

iBasicValueType_type inst_of iMetaType_type
          name(iBasicValueType_type =   BasicValueType_type
          supertype(iBasicValueType_type) =
                                        iValueType_type
          propdescriptors(iBasicValueType_type) =
                                        { iName', iSupertype }
          ownpropdescriptors(iBasicValueType_type) =
                                        { iName' }
          operations(iBasicValueType_type) =
                                        { }
          ownoperations(iBasicValueType_type) =
                                        { }
          primaltype(iBasicValueType_type) =
                                        iBasicValue_type
                    iName' inst_of iStructPropDescr_type
                              name(iName') =    Name
                              iName' subtype_of iName
```

```
                                elttype(iName') =       iBVTname_type
                                minelt(iName') =        1
                                maxelt(iName') =        1
                                changeable(iName') =
                                                        false
                                unique(iName') =        true

iStructPropDescr_type inst_of iMetaType_type
        name(iStructPropDescr_type) =
                                        StructPropDescr_type
        supertype(iStructPropDescr_type) =
                                        iValueType_type
        propdescriptors(iStructPropDescr_type) =
                                        { iName'', iSupertype, iEltType, iMinelt,
                                        iMaxelt, iChangeable, iUnique }
        ownpropdescriptors(iStructPropDescr_type) =
                                        { iName'', iEltType, iMinelt, iMaxelt,
                                        iChangeable, iUnique }
        operations(iStructPropDescr_type) =
                                        { }
        ownoperations(iStructPropDescr_type) =
                                        { }
        primaltype(iStructPropDescr_type) =
                                        iPrimalProp_descr
                iName'' inst_of iStructPropDescr_type
                        name(iName'') =         Name
                        iName'' subtype_of      iName
                        elttype(iName'') =      iPname_type
                        minelt(iName'') =       1
                        maxelt(iName'') =       1
                        changeable(iName'') =
                                                false
                        unique(iName'') =       true
                iEltType inst_of iStructPropDescr_type
                        name(iEltType) =        EltType
                        iEltType subtype_of* iPrimalProp_descr
                        elttype(iEltType) =     iType_type
                        minelt(iEltType) =      1
                        maxelt(iEltType) =      1
                        changeable(iEltType) =
                                                false
                        unique(iEltType) =      true
                iMinelt inst_of iStructPropDescr_type
                        name(iMinelt) =         Minelt
                        iMinelt subtype_of* iPrimalProp_descr
                        elttype(iMinelt) =      iPosInteger_type
```

```
                    minelt(iMinelt) =       1
                    maxelt(iMinelt) =       1
                    changeable(iMinelt) =
                                            false
                    unique(Minelt) =        true
                iMaxelt inst_of iStructPropDescr_type
                    name(iMaxelt) =         Maxelt
                    iMaxelt subtype_of* iPrimalProp_descr
                    elttype(iMaxelt) =      iPosInteger_type
                    minelt(iMaxelt) =       1
                    maxelt(iMaxelt) =       1
                    changeable(iMaxelt) =
                                            false
                    unique(Maxelt) =        true
                iChangeable inst_of iStructPropDescr_type
                    name(iChangeable) =
                                            Changeable
                    iChangeable subtype_of* iPrimalProp_descr
                    elttype(iChangeable) =
                                            iBool_type
                    minelt(iChangeable) =
                                            1
                    maxelt(iChangeable) =
                                            1
                    changeable(iChangeable) =
                                            false
                    unique(iChangeable) =
                                            true
                iUnique inst_of iStructPropDescr_type
                    name(iUnique) =         Unique
                    iUnique subtype_of* iPrimalProp_descr
                    elttype(iUnique) =      iBool_type
                    minelt(iUnique) =       1
                    maxelt(iUnique) =       1
                    changeable(iUnique) =
                                            false
                    unique(iUnique) =       true

iObjectType_type inst_of iMetaType_type
        name(iObjectType_type) =            ObjectType_type
        supertype(iObjectType_type) =
                                            iType_type
        propdescriptors(iObjectType_type) =
                                            { iName, iSupertype, iPropDescriptors,
                                            iOwnPropDescriptors, iOperations,
                                            iOwnOperations }
```

```
ownpropdescriptors(iObjectType_type) =
                                { iPropDescriptors, iOwnPropDescriptors,
                                iOperations, iOwnOperations }
operations(iObjectType_type) =
                                { iDeriveProperties, iDeriveOperations }
ownoperations(iObjectType_type) =
                                { iDeriveProperties, iDeriveOperations }
primaltype(iObjectType_type) =
                                iObject_type
        iPropDescriptors inst_of iStructPropDescr_type
                name(iPropDescriptors) =
                                PropDescriptors
                iPropDescriptors subtype_of* iPrimalProp_descr
                elttype(iPropDescriptors) =
                                iStructPropDescr_type
                minelt(iPropDescriptors) =
                                0
                maxelt(iPropDescriptors) =
                                ∞
                changeable(iPropDescriptors) =
                                false
                unique(iPropDescriptors) =
                                true
        iOwnPropDescriptors inst_of iStructPropDescr_type
                name(iOwnPropDescriptors) =
                                OwnPropDescriptors
                supertype(iOwnPropDescriptors) =
                                iPrimalProp_descr
                elttype(iOwnPropDescriptors) =
                                iStructPropDescr_type
                minelt(iOwnPropDescriptors) =
                                0
                maxelt(iOwnPropDescriptors) =
                                ∞
                changeable(iOwnPropDescriptors) =
                                false
                unique(iOwnPropDescriptors) =
                                true
        iOperations inst_of iStructPropDescr_type
                name(iOperations) =
                                Operations
                iOperations subtype_of* iPrimalProp_descr
                elttype(iOperations) =
                                iOperationDescr_type
                minelt(iOperations) =
                                0
```

```
                    maxelt(iOperations) =
                                        ∞
                    changeable(iOperations) =
                                        false
                    unique(iOperations) =
                                        true
                iOwnOperations inst_of iStructPropDescr_type
                    name(iOwnOperations) =
                                        OwnOperations
                    supertype(iOwnOperations) =
                                        iPrimalProp_descr
                    elttype(iOwnOperations) =
                                        iOperationDescr_type
                    minelt(iOwnOperations) =
                                        0
                    maxelt(iOwnOperations) =
                                        ∞
                    changeable(iOwnOperations) =
                                        false
                    unique(iOwnOperations) =
                                        true

                iDeriveProperties inst_of iOperationDescr_type
                iDeriveOperations inst_of iOperationDescr_type

iOperationDescr_type inst_of iMetaType_type
        propdescriptors(iOperationDescr_type) =
                                { iName, iSupertype, iPropDescriptors,
                                iOwnPropDescriptors, iOperations,
                                iOwnOperations }
        ownpropdescriptors(iOperationDescr_type) =
                                { }
        operations(iOperationDescr_type) =
                                { iDeriveProperties, iDeriveOperations }
        ownoperations(iOperationDescr_type) =
                                { }

iMetaType_type inst_of iMetaType_type
        name(iMetaType_type) =          MetaType_type
        iMetaType_type subtype_of iObjectType_type
        propdescriptors(iMetaType_type) =
                                { iName, iSupertype, iPropDescriptors,
                                iOwnPropDescriptors, iOperations,
                                iOwnOperations, iPrimalType }
        ownpropdescriptors(iMetaType_type) =
                                { iPrimalType }
```

operations(iMetaType_type) =
{ iDeriveProperties, iDeriveOperations }
ownoperations(iMetaType_type) =
{ }
primaltype(iMetaType_type) =
iType_type

iPrimalType inst_of iStructPropDescr_type
name(iPrimalType) = PrimalType
supertype(iPrimalType) = iPrimalProp_descr
elttype(iPrimalType) = iType_type
minelt(iPrimalType) = 0
maxelt(iPrimalType) = 1
changeable(iPrimalType) = false
unique(iPrimalType) = true

23a: (Auxiliary Type definitions:)

iPrimalProp_descr inst_of iStructPropDescr_type
name(iPrimalProp_descr) = PrimalProp_descr
supertype(iPrimalProp_descr) =
iϕ
elttype(iPrimalProp_descr) = iϕ
minelt(iPrimalProp_descr) = 0
maxelt(iPrimalProp_descr) = ∞
changeable(iPrimalProp_descr) =
true
unique(iPrimalProp_descr) = false

iBasicValue_type inst_of iBasicValueType_type
name(iBasicValue_type) = Basic_val
Basic_val subtype_of* iϕ

iBool_type inst_of iBasicValueType_type
name(iBool_type) = Bool
supertype(iBool_type) = iBasicValue_type

iPosInteger_type inst_of iBasicValueType_type
name(iPosInteger_type) = PosInteger
supertype(iPosInteger_type) = iBasicValue_type

domain PosInteger = ℕ

iReal_type inst_of iBasicValueType_type
name(iReal_type) = Real
supertype(iReal_type) = iBasicValue_type

iChar_type inst_of iBasicValueType_type
name(iChar_type) = Char
supertype(iChar_type) = iBasicValue_type

iString_type inst_of iBasicValueType_type
name(iString_type) = String
supertype(iString_type) = iBasicValue_type

iTname_type inst_of iBasicValueType_type
name(iTname_type) = Tname
supertype(iTname_type) = iBasicValue_type

iBVTname_type inst_of iBasicValueType_type
name(iBVTname_type) = BVTname
supertype(iBVTname_type) = iBasicValue_type

iPname_type inst_of iBasicValueType_type
name(iPname_type) = Pname
supertype(iPname_type) = iBasicValue_type

(See also definition 1b).

23b: (Auxiliary methods)
ownpropdescriptors : IOT' → ISPD'-set
ownpropdescriptors(iot) ≙ get(iot,OwnPropDescriptors)

ownoperations : IOT' → IOD'-set
ownoperations(iot) ≙
get(iot,OwnOperations)
IMT' = M'(iMetaType_type)
primaltype : IMT' → IT'
primaltype(imt) ≙
if(singet(imt,PrimalType) = Nil) then iϕ
else singet(imt,PrimalType)

23c: (Extra requirements)
req_{bvt1} : IBVT' → B
req_{bvt1}(ibvt) ≙ domain name(ibvt) ⊆ domain name(supertype (ibvt))
erequirements(iBasicValueType_type) = { req_{bvt1} }

req_{ispd1} : ISPD' → B
req_{ispd2} : ISPD' → B
req_{ispd3} : ISPD' → B

req_{ispd1}(ispd) ≙ elttype(ispd) inst_of iObjectType_type ∨ elttype(ispd) inst_of iBasicValueType_type

$req_{spd2}(ispd) \triangleq$ maxelt(ispd) ≥ minelt(ispd)
$req_{spd3}(ispd) \triangleq$ (elttype(ispd) inst_of iObjectType_type)
=> unique(ispd) = true)

erequirements(iStructPropDescr_type) = { req_{spd1}, req_{spd2}, req_{spd3} }

$req_{iot1} : IOT` \rightarrow B$
$req_{iot1}(iot) \triangleq$
$\forall ipd_1 \in$ propdescriptors(iot) •
$\neg(\exists ipd_2 \in$ propdescriptors(iot) •
$ipd_1 \neq ipd_2 \wedge name(ipd_1) = name(ipd_2))$

$req_{iot2} : IOT` \rightarrow B$
$req_{iot2}(iot) \triangleq$
$\forall iod_1 \in$ operations(iot) •
$\neg(\exists iod_2 \in$ operations(iot) •
$iod_1 \neq iod_2 \wedge name(iod_1) = name(iod_2))$

$req_{iot3} : IOT` \rightarrow B$
$req_{iot3}(iot) \triangleq$
$\forall ipd_2 \in$ propdescriptors(supertype(iot)) •
$\exists ipd_1 \in$ propdescriptors(iot) •
ipd_1 subtype_of* ipd_2
$\vee$
$ipd_1 \leq_{st}^2 ipd_2$

$req_{iot4} : IOT` \rightarrow B$
$req_{iot4}(iot) \triangleq$
$\forall iod_2 \in$ operations(supertype(iot)) •
$iod_2 \in$ operations(iot)

erequirements(iObjectType_type) = { req_{iot1}, req_{iot2}, req_{iot3}, req_{iot4} }

$req_{it1} : IT` \rightarrow B$
$req_{it1}(it) \triangleq$ let imt = type(it)
it subtype_of* primaltype(imt)
$req_{it1} \in$ erequirements(iType_type)
■

Definition 24: (Final definition of sets)
IO`` =
{ id ∈ iγ | type(id) ∈ IOT }

IT`` = M``(iType_type)
IVT`` = M``(iValueType_type)
IBVT`` = M``(iBasicValueType_type

ISPD'' = M''(iStructPropDescr_type)
IOT'' = M''(iObjectType_type)
IMT'' = M''(iMetaType_type)

IT' = M'(iType_type)
IVT' = M'(iValueType_type)
IBVT' = M'(iBasicValueType_type
ISPD' = M'(iStructPropDescr_type)
IOT' = M'(iObjectType_type)
IMT' = M'(iMetaType_type)

IT = M(iType_type)
IVT = M(iValueType_type)
IBVT = M(iBasicValueType_type
ISPD = M(iStructPropDescr_type)
IOT = M(iObjectType_type)
IMT = M(iMetaType_type)
■

Lemma 9: (Metatypes are well formed)
$\forall$ imt $\in$ InitTypeSet • wf(imt)

from the definition of wf (Def. 17) and Def. 23.
■

Definition 25: (inheritance mechanism, redefinition of req_{iot3} and req_{iot4})
mk_propdescriptors : IOT' $\rightarrow$ ISPD'-set
mk_propdescriptors(iot) $\triangleq$
if(supertype(iot) = iϕ) then ownpropdescriptors(iot)
else ownpropdescriptors(iot) $\cup$ propdescriptors(supertype(iot)) $\setminus$ { $ipd_2 \in$ propdescriptors(supertype(iot)) | ($\exists$ $ipd_1 \in$ ownpropdescriptors(iot) • $ipd_1 \leq_{st}^2$ ipd_2) }

req_{iot3}(iot) $\triangleq$ propdescriptors(iot) = mk_propdescriptors(iot)
$req_{iot3} \in$ erequirements(iObjectType_type)

post_DeriveProperties : IOT' $\times$ IOT' $\times$ IDOBJ $\times$ IDOBJ $\rightarrow$ B
post_DeriveProperties(iot, iot', γ, γ') $\triangleq$
propdescriptors(iot') = mk_propdescriptors(iot) $\wedge$
synt_wf(iot') $\wedge$ RestObjSameVal(iot, iot', { PropDescriptors }) $\wedge$ γ' = γ

mk_operations : IOT' $\rightarrow$ IOD'-set
mk_operations(iot) $\triangleq$
ownoperations(iot) $\cup$ operations(supertype(iot)) $\setminus$
{ $iopd_1 \in$ ownoperations(iot) | ($\exists$ $iopd_2 \in$ operations(supertype(iot)) • $iopd_1$ subtype_of* $iopd_2$) }

req_{iot4} : IOT' → B
req_{iot4}(iot) ≙ operations(iot) = mk_operations(iot)

post_DeriveOperations : IOT' × IOT' × IDOBJ × IDOBJ → B
post_DeriveOperations(iot, iot', γ, γ') ≙
operations(iot') = mk_operations(iot) ∧ synt_wf(iot') ∧
RestObjSameVal(iot, iot', { Operations }) ∧ γ' = γ
■

Definition 26: (Reference and attribute descriptors)
iReferenceDescr_type inst_of iMetaType_type
name(iReferenceDescr_type) =
ReferenceDescr_type
supertype(iReferenceDescr_type) =
iStructPropDescr_type
propdescriptors(iReferenceDescr_type) =
{ iName'', iSupertype, iEltType, iMinelt,
iMaxelt, iChangeable, iUnique }
ownpropdescriptors(iReferenceDescr_type) =
{ }
operations(iReferenceDescr_type) =
{ }
ownoperations(iReferenceDescr_type) =
{ }
primaltype(iReferenceDescr_type) =
iPrimalRef_descr

iAttributeDescr_type inst_of iMetaType_type
name(iAttributeDescr_type) = AttributeDescr_type
supertype(iAttributeDescr_type) =
iStructPropDescr_type
propdescriptors(iAttributeDescr_type) =
{ iName'', iSupertype, iEltType, iMinelt,
iMaxelt, iChangeable, iUnique }
ownpropdescriptors(iAttributeDescr_type) =
{ }
operations(iAttributeDescr_type) =
{ }
ownoperations(iAttributeDescr_type) =
{ }
primaltype(iAttributeDescr_type) =
iPrimalAttr_descr

iPrimalRef_descr inst_of iReferenceDescr_type
name(iPrimalRef_descr) = PrimalRef_descr
supertype(iPrimalRef_descr) = iPrimalProp_descr

elttype(iPrimalRef_descr) = iObject_type
minelt(iPrimalRef_descr) = 0
maxelt(iPrimalRef_descr) = ∞
changeable(iPrimalRef_descr) = true
unique(iPrimalRef_descr) = false

iPrimalAttr_descr inst_of iAttributeDescr_type
name(iPrimalAttr_descr) = PrimalAttr_descr
supertype(iPrimalAttr_descr) = iPrimalProp_descr
elttype(iPrimalAttr_descr) = iBasicValue_type
minelt(iPrimalAttr_descr) = 0
maxelt(iPrimalAttr_descr) = ∞
changeable(iPrimalAttr_descr) = true
unique(iPrimalAttr_descr) = false

■

Definition 27: (Preparing for relationships: superobj, refobjs)
iObject_type :
ownpropdescriptors(iObject_type) = { iSuperObj, iRefObjs }

iSuperObj inst_of iReferenceDescr_type
name(iSuperObj) = SuperObj
supertype(iSuperObj) = iPrimalRef_descr
elttype(iSuperObj) = iObject_type
minelt(iSuperObj) = 0
maxelt(iSuperObj) = 1
changeable(iSuperObj) = true
unique(iSuperObj) = false

iRefObjs inst_of iReferenceDescr_type
name(iRefObjs) = RefObjs
supertype(iRefObjs) = iPrimalRef_descr
elttype(iRefObjs) = iObject_type
minelt(iRefObjs) = 0
maxelt(iRefObjs) = ∞
changeable(iRefObjs) = true
unique(iRefObjs) = false

```
superobj : IO → IO
superobj(io) ≜
        singet(io,SuperObj)

refobjs : IO → IO-set
refobjs(io) ≜
        get(io,RefObjs)
```
■

Definition 28: (Cross-reference and component relationships)

```
iCrossRefDescr_type inst_of iMetaType_type
        name(iCrossRefDescr_type) =     CrossRefDescr_type
        supertype(iCrossRefDescr_type) =
                                        iStructPropDescr_type
        propdescriptors(iCrossRefDescr_type) =
                                        { iName'', iSupertype, iEltType, iMinelt,
                                        iMaxelt, iChangeable, iUnique }
        ownpropdescriptors(iCrossRefDescr_type) =
                                        { }
        operations(iCrossRefDescr_type) =
                                        { }
        ownoperations(iCrossRefDescr_type) =
                                        { }
        primaltype(iCrossRefDescr_type) =
                                        iPrimalCrRef_descr

                iPrimalCrRef_descr inst_of iStructPropDescr_type
                        name(iPrimalCrRef_descr) =
                                        PrimalCrRef_descr
                        supertype(iPrimalCrRef_descr) =
                                        iPrimalProp_descr
                        elttype(iPrimalCrRef_descr) =
                                        iObject_type
                        minelt(iPrimalCrRef_descr) =
                                        0
                        maxelt(iPrimalCrRef_descr) =
                                        ∞
                        changeable(iPrimalCrRef_descr) =
                                        true
                        unique(iPrimalCrRef_descr) =
                                        false

iComponentDescr_type inst_of iMetaType_type
        name(iComponentDescr_type) =   ComponentDescr_type
        supertype(iComponentDescr_type) =
                                        iStructPropDescr_type
```

```
propdescriptors(iComponentDescr_type) =
                                    { iName'', iSupertype, iEltType, iMinelt,
                                    iMaxelt, iChangeable, iUnique }
ownpropdescriptors(iComponentDescr_type) =
                                    { }
operations(iComponentDescr_type) =
                                    { }
ownoperations(iComponentDescr_type) =
                                    { }
primaltype(iComponentDescr_type) =
                                                iPrimalComp_descr

        iPrimalComp_descr inst_of iStructPropDescr_type
                name(iPrimalComp_descr) =
                                                PrimalComp_descr
                supertype(iPrimalComp_descr) =
                                                iPrimalProp_descr
                elttype(iPrimalComp_descr) =
                                                iObject_type
                minelt(iPrimalComp_descr) =
                                                0
                maxelt(iPrimalComp_descr) =
                                                ∞
                changeable(iPrimalComp_descr) =
                                                true
                unique(iPrimalComp_descr) =
                                                false
```

■

Definition 29: (More detailed definition of property descriptors in InitPropDescriptors in definition 23, taking relationships, attributes, etc. into account)

```
iName inst_of iAttrDescr_type
supertype(iName) =          iPrimalAttr_descr

iName' inst_of iAttrDescr_type
supertype(iName') =         iName

iName'' inst_of iAttrDescr_type
supertype(iName'') = iName

iSupertype inst_of iCrossRefDescr_type
supertype(iSupertype) =     iPrimalCrRef_descr

iEltType inst_of iCrossRefDescr_type
supertype(iEltType) = iPrimalCrRef_descr
```

iMinelt inst_of iAttrDescr_type
supertype(iMinelt) = iPrimalAttr_descr

iMaxelt inst_of iAttrDescr_type
supertype(iMaxelt) = iPrimalAttr_descr

iChangeable inst_of iAttrDescr_type
supertype(iChangeable) = iPrimalAttr_descr

iUnique inst_of iAttrDescr_type
supertype(iUnique) = iPrimalAttr_descr

iPropDescriptors inst_of iCrossRefDescr_type
supertype(iPropDescriptors) = iPrimalCrRef_descr

iOwnPropDescriptors inst_of iComponentDescr_type
supertype(iOwnPropDescriptors) =
iPrimalComp_descr

iOperations inst_of iCrossRefDescr_type
supertype(iOperations) = iPrimalCrRef_descr

iOwnOperations inst_of iCrossRefDescr_type
supertype(iOwnOperations) = iPrimalCrRef_descr

iPrimalType inst_of iCrossRefDescr_type
supertype(iPrimalType) = iPrimalCrRef_descr
■

Definition 30: (Redefinition of propreq, to incorporate relationships)
allbackrefs(io) ≙
{ ioback ∈ IO'' | ioback ∈ refobjs(io) ∪ superobj(io) ∨
∃ ioback2 ∈ allbackrefs(io) •
ioback ∈ refobjs(ioback2) ∪ superobj(ioback2) }

propreq(io,ipd) ≙ let Pn = name(ipd), ieot = elttype(ipd) in
Pn ∈ dom value(io) ∧
if(ieot inst_of iBasicValueType_type) then
(∀ v ∈ get(io,Pn) • v ∈ domain name(ieot))
elseif(ieot inst_of iObjectType_type) then
if(ipd inst_of iReferenceDescr_type) then
(∀ id ∈ get(io,Pn) • id templ_inst_of ieot)
else
(∀ id ∈ get(io,Pn) • id synt_inst_of ieot)
else false ∧

```
        minelt(ipd) ≤ len value(io)(Pn) ≤ maxelt(ipd) ∧ (if(unique(ipd) = true) then
        unqelt(value(io)(Pn)) else true ) ∧
        if(ipd inst_of iCrossRefDescr_type ∨ ipd inst_of iComponentDescr_type) then
                (∀ iob ∈ get(io,Pn) •
                        (¬ iob ∈ allbackrefs(io)) ∧                    --No cycles!
                        (ipd inst_of iCrossRefDescr_type) =>
                                io ∈ get(iob,RefObjs)
                        ∧
                        (ipd inst_of iComponentDescr_type) =>
                                io = singet(iob,SuperObj)
                ) else true

wfprop(io,ipd) ≜
        ∃ ieot ∈ IT • ∃ Pn ∈ PNAMES • Pn = name(ipd) ∧ ieot = elttype(ipd) ∧
        if(ieot inst_of iBasicValueType_type) then
                true
                        --extra requirements are only implemented on objects!
        elseif(ieot inst_of iObjectType_type) then
                if(ipd inst_of iReferenceDescr_type) then
                        true
                else
                        (∀ id ∈ get(io,Pn) • id inst_of ieot)
        else false
∎
```

Definition 31: (operand descriptors)

```
iOperandDescr_type inst_of iMetaType_type
        name(iOperandDescr_type) =      OperandDescr_type
        supertype(iOperandDescr_type) =
                                        iCrossRefDescr_type
        propdescriptors(iOperandDescr_type) =
                                        { iName'', iSupertype, iEltType, iMinelt,
                                        iMaxelt, iChangeable, iUnique, iSort,
                                        iOprVlidty }
        ownpropdescriptors(iOperandDescr_type) =
                                        { iSort, iOprVlidty }
        operations(iOperandDescr_type) =
                                        { }
        ownoperations(iOperandDescr_type) =
                                        { }
        primaltype(iOperandDescr_type) =
                                        iPrimalOprnd_descr

                iSort inst_of iAttributeDescr_type
                        name(iSort) =           Sort
                        supertype(iSort) =      iPrimalAttr_descr
```

```
                                elttype(iSort) =        iSort_type
                                minelt(iSort) =         1
                                maxelt(iSort) =         1
                        iOprVlidty inst_of iAttributeDescr_type
                                name(iOprVlidty) =      OprVlidty
                                supertype(iOprVlidty) =
                                                        iPrimalAttr_descr
                                elttype(iOprVlidty) =   iValidity_type
                                minelt(iOprVlidty) =    1
                                maxelt(iOprVlidty) =    1

iSort_type inst_of iBasicValueType_type
        name(iSort_type) =                              Sort_val
        supertype(iSort_type) =                         iBasicValue_type

domain Sort_val = SORT

iValidity_type inst_of iBasicValueType_type
        name(iValidity_type) =                          Validity_val
        supertype(iValidity_type) =                     iBasicValue_type

domain Validity_val = VALIDITY

IOPD'' = M''(iOperandDescr_type)
IOPD' = M'(iOperandDescr_type)
IOPD = M(iOperandDescr_type)

sort : IOPD' → SORT
sort(iopd) ≙ singet(iopd,Sort)

oprvlidty : IOPD' → VALIDITY
oprvlidty(iopd) ≙ singet(iopd,OprVlidty)

req_opd1 : IOPD' → B
req_opd1(iopd) ≙
        if(sort(iopd) = out) then minelt(iopd) = 0
        else true

iPrimalOprnd_descr inst_of iOperandDescr_type
        name(iPrimalOprnd_descr) =
                                                        PrimalOprnd_descr
        supertype(iPrimalOprnd_descr) =
                                                        iPrimalCrRef_descr
        elttype(iPrimalOprnd_descr) =
                                                        iObject_type
```

```
        minelt(iPrimalOprnd_descr) =
                                        0
        maxelt(iPrimalOprnd_descr) =
                                        ∞
        changeable(iPrimalOprnd_descr) =
                                        true
        unique(iPrimalOprnd_descr) =    false
        sort(iPrimalOprnd_descr) =
                                        in_out
        oprvlidty(iPrimalOprnd_descr) =
                                        invalid
```

■

Definition 32:

```
iOperationDescr_type inst_of iMetaType_type
        propdescriptors(iOperationDescr_type) =
                                { iName, iSupertype, iPropDescriptors,
                                iOwnPropDescriptors, iOperations,
                                iOwnOperations, iOtherPropDescriptors,
                                iOperandDescriptors, iSettingDescriptors }
        ownpropdescriptors(iOperationDescr_type) =
                                { iOtherPropDescriptors,
                                iOperandDescriptors, iSettingDescriptors }
        operations(iOperationDescr_type) =
                                { iDeriveOperations }
        ownoperations(iOperationDescr_type) =
                                { iDeriveProperties' }
        primaltype(iOperationDescr_type) =
                                iRequest_type

                iOtherPropDescriptors inst_of iComponentDescr_type
                        name(iOtherPropDescriptors) =
                                        OtherPropDescriptors
                        supertype(iOtherPropDescriptors) =
                                        iPrimalComp_descr

                        elttype(iOtherPropDescriptors) =
                                        iStructPropDescr_type
                        minelt(iOtherPropDescriptors) =
                                        0
                        maxelt(iOtherPropDescriptors) =
                                        ∞
                        changeable(iOtherPropDescriptors) =
                                        false
                        unique(iOtherPropDescriptors) =
                                        true
```

iOperandDescriptors inst_of iComponentDescr_type
name(iOperandDescriptors) =
OperandDescriptors
supertype(iOperandDescriptors) =
iPrimalComp_descr
elttype(iOperandDescriptors) =
iOperandDescr_type
minelt(iOperandDescriptors) =
0
maxelt(iOperandDescriptors) =
∞
changeable(iOperandDescriptors) =
false
iSettingDescriptors inst_of iComponentDescr_type
name(iSettingDescriptors) =
SettingDescriptors
supertype(iSettingDescriptors) =
iPrimalProp_descr
elttype(iSettingDescriptors) =
iStructPropDescr_type
minelt(iSettingDescriptors) =
0
maxelt(iSettingDescriptors) =
∞
changeable(iSettingDescriptors) =
false

otherpropdescriptors : IOD' → IOPD-set
otherpropdescriptors(iod) ≜ get(iod,OtherPropDescriptors)

oprnddescriptors : IOD' → IOPD-set
oprnddescriptors(iod) ≜ get(iod,OperandDescriptors)

sttngdescriptors : IOD' → ISPD-set
sttngdescriptors(iod) ≜ get(iod,SettingDescriptors)

req_{iod1} : IOD' → B
req_{iod1}(iod) ≜
∀ ispd ∈ sttngdescriptors(iod) •
ispd inst_of iCrossRefDescr_type ∨ ispd inst_of iAttributeDescr_type

req_{iod1} ∈ erequirements(iOperationDescr_type)

iRequest_type inst_of iOperationDescr_type
name(iRequest_type) =
Request_type

iRequest_type subtype_of* Object_type
propdescriptors(iRequest_type) =
{ iSuperObj, iRefObjs }
ownpropdescriptors(iRequest_type) =
{ }
otherpropdescriptors(iRequest_type) =
{ }
oprnddescriptors(iRequest_type) =
{ }
sttngdescriptors(iRequest_type) =
{ }

IRO'' = M''(iRequest_type)
IRO' = M'(iRequest_type)
IRO = M(iRequest_type)

requests : IO'' → IRO''-set
requests(io) ≙
{ ioref ∈ refobjs(io) | ioref templ_inst_of iRequest_type) }

req_{iot5} : IOT' → B
req_{iot5}(iot) ≙
∀ iod ∈ operations(iot) • ∃ ioprd ∈ oprnddescriptors(iod) •
iot subtype_of* elttype(ioprd)

req_{iot5} ∈ erequirements(iObjectType_type)

mk_ownpropdescriptors : IOD' → ISPD'-set
mk_ownpropdescriptors(iod) ≙
otherpropdescriptors(iod) ∪ oprnddescriptors(iod) ∪ sttngdescriptors(iod)

post_DeriveProperties' : IOD' × IOT' × IDOBJ × IDOBJ → B
post_DeriveProperties'(iod, iod', γ, γ') ≙
ownpropdescriptors(iod') = mk_ownpropdescriptors(iod) ∧
propdescriptors(iod') = mk_propdescriptors(iod') ∧
RestObjSameVal(iod, iod', { PropDescriptors, OwnPropDescriptors })

req_{iod2} : IOD' → B
req_{iod2}(iod) ≙
ownpropdescriptors(iod) = mk_ownpropdescriptors(iod) ∧
propdescriptors(iod) = mk_propdescriptors(iod)

req_{iod2} ∈ erequirements(iOperationDescr_type)
■

Definition 33: (Redefinition of propreq, and wfprop to incorporate operand descriptors)

```
propreq(io,ipd) ≜
        ∃ ieot ∈ IT' • ∃ Pn ∈ PNAMES • Pn = name(ipd) ∧ ieot = elttype(ipd) ∧
        Pn ∈ dom value(io) ∧
        if(ieot inst_of iBasicValueType_type) then
                (∀ v ∈ get(io,Pn) • v ∈ domain name(ieot))
        elseif(ieot inst_of iObjectType_type) then
                if(ipd inst_of iOperandDescr_type ∧ oprvldity(ipd) = invalid) then
                        (∀ id ∈ get(io,Pn) • id templ_inst_of ieot)
                elseif(ipd inst_of iReferenceDescr_type) then
                        true
                else
                        (∀ id ∈ get(io,Pn) • id synt_inst_of ieot)
        else false ∧
        minelt(ipd) ≤ len value(io)(Pn) ≤ maxelt(ipd) ∧ (if(unique(ipd) = true) then
        unqelt(value(io)(Pn)) else true ) ∧
        if(ipd inst_of iCrossRefDescr_type ∨ ipd inst_of iComponentDescr_type) then
                (∀ iob ∈ get(io,Pn) •
                        (¬ iob ∈ allbackrefs(io)) ∧
                        (ipd inst_of iCrossRefDescr_type) =>
                                io ∈ get(iob,RefObjs)
                        ∧
                        (ipd inst_of iComponentDescr_type) =>
                                io = singet(iob,SuperObj)
                ) else true

wfprop(io,ipd) ≜
        ∃ ieot ∈ IT • ∃ Pn ∈ PNAMES • Pn = name(ipd) ∧ ieot = elttype(ipd) ∧
        if(ieot inst_of iBasicValueType_type) then
                true
                        --extra requirements are only implemented on objects!
        elseif(ieot inst_of iObjectType_type) then
                if(ipd inst_of iOperandDescr_type ∧ oprvldity(ipd) = invalid) then
                        (∀ id ∈ get(io,Pn) • id templ_inst_of ieot)
                elseif(ipd inst_of iOperandDescr_type ∧ oprvldity(ipd) =
                synt_valid) then
                        (∀ id ∈ get(io,Pn) • id synt_inst_of ieot)
                elseif(ipd inst_of iReferenceDescr_type) then
                        true
                else
                        (∀ id ∈ get(io,Pn) • id inst_of ieot)
        else false
```

■

Definition 34: (subtyping rule on operation descriptions)

$req_{iod3} : IOD' \rightarrow B$

$req_{iod3}(iod_1) \triangleq$ let $iod_2 = supertype(iod_1)$ in

$(iod_2 = iRequest\_type \vee ($

$name(iod_1) = name(iod_2) \wedge ($

$\forall\ iprd_1 \in oprnddescriptors(iod_1) \bullet$

$\exists\ iprd_2 \in propdescriptors(iod_2) \bullet$

$iprd_2\ inst\_of\ iOperandDescr\_type \wedge$

$iprd_1 \leq_{st}^2 iprd_2$

$)))$

$req_{iod3} \in erequirements(iOperationDescr\_type)$

■

Definition 35: (The frozen property)

iObject_type :

ownpropdescriptors(iObject_type) =

{ iFrozen }

iFrozen inst_of iAttributeDescr_type

name(iFrozen) = Frozen

supertype(iFrozen) =

iPrimalAttr_descr

elttype(iFrozen) = iBool_type

minelt(iFrozen) = 1

maxelt(iFrozen) = 1

frozen : IO' → B

frozen(io) ≙ singet(io,Frozen)

■

Lemma 10: (valid objects will be frozen)

∀ io ∈ IO • cond_frozen(io)

To be certified. See def. 36.

cond_frozen : IO' → B

cond_frozen(io) ≙

if(∃ ipd ∈ propdescriptors(type(io)) • ¬changeable(ipd)) then frozen(io) else true

■

Definition 36:

```
set_cond_synt_valid
ext wr io : IO''
pre
post        post_set_cond_synt_valid(io, io)

post_set_cond_synt_valid : IO'' × IO'' → B
post_set_cond_synt_valid(io, io') ≜
            value(io') = value(io) ∧ type(io') = type(io) ∧
            if(synt_wf(io)) then validity(io') = synt_valid
            else validity(io') = invalid
            ∧
            if(validity(io') ≠ validity(io)) then (
                        ∀ ioref ∈ refobjs(io) • ∃ ioref' • post_set_cond_synt_valid(ioref,
                        ioref')
                        --And replace ioref by ioref'; since ioref and ioref' have the same
                        value we don't do this.
                        ∧ ∃ iosuper' •
                        post_set_cond_synt_valid(superobj(io), iosuper')
                        --And replace superobj(io) by iosuper'; here it is easy, but we are
                        consistent
            )

set_cond_valid
ext wr io : IO'
pre         synt_valid(io)
post        post_set_cond_valid(io, io)

post_set_cond_valid : IO' × IO' → B
post_set_cond_valid(io, io') ≜
            value(io') = value(io) ∧ type(io') = type(io) ∧
            if(wf(io)) then (validity(io') = valid ∧ cond_frozen(io'))
            else
                        post_set_cond_valid_props(io, io') ∧
                        if(wf(io')) then (validity(io') = valid ∧ cond_frozen(io'))
                        else validity(io') = validity(io)

post_set_cond_valid_props : IO' × IO' → B
post_set_cond_valid_props(io, io') ≜
            let iot = type(io) in
                        ∀ ipd ∈ propdescriptors(iot) •
                                    post_set_cond_valid_prop(io, io', ipd)
```

post_set_cond_valid_prop : IO' × IO' × ISPD → B
post_set_cond_valid_prop(io, io', ipd) ≙
∃ Pn ∈ PNAMES • Pn = name(ipd) ∧
if(ipd inst_of iComponentDescr_type ∨ ipd inst_of iCrossRefDescr_type) then
if(ipd inst_of iOperandDescr_type ∧ oprvldity(ipd) < valid) then
true
else
(∀ id ∈ get(io, Pn) • ∃ id' ∈ get(io', Pn) •
post_set_cond_valid(id, id')
else true
■

Definition 37: (New, final)
New(iot : IOT) inewo : IO''
ext wr : γ ∈ IDOBJ
Pre:
Post: post_New(iot, γ, γ, inewo)

post_New : IOT × IDOBJ × IDOBJ × IO'' → B
post_New(iot,γ,γ',inewo') ≙
∃ inewo'' ∈ ID | inewo'' ∉ dom γ •
∃ newo' ∈ O | (∀ ispd ∈ propdescriptors(iot) •
value'(newo')(name(ispd)) = []) ∧ γ' = γ ⊕ (inewo'' ⟼ newo') ∧
post_set_cond_valid(inewo'', inewo') ∧ frozen(inewo') = false
■

Definition 38: (Delete)
Delete
ext wr : io : ID
ext wr : γ ∈ IDOBJ
Pre:
Post: post_Delete(γ, γ, io)

post_Delete(γ,γ',io) ≙
γ' = { io } ⩤ γ
■

Definition 39: Lowest level: adding, removing and changing elements
Gencorrect : IO'' × ISPD → B

Gencorrect(io, ispd) ≙
validity(type(io)) = valid ∧
ispd ∈ propdescriptors(type(io)) ∧
(¬changeable(ispd) => ¬frozen(io))

RemoveElt(ispd : ISPD, n : ℕ+) --remove from the nth position
ext wr: io : IO''
ext rd: ev : EV
Pre: pre_RemoveElt(io, ispd, n)
Post: post_RemoveElt(io, io, ispd, n, ev)

pre_RemoveElt : IO'' × ISPD × ℕ → B
pre_RemoveElt(io, ispd, n) ≜ let Pn = name(ispd) in
Gencorrect(io, ispd) ∧ n > 0 ∧ n ≤ len value(io)(Pn)

post_RemoveElt : IO'' × IO'' × ISPD × ℕ × EV → B
post_RemoveElt(io, io', ispd, n, ev) ≜
let Pn = name(ispd) in
∃ seq, seq' ∈ PV •
seq = value(io)(Pn) ∧
post_Seqremove(seq, seq',n, ev) ∧
value(io')(Pn) = seq' ∧
RestObjSameVal(io, io', { Pn })

RemoveElt2(ispd : ISPD)
ext wr: io : IO''
ext rd: ev : EV
Pre: pre_RemoveElt2(io, ispd, ev)
Post: post_RemoveElt2(io, io, ispd, ev)

pre_RemoveElt2 : IO'' × ISPD × EV → B
pre_RemoveElt2(io, ispd, ev) ≜ let Pn = name(ispd) in
Gencorrect(io, ispd) ∧ ev ∈ elems value(io)(Pn)

post_RemoveElt2 : IO'' × IO'' × ISPD × EV → B
post_RemoveElt2(io, io', ispd, ev) ≜
let Pn = name(ispd) in
∃ seq, seq' ∈ PV •
seq = value(io)(Pn) ∧
post_Seqremove2(seq, seq', ev) ∧
value(io')(Pn) = seq' ∧
RestObjSameVal(io, io', { Pn })

AddElt(ispd : ISPD)
ext wr: io : IO''
ext rd: ev : EV
Pre: pre_AddElt(io, ispd, ev)
Post: post_AddElt(io, io, ispd, ev)

```
pre_AddElt : IO'' × ISPD × EV → B
pre_AddElt(io, ispd, ev) ≜ let Pn = name(ispd) in
        Gencorrect(io, ispd) ∧
        (uniqueness(ispd) = unique => ¬(ev ∈ elems value(io)(Pn)))

post_AddElt : IO'' × IO'' × ISPD × EV → B
post_AddElt(io, io', ispd, ev) ≜
let Pn = name(ispd) in
        ∃ seq, seq' ∈ PV •
        seq = value(io)(Pn) ∧
        post_Seqadd(seq, seq', ev) ∧
        value(io')(Pn) = seq' ∧
        RestObjSameVal(io, io', { Pn })

ChangeElt(ispd : ISPD, n : ℕ+)
ext wr: io : IO''
ext rd: ev : EV
Pre:    pre_ChangeElt(io, ispd, n, ev)
Post:   post_ChangeElt(io, io, ispd, n, ev)

pre_ChangeElt : IO'' × ISPD × ℕ × EV → B
pre_ChangeElt(io, ispd, n, ev) ≜ let Pn = name(ispd) in
        Gencorrect(io, ispd) ∧ n > 0 ∧ n ≤ len value(io)(Pn) ∧
        (uniqueness(ispd) = unique => ¬(ev ∈ elems value(io)(Pn)))

post_ChangeElt : IO'' × IO'' × ISPD × ℕ × EV → B
post_ChangeElt(io, io', ispd, n, ev) ≜
let Pn = name(ispd) in
        ∃ seq, seq' ∈ PV •
        seq = value(io)(Pn) ∧
        post_Seqchange(seq, seq',n, ev) ∧
        value(io')(Pn) = seq' ∧
        RestObjSameVal(io, io', { Pn })

RestObjSameVal : IO'' × IO'' × PNAMES-set → B
RestObjSameVal(io, io', Pnset) ≜ ObjCopy(io, io', Pnset) ∧
        io' = io                                        --identity stays the same!

ObjCopy : IO'' × IO'' × PNAMES-set → B
ObjCopy(io, io', Pnset) ≜  let iot = type(io) in
        ∀ ispd ∈ propdescriptors(iot) | name(ispd) ∉ Pnset •
                (value(io')(name(ispd)) = value(io)(name(ispd)))
        ∧ validity(io') = validity(io) ∧ type(io') = type(io)
■
```

Distinguishing between objects and values; testing on element type

Definition 40: (Objects)
RemoveObj(ispd : ISPD)
ext wr: ioindir : IO''
ext rd: iodir : IO''
Pre: pre_RemoveObj(ioindir, ispd, iodir)
Post: post_RemoveObj(ioindir, ioindir, ispd, iodir)

pre_RemoveObj : IO'' × ISPD × IO'' → B
pre_RemoveObj(ioindir, ispd, iodir) ≙
pre_RemoveElt2(ioindir, ispd, iodir)

post_RemoveObj : IO'' × IO'' × ISPD × IO'' → B
post_RemoveObj(ioindir, ioindir', ispd, iodir) ≙
post_RemoveElt2(ioindir, ioindir', ispd, iodir)

AddObj(ispd : ISPD)
ext wr: ioindir : IO''
ext rd: iodir : IO''
Pre: pre_AddObj(ioindir, ispd, iodir)
Post: post_AddObj(ioindir, ioindir, ispd, iodir)

pre_AddObj : IO'' × ISPD × IO'' → B
pre_AddObj(ioindir, ispd, iodir) ≙ let ieot = elttype(ispd) in
pre_AddElt(ioindir, ispd, iodir) ∧
type(iodir) subtype_of* ieot

post_AddObj : IO'' × IO'' × ISPD × IO'' → B
post_AddObj(ioindir, ioindir', ispd, iodir) ≙
post_AddElt(ioindir, ioindir', ispd, iodir)
∎

Definition 41: (Values)
RemoveVal(ispd : ISPD, n : ℕ+)
ext wr: ioindir : IO''
ext rd: val : D
Pre: pre_RemoveVal(ioindir, ispd, n)
Post: post_RemoveVal(ioindir, ioindir, ispd, n, val)

pre_RemoveVal : IO'' × ISPD × ℕ → B
pre_RemoveVal(ioindir, ispd, n) ≙
pre_RemoveElt(ioindir, ispd, n)

post_RemoveVal : IO'' × IO'' × ISPD × ℕ × D → B
post_RemoveVal(ioindir, ioindir', ispd, n, val) ≜
post_RemoveElt(ioindir, ioindir', ispd, n, val)

AddVal(ispd : ISPD)
ext wr: ioindir : IO''
ext rd: val : D
Pre: pre_AddVal(ioindir, ispd, val)
Post: post_AddVal(ioindir, ioindir, ispd, val)

pre_AddVal : IO'' × ISPD × D → B
pre_AddVal(ioindir, ispd, val) ≜ let ieot = elttype(ispd) in
pre_AddElt(ioindir, ispd, val) ∧
val ∈ domain name(ieot)

post_AddVal : IO'' × IO'' × ISPD × D → B
post_AddVal(ioindir, ioindir', ispd, val) ≜
post_AddElt(ioindir, ioindir', ispd, val)

ChangeVal(ispd : ISPD, n : ℕ+)
ext wr: ioindir : IO''
ext rd: val : D
Pre: pre_ChangeVal(ioindir, ispd, n, val)
Post: post_ChangeVal(ioindir, ioindir, ispd, n, val)

pre_ChangeVal : IO'' × ISPD × ℕ × D → B
pre_ChangeVal(ioindir, ispd, n, val) ≜ let ieot = elttype(ispd) in
pre_ChangeElt(ioindir, ispd, n, val) ∧
val ∈ domain name(ieot)

post_ChangeVal : IO'' × IO'' × ISPD × ℕ × D → B
post_ChangeVal(ioindir, ioindir', ispd, n, val) ≜
post_ChangeElt(ioindir, ioindir', ispd, n, val)
■

Carry through the complete change; test the result on syntactic validity

Definition 42: (References)
RemoveRef(ispd : ISPD)
ext wr: ioindir : IO''
ext rd: iodir : IO''
Pre: pre_RemoveRef(ioindir, ispd, iodir)
Post: post_RemoveRef(ioindir, ioindir, ispd, iodir)

pre_RemoveRef : IO'' × ISPD × IO'' → B
pre_RemoveRef(ioindir, ispd, iodir) ≙
pre_RemoveObj(ioindir, ispd, iodir) ∧ ispd inst_of iReferenceDescr_type

post_RemoveRef : IO'' × IO'' × ISPD × IO'' → B
post_RemoveRef(ioindir, ioindir', ispd, iodir) ≙
∃ ioindir'' ∈ IO'' • post_RemoveObj(ioindir, ioindir'', ispd, iodir) ∧
post_set_cond_synt_valid(ioindir'', ioindir')

AddRef(ispd : ISPD)
ext wr: ioindir : IO''
ext rd: iodir : IO''
Pre: pre_AddRef(ioindir, ispd, iodir)
Post: post_AddRef(ioindir, ioindir, ispd, iodir)

pre_AddRef : IO'' × ISPD × IO'' → B
pre_AddRef(ioindir, ispd, iodir) ≙ let ieot = elttype(ispd) in
pre_AddObj(ioindir, ispd, iodir) ∧
ispd inst_of iReferenceDescr_type

post_AddRef : IO'' × IO'' × ISPD × IO'' → B
post_AddRef(ioindir, ioindir', ispd, iodir) ≙
∃ ioindir'' ∈ IO'' • post_AddObj(ioindir, ioindir'', ispd, iodir) ∧
post_set_cond_synt_valid(ioindir'', ioindir')

post_AddRecRef : IO'' × IO'' × ISPD × IO''-set → B
post_AddRecCrossRef(ioindir, ioindir', ispd, ioset) ≙
let io ∈ ioset in
∃ ioindir'' ∈ IO'' • post_AddRef(ioindir, ioindir'', ispd, io) ∧
let ioset' = ioset \ { io } in
if(ioset' = {}) then ioindir' = ioindir''
else post_AddRecRef(ioindir'', ioindir', ispd, ioset')
■

Definition 43: (Attributes)
RemoveAttr(ispd : ISPD, n : ℕ+)
ext wr: ioindir : IO''
ext rd: val : D
Pre: pre_RemoveAttr(ioindir, ispd, n)
Post: post_RemoveAttr(ioindir, ioindir, ispd, n, val)

pre_RemoveAttr : IO'' × ISPD × ℕ → B
pre_RemoveAttr(ioindir, ispd, n) ≙
pre_RemoveVal(ioindir, ispd, n) ∧ ispd inst_of iAttributeDescr_type

post_RemoveAttr : IO'' × IO'' × ISPD × ℕ × D → B
post_RemoveAttr(ioindir, ioindir', ispd, n, val) ≙
∃ ioindir'' ∈ IO'' • post_RemoveVal(ioindir, ioindir'', ispd, n, val) ∧
post_set_cond_synt_valid(ioindir'', ioindir')

AddAttr(ispd : ISPD)
ext wr: ioindir : IO''
ext rd: val : D
Pre: pre_AddAttr(ioindir, ispd, val)
Post: post_AddAttr(ioindir, ioindir, ispd, val)

pre_AddAttr : IO'' × ISPD × D → B
pre_AddAttr(ioindir, ispd, val) ≙ let ieot = elttype(ispd) in
pre_AddVal(ioindir, ispd, val) ∧
ispd inst_of iAttributeDescr_type

post_AddAttr : IO'' × IO'' × ISPD × D → B
post_AddAttr(ioindir, ioindir', ispd, val) ≙
∃ ioindir'' ∈ IO'' • post_AddVal(ioindir, ioindir'', ispd, val) ∧
post_set_cond_synt_valid(ioindir'', ioindir')

post_AddRecAttr : IO'' × IO'' × ISPD × D-set → B
post_AddRecAttr(ioindir, ioindir', ispd, valset) ≙
let val ∈ valset in
∃ ioindir'' ∈ IO'' • post_AddAttr(ioindir, ioindir'', ispd, val) ∧
let valset' = valset \ { val } in
if(valset' = {}) then ioindir' = ioindir''
else post_AddRecAttr(ioindir'', ioindir', ispd, valset')

ChangeAttr(ispd : ISPD, n : ℕ+)
ext wr: ioindir : IO''
ext rd: val : D
Pre: pre_ChangeAttr(ioindir, ispd, n, val)
Post: post_ChangeAttr(ioindir, ioindir, ispd, n, val)

pre_ChangeAttr : IO'' × ISPD × ℕ × D → B
pre_ChangeAttr(ioindir, ispd, n, val) ≙
pre_ChangeVal(ioindir, ispd, n, val) ∧
ispd inst_of iAtrributeDescr_type

post_ChangeAttr : IO'' × IO'' × ISPD × ℕ × D → B
post_ChangeAttr(ioindir, ioindir', ispd, n, val) ≙
∃ ioindir'' ∈ IO'' • post_ChangeVal(ioindir, ioindir'', ispd, n, val) ∧
post_set_cond_synt_valid(ioindir'', ioindir')
■

Definition 44: (Cross-References)
RemoveCrossRef(ispd : ISPD)
ext wr: ioindir : IO''
ext wr: iodir : IO''
Pre: pre_RemoveCrossRef(ioindir, ispd, iodir)
Post: post_RemoveCrossRef(ioindir, ioindir, ispd, iodir, iodir)

pre_RemoveCrossRef : IO'' × ISPD × IO'' → B
pre_RemoveCrossRef(ioindir, ispd, iodir) ≙
pre_RemoveObj(ioindir, ispd, iodir) ∧ ispd inst_of iCrossRefDescr_type

post_RemoveCrossRef : IO'' × IO'' × ISPD × IO'' × IO'' → B
post_RemoveCrossRef(ioindir, ioindir', ispd, iodir, iodir') ≙
∃ ioindir'' ∈ IO'' • post_RemoveObj(ioindir, ioindir'', ispd, iodir') ∧
post_RemoveObj(iodir, iodir', iRefObjs, ioindir') ∧
post_set_cond_synt_valid(ioindir'', ioindir')
--Note that iodir is not tested on syntactic validity; since the property RefObjs allows any type and number of referenced objects, this is not necessary

AddCrossRef(ispd : ISPD)
ext wr: ioindir : IO''
ext wr: iodir : IO''
Pre: pre_AddCrossRef(ioindir, ispd, iodir)
Post: post_AddCrossRef(ioindir, ioindir, ispd, iodir, iodir)

pre_AddCrossRef : IO'' × ISPD × IO'' → B
pre_AddCrossRef(ioindir, ispd, iodir) ≙ let ieot = elttype(ispd) in
pre_AddObj(ioindir, ispd, iodir) ∧ ¬(iodir ∈ allbackrefs(ioindir) ∧
ispd inst_of iCrossRefDescr_type

post_AddCrossRef : IO'' × IO'' × ISPD × IO'' × IO'' → B
post_AddCrossRef(ioindir, ioindir', ispd, iodir, iodir') ≙
∃ ioindir'' ∈ IO'' • post_AddObj(ioindir, ioindir'', ispd, iodir') ∧
post_AddObj(iodir, iodir', iRefObjs, ioindir') ∧
post_set_cond_synt_valid(ioindir'', ioindir')

post_AddRecCrossRef : IO'' × IO'' × ISPD × IO''-set → B
post_AddRecCrossRef(ioindir, ioindir', ispd, ioset) ≙
let io ∈ ioset in
∃ ioindir'', io' ∈ IO'' • post_AddCrossRef(ioindir, ioindir'', ispd, io, io') ∧
let ioset' = ioset \ { io } in
if(ioset' = {}) then ioindir' = ioindir''
else post_AddRecCrossRef(ioindir'', ioindir', ispd, ioset')
■

Definition 45: (Components)

RemoveComp(ispd : ISPD)
ext wr: ioindir : IO''
ext wr: iodir : IO''
Pre: pre_RemoveComp(ioindir, ispd, iodir)
Post: post_RemoveComp(ioindir, ioindir, ispd, iodir, iodir)

pre_RemoveComp : IO'' × ISPD × IO'' → B
pre_RemoveComp(ioindir, ispd, iodir) ≙
pre_RemoveObj(ioindir, ispd, iodir) ∧ ispd inst_of iComponentDescr_type

post_RemoveComp : IO'' × IO'' × ISPD × IO'' × IO'' → B
post_RemoveComp(ioindir, ioindir', ispd, iodir, iodir') ≙
∃ ioindir'' ∈ IO'' • post_RemoveObj(ioindir, ioindir'', ispd, iodir') ∧
post_RemoveObj(iodir, iodir', iSuperObj, ioindir') ∧
post_set_cond_synt_valid(ioindir'', ioindir')

AddComp(ispd : ISPD)
ext wr: ioindir : IO''
ext wr: iodir : IO''
Pre: pre_AddComp(ioindir, ispd, iodir)
Post: post_AddComp(ioindir, ioindir, ispd, iodir, iodir)

pre_AddComp : IO'' × ISPD × IO'' → B
pre_AddComp(ioindir, ispd, iodir) ≙ let ieot = elttype(ispd) in
pre_AddObj(ioindir, ispd, iodir) ∧
--we don't require that superobj is empty. See pre_AddCmpinCntxt
ispd inst_of iComponentDescr_type

post_AddComp : IO'' × IO'' × ISPD × IO'' × IO'' → B
post_AddComp(ioindir, ioindir', ispd, iodir, iodir') ≙
∃ ioindir'' ∈ IO'' • post_AddObj(ioindir, ioindir'', ispd, iodir') ∧
post_AddObj(iodir, iodir', iRefObjs, ioindir') ∧
post_set_cond_synt_valid(ioindir'', ioindir')

post_AddRecComp : IO'' × IO'' × ISPD × IO''-set → B
post_AddRecComp(ioindir, ioindir', ispd, ioset) ≙
let io ∈ ioset in
∃ ioindir'', io' ∈ IO'' • post_AddComp(ioindir, ioindir'', ispd, io, io') ∧
let ioset' = ioset \ { io } in
if(ioset' = {}) then ioindir' = ioindir''
else post_AddRecComp(ioindir'', ioindir', ispd, ioset')

∎

Definition 46: (Value Propagation)

NewComp(iot : IOT, ispd : ISPD) inewo : IO''
ext wr : γ ∈ IDOBJ
ext wr : iosuper ∈ IO''
Pre: pre_NewComp(iot, iosuper, ispd)
Post: post_NewComp(iot, iosuper, iosuper, γ, γ, ispd, inewo)

pre_NewComp : IOT × IO'' × ISPD → B
pre_NewComp(iot, iosuper, ispd) ≙
ispd ∈ propdescriptors(type(iosuper)) ∧ iot subtype_of* elttype(ispd)

post_NewComp : IOT × IO'' × IO'' × IDOBJ × IO'' → B
post_NewComp(iot, iosuper, iosuper`, γ, γ`, ispd, inewo) ≙
∃ inewo`, inewo'' ∈ IO'' • ∃ γ'' ∈ IDOBJ • post_New(iot, γ, γ'', inewo`) ∧
post_Propagate(iosuper, inewo`, inewo'') ∧
post_AddComp(iosuper, iosuper`, ispd, inewo'', inewo)

post_Propagate : IO'' × IO'' × IO'' → B
post_Propagate(iosuper, io, io') ≙
let iosupert = type(iosuper), iot = type(io) in
let props = { ispd ∈ propdescriptors(iot) | (
∃ ispdsuper ∈ propdescriptors(iosupert) •
ispdsuper $\leq_{st}^{2}$ ispd ∧ ¬(ispd inst_of iComponentDescr_type ∨
ispd inst_of iReferenceDescr_type) } in
post_PropagtRecProps(iosuper, props, io, io')

post_PropagtRecProps : IO'' × ISPD-set × IO'' × IO'' → B
post_PropagtRecProps(iosuper, props, io, io')
let ispd ∈ props in
let pn = name(ispd) in
∃ io'' ∈ IO'' •
if(ispd inst_of iAttributeDescr_type) then
let valset = get(iosuper, pn) in
post_AddRecAttr(io, io'', ispd, valset)
elseif(ispd inst_of iCrossRefDescr_type) then
let ioset = get(iosuper, pn) in
post_AddRecCrossRef(io, io'', ispd, ioset)
else false
∧
let props` = props \ { ispd } in
if(props` = { } ) then io` = io''
else post_PropagtRecProps(iosuper, props, io'', io')
∎

Definition 47: (Delete object and its components recursively)

DeleteRec
ext wr : $\gamma \in$ IDOBJ
ext wr : iodir : IO''
Pre: pre_DeleteRec(iodir)
Post: post_DeleteRec(γ, γ, iodir, iodir)

pre_DeleteRec : IO'' $\rightarrow$ B
pre_DeleteRec(iodir) $\triangleq$
let iosuper = iosuper(iodir), iotsuper = type(iosuper) in
iotsuper inst_of iObjectType_type $\wedge$
$\exists$ ipd $\in$ propdescriptors(type(iosuper)) | iodir in value(iosuper)(name(ipd)) $\wedge$
ipd inst_of iComponentDescr_type •
pre_RemoveComp(iosuper,ipd,iodir)
$\wedge$
pre_DeleteRec2(iodir)

pre_DeleteRec2 : IO'' $\rightarrow$ B
pre_DeleteRec2(iodir) $\triangleq$ let iot = type(iodir) in
iot inst_of iObjectType_type $\wedge$
$\forall$ ioref $\in$ refobjs(iodir) •
$\exists$ ipd $\in$ propdescriptors(type(ioref)) | iodir in
value(ioref)(name(ipd)) $\wedge$ ipd inst_of iCrossReferenceDescr_type •
pre_RemoveCrossRef(ioref,ipd,iodir)
$\wedge$
$\forall$ ipd $\in$ propdescriptors(iot) | ipd inst_of iComponentDescr_type • $\forall$ iosub $\in$
get(io,name(ipd)) • pre_DeleteRec2(iosub)

post_DeleteRec : IDOBJ $\times$ IDOBJ $\times$ IO'' $\times$ IO'' $\rightarrow$ B
post_DeleteRec(γ, γ', iodir, iodir') $\triangleq$
let iosuper = iosuper(iodir), iotsuper = type(iosuper) in
iotsuper inst_of iObjectType_type $\wedge$
$\exists$ ipd $\in$ propdescriptors(type(iosuper)) | iodir in value(iosuper)(name(ipd)) $\wedge$
ipd inst_of iComponentDescr_type •
post_RemoveComp(iosuper,ipd,iodir)
$\wedge$
post_DeleteRec2(γ, γ', iodir, iodir')

post_DeleteRec2 : IDOBJ $\times$ IDOBJ $\times$ IO'' $\times$ IO'' $\rightarrow$ B
post_DeleteRec2(γ, γ', iodir, iodir') $\triangleq$
let iot = type(iodir) in
iot inst_of iObjectType_type $\wedge$
$\forall$ ioref $\in$ refobjs(iodir) •
$\exists$ ipd $\in$ propdescriptors(type(ioref)) | iodir in
value(ioref)(name(ipd)) $\wedge$ ipd inst_of iCrossReferenceDescr_type •
post_RemoveCrossRef(ioref,ipd,iodir)

$\wedge$
$\exists$ allobjects $\in$ IO''-set $\bullet$ (AllObjectsRec(iodir, allobjects) $\wedge$
DelGroup(allobjects, γ, γ'))
$\wedge$
$\forall$ ipd $\in$ propdescriptors(iot) | ipd inst_of iCrossReferenceDescr_type $\bullet$ $\forall$ iocr
$\in$ get(io,name(ipd)) $\bullet$ $\exists$ iocr' $\bullet$ post_RemoveObj(iocr, iocr', iRefObjs, iodir)

AllobjectsRec : IO'' $\times$ IO''-set $\rightarrow$ B
AllobjectsRec(io, ioset) $\triangleq$
let iot = type(io) in
io $\in$ ioset $\wedge$ $\forall$ ispd $\in$ propdescriptors(iot) | ispd inst_of
iComponentDescr_type $\bullet$ $\forall$ io' $\in$ get(io, name(ispd)) $\bullet$ $\exists$ ioset' $\bullet$ ioset' $\subset$ ioset
$\wedge$ AllobjectsRec(io', ioset')

DelGroup : IO''-set $\times$ IDOBJ $\times$ IDOBJ $\rightarrow$ B
DelGroup(ioset, γ, γ') $\triangleq$
$\exists$ io $\in$ IO'' $\bullet$ $\exists$ ioset' $\in$ IO''-set $\bullet$ $\exists$ γ'' $\in$ IDOBJ $\bullet$
io $\in$ ioset $\wedge$ post_Delete(γ, γ'', io) $\wedge$ ioset' = ioset \ { io } $\wedge$
DelGroup(ioset', γ'', γ')
■

Chapter 3

The User Interface model

Formalization

Definition 1: (Conceptual objects and data objects)

```
iConceptualObject_type inst_of iObjectType_type
        supertype(iConceptualObject_type) =
                                        iObject_type
ICO'' =   M''(iConceptualObject_type)
ICO' =    M'(iConceptualObject_type)
ICO =     M(iConceptualObject_type)

iDataObject_type inst_of iObjectType_type
        supertype(iDataObject_type) =
                                        iConceptualObject_type
        ownpropdescriptors(iDataObject_type) =
                                        { iContext }
                iContext inst_of iReferenceDescr_type
                        name(iContext) =
                                                Context
                        supertype(iContext) =
                                                iPrimalRef_descr
                        elttype(iContext) =
                                                iDataContextObject_type
                        minelt(iContext) =
                                                0
                        maxelt(iContext) =
                                                1

IDO'' =   M''(iDataObject_type)
IDO' =    M'(iDataObject_type)
IDO =     M(iDataObject_type)

context : IDO' → IDO'
context(ido) ≙ singet(ido,Context)

iDataContextObject_type inst_of iObjectType_type
        supertype(iDataContextObject_type) =
                                        iDataObject_type
        ownpropdescriptors(iDataContextObject_type) =
                                        { iStub, iWorkspace }
```

```
        ownoperations(iDataContextObject_type) =
                                        { iSaveObject }

                iStub inst_of iAttributeDescr_type
                        name(iStub) =
                                                Stub
                        supertype(iStub) =
                                                iPrimalAttr_descr
                        elttype(iStub) =
                                                iBool_type
                        minelt(iStub) =
                                                1
                        maxelt(iStub) =
                                                1
                iWorkspace inst_of iReferenceDescr_type
                        name(iWorkspace) =
                                                Workspace
                        supertype(iWorkspace) =
                                                iPrimalRef_descr
                        elttype(iWorkspace) =
                                                iWorkspaceReq_type
                        minelt(iWorkspace) =
                                                0
                        maxelt(iWorkspace) =
                                                1

IDCO'' =  M''(iDataObject_type)
IDCO' =   M'(iDataObject_type)
IDCO =    M(iDataObject_type)

stub : IDCO' → B
stub(idco) ≙ singet(idco, Stub)

workspace : IDCO' → IWSP''
workspace(idco) ≙
        singet(idco,Workspace)

iRequest_type subtype_of iConceptualObject_type
iType_type subtype_of iDataObject_type
■
```

Definition 2: (Modules and workspaces)

```
iModule_type inst_of iMetaType_type
        name(iModule_type =         Module_type
        supertype(iModule_type) =   iOperationDescr_type
```

```
ownpropdescriptors(iModule_type) =
                                { iOperandDescriptors', iCompDescriptors,
                                iCrossRefDescriptors, iConttypes,
                                iContImpops, iContops, iCRefops }
primaltype(iModule_type) =      iWorkspaceReq_type

        iOperandDescriptors' inst_of iComponentDescr_type
                name(iOperandDescriptors') =
                                OperandDescriptors
                supertype(iOperandDescriptors') =
                                iPrimalComp_descr
                elttype(iOperandDescriptors') =
                                iOperandDescr_type
                minelt(iOperandDescriptors') =
                                0
                maxelt(iOperandDescriptors') =
                                2                                --!!
        iConttypes inst_of iComponentDescr_type
                name(iConttypes) =
                                Conttypes
                supertype(iConttypes) =
                                iPrimalComp_descr
                elttype(iConttypes) =
                                iType_type
                minelt(iConttypes) =
                                0
                maxelt(iConttypes) =
                                ∞
        iContImpops inst_of iComponentDescr_type
                name(iContImpops) =
                                ContImpops
                supertype(iContImpops) =
                                iPrimalComp_descr
                elttype(iContImpops) =
                                iOperationDescr_type
                minelt(iContImpops) =
                                0
                maxelt(iContImpops) =
                                ∞
        iContops inst_of iComponentDescr_type
                name(iContops) =
                                Contops
                supertype(iContops) =
                                iPrimalComp_descr
                elttype(iContops) =
                                iOperationDescr_type
```

```
                                    minelt(iContops) =
                                                        0
                                    maxelt(iContops) =
                                                        ∞
                            iCRefops inst_of iComponentDescr_type
                                    name(iCRefops) =
                                                        CRefops
                                    supertype(iCRefops) =
                                                        iPrimalComp_descr
                                    elttype(iCRefops) =
                                                        iOperationDescr_type
                                    minelt(iCRefops) =
                                                        0
                                    maxelt(iCRefops) =
                                                        ∞

IMD'' =   M''(iModule_type)
IMD' =    M'(iModule_type)
IMD =     M(iModule_type)

conttypes : IMD' → IT-set
conttypes(imd) ≙ get(imd,Conttypes)

contimpops : IMD' → IOD-set
contimpops(imd) ≙ get(imd,ContImpops)

contops : IMD' → IOD-set
contops(imd) ≙ get(imd,Contops)

crefops : IMD' → IOD-set
crefops(imd) ≙ get(imd,CRefops)

iCreatableModule_type inst_of MetaType_type
        iCreatableModule_type subtype_of iModule_type
        operations(iCreatableModule_type) =
                                    { iNewModule }

iWorkspaceReq_type inst_of iModule_type
        name(iWorkspaceReq_type) =
                                    WorkspaceReq_type
        supertype(iWorkspaceReq_type) =
                                    iRequest_type
        ownpropdescriptors(iWorkspaceReq_type) =
                                    { iOwnrequests, iFreeobjects }
        conttypes(iWorkspaceReq_type) =
                                    {}
```

```
contops(iWorkspaceReq_type) =   {}
crefops(iWorkspaceReq_type) =   {}
oprnddescriptors(iWorkspaceReq_type) =
                                { iInspectObject, iNewInspectObject }

        iInspectObject inst_of iOperandDescr_type
                name(iInspectObject) =
                                InspectObject
                supertype(iInspectObject) =
                                iPrimalOprnd_descr
                elttype(iInspectObject) =
                                iDataContextObject_type
                minelt(iInspectObject) =
                                1
                maxelt(iInspectObject) =
                                1
                sort(iInspectObject) =
                                in_out
                oprvlidty(iInspectObject) =
                                invalid
        iNewInspectObject inst_of iOperandDescr_type
                name(iNewInspectObject) =
                                NewInspectObject
                supertype(iNewInspectObject) =
                                iPrimalOprnd_descr
                elttype(iNewInspectObject) =
                                iDataContextObject_type
                minelt(iNewInspectObject) =
                                0
                maxelt(iNewInspectObject) =
                                1
                sort(iNewInspectObject) =
                                out
                oprvlidty(iNewInspectObject) =
                                invalid
        iOwnrequests inst_of iComponentDescr_type
                name(iOwnrequests) =
                                Ownrequests
                supertype(iOwnrequests) =
                                iPrimalComp_descr
                elttype(iOwnrequests) =
                                iRequest_type
                minelt(iOwnrequests) =
                                0
                maxelt(iOwnrequests) =
                                ∞
```

```
                    iFreeobjects inst_of iComponentDescr_type
                                name(iFreeobjects) =  Freeobjects
                                supertype(iFreeobjects) =
                                                    iPrimalComp_descr
                                elttype(iFreeobjects) =
                                                    iObject_type
                                minelt(iFreeobjects) =
                                                    0
                                maxelt(iFreeobjects) =
                                                    ∞

IWSP'' =  M''(iWorkspaceReq_type)
IWSP' =   M'(iWorkspaceReq_type)
IWSP =    M(iWorkspaceReq_type)

freeobjects : IWSP'' → IO''-set
freeobjects(iwsp) ≙
          get(iwsp,FreeObjects)

ownrequests : IWSP'' → IRO''-set
ownrequests(iwsp) ≙
          get(iwsp, Ownrequests)
■
```

Definition 3: (general operations)

```
iStubCopy inst_of iOperationDescr_type
          name(iStubCopy) =
                                        StubCopy
          supertype(iStubCopy) =
                                        iRequest_type
          ownpropdescriptors(iStubCopy) =
                                        { }
          oprnddescriptors(iStubCopy) =
                                        { iInObject, iOutObject }
                    iInObject inst_of iOperandDescr_type
                                name(iInObject) =
                                                    InObject
                                supertype(iInObject) =
                                                    iPrimalOprnd_descr
                                elttype(iInObject) =
                                                    iDataContextObject_type
                                minelt(iInObject) =
                                                    1
                                maxelt(iInObject) =
                                                    1
```

```
                    sort(iInObject) =
                                        in
                    oprvlidty(iInObject) =
                                        invalid
                iOutObject inst_of iOperandDescr_type
                    name(iOutObject) =
                                        OutObject
                    supertype(iOutObject) =
                                        iPrimalOprnd_descr
                    elttype(iOutObject) =
                                        iDataContextObject_type
                    minelt(iOutObject) =
                                        0
                    maxelt(iOutObject) =
                                        1
                    sort(iOutObject) =
                                        in_out
                                --In ClosenSave output object already exists!
                    oprvlidty(iOutObject) =
                                        invalid

iInspectObject inst_of iOperationDescr_type
        name(iInspectObject) =
                                    InspectObject
        supertype(iInspectObject) =
                                    iRequest_type
        ownpropdescriptors(iInspectObject) =
                                    { }
        oprnddescriptors(iInspectObject) =
                                    { iInspObject }

                iInspObject inst_of iOperandDescr_type
                    name(iInspObject) =
                                        InspObject
                    supertype(iInspObject) =
                                        iPrimalOprnd_descr
                    elttype(iInspObject) =
                                        iDataContextObject_type
                    minelt(iInspObject) =
                                        1
                    maxelt(iInspObject) =
                                        1
                    sort(iInspObject) =
                                        in_out
                    oprvlidty(iInspObject) =
                                        invalid
```

```
iSaveObject inst_of iOperationDescr_type
        name(iSaveObject) =
                                        SaveObject
        supertype(iSaveObject) =
                                        iRequest_type
        ownpropdescriptors(iSaveObject) =
                                        { }
        oprnddescriptors(iSaveObject) =
                                        { iInspObject }
                iSavObject inst_of iOperandDescr_type
                        name(iSavObject) =
                                                SavObject
                        supertype(iSavObject) =
                                                iPrimalOprnd_descr
                        elttype(iSavObject) =
                                                iDataContextObject_type
                        minelt(iSavObject) =
                                                1
                        maxelt(iSavObject) =
                                                1
                        sort(iSavObject) =
                                                in_out
                        oprvlidty(iSavObject) =
                                                invalid
```

■

Definition 4: (opening, closing of workspaces)

$pre\_Open : IWSP' \rightarrow B$

$pre\_Open(iwsp) \triangleq$

let inspobj = singet(iwsp, InspectObject) in
stub(InspectObject)

$post\_Open : IWSP \times IWSP \times IDOBJ \times IDOBJ \rightarrow B$

$post\_Open(iwsp, iwsp', \gamma, \gamma') \triangleq$ let inspobj = singet(iwsp, InspectObject) in
let inspobjtype = type(inspobj) in
$\exists$ io, io1, io2, io3, io4, io5, io6, inspobj', inspobj'' $\in$ IO'' •
$\exists \gamma 1, \gamma 2, \gamma 3, \gamma 4, \gamma 5, \gamma 6 \in$ IDOBJ • post_New(inspobjtype, γ, $\gamma 1$, io) $\wedge$
($\exists$ iro, iro1, iro2, iro3 $\in$ IRO •
(post_New(iStubCopy, $\gamma 1$, $\gamma 2$, iro) $\wedge$
post_AddCrossRef(iro, iro1, iInObject, inspobj, inspobj') $\wedge$
post_AddCrossRef(iro1, iro2, iOutObject, io, io1) $\wedge$
post_Execute(iro2, iro3, $\gamma 2$, $\gamma 3$))) $\wedge$
io2 = singet(iro2, OutObject) $\wedge$
($\exists$ iro4, iro5, iro6 $\in$ IRO •
(post_New(iInspectObject, $\gamma 3$, $\gamma 4$, iro4) $\wedge$
post_AddCrossRef(iro4, iro5, iInspObject, io2, io3) $\wedge$

```
                    post_Execute(iro5, iro6, γ4, γ5))) ∧
          io4 = singet(iro5, InspObject) ∧
          post_AddCrossRef(iwsp, iwsp', NewIspectObject, io4, io5) ∧ ¬ stub(io6) ∧
          iwsp' = workspace(io6) ∧ RestObjSameVal(io5, io6, { Stub, Workspace } ) ∧
          ¬ stub(inspobj'') ∧ RestObjSameVal(inspobj', inspobj'') ∧
          post_DeleteRec(γ5, γ6, iro3) ∧
          post_DeleteRec(γ6, γ', iro6)

pre_ClosenSave : IWSP → B
pre_ClosenSave(iwsp) ≙
          let(inspo = singet(iwsp, iInspectObject)) in
                    ¬stub(inspo) ∧
          (∀ io ∈ freeobjects(iwsp) • pre_DeleteRec(io)) ∧
          (∀ iro ∈ ownrequests(iwsp) | iro inst_of iWorkspaceReq_type •
          let(inspos = singet(iro, iInspectObject)) in
                    stub(inspos))

post_ClosenSave : IWSP × IDOBJ × IDOBJ → B
post_ClosenSave(iwsp, γ, γ') ≙
          let inspobj = singet(iwsp, InspectObject), inewinspobj =
          singet(NewInspectObject) in

          ∃ inewinspobj', inewinspobj'', inewinspobj''', inewinspobj'''' ∈ IO'' •
          ∃ inspobj', inspobj'', inspobj''' ∈ IO'' •
          (∃ γ1, γ2, γ3, γ4, γ5, γ6, γ7 ∈ IDOBJ •
          ∃ iro, iro1, iro2 ∈ IRO •
                    (post_New(iSaveObject, γ, γ1, iro) ∧
                    post_AddCrossRef(iro, iro1, iSavObject, inewinspobj, inewinspobj')
                    ∧
                    post_Execute(iro1, iro2, γ1, γ2) ∧
                    inewinspobj'' = singet(iro2, iSavObject))) ∧
          post_Shrink(inewinspobj'', inewinspobj''') ∧
          (∃ iro3, iro4, iro5, iro6 ∈ IRO •
                    (post_New(iStubCopy, γ2, γ3, iro3) ∧
                    post_AddCrossRef(iro3, iro4, iInObject, inewinspobj''',
                    inewinspobj'''') ∧
                    post_AddCrossRef(iro4, iro5, iOutObject, inspobj, inspobj') ∧
                    post_Execute(iro5, iro6, γ3, γ4) ∧
                    inspobj'' = singet(iro6, OutObject))) ∧
          stub(inspobj'') ∧ RestObjSameVal(inspobj', inspobj'', { Stub }) ∧
          post_Delete(γ4, γ5, inewinspobj'''') ∧
          post_DeleteRec(γ5, γ6, iwsp) ∧
          post_DeleteRec(γ6, γ7, iro2) ∧
          post_DeleteRec(γ7, γ', iro6)
```

```
post_Shrink : IO'' × IO'' × IDOBJ × IDOBJ → B
post_Shrink(io, io', γ, γ') ≙ let iotype = type(io) in
        ∀ ispd ∈ propdescriptors(iotype) | ispd inst_of iComponentDescr_type • (
                let allobjects = get(io, name(ispd)) in DelGroup(allobjects, γ, γ'))

pre_ClosenDiscard : IWSP → B
pre_ClosenDiscard(iwsp) ≙
        let(inspo = singet(iwsp, iInspectObject)) in
                ¬stub(inspo)

post_ClosenDiscard : IWSP × IDOBJ × IDOBJ → B
post_ClosenDiscard(iwsp, γ, γ'') ≙
        let newinspobj = singet(iwsp, NewInspectObject), inspobj = singet(iwsp,
        InspectObject) in
        post_DeleteRec(γ, γ', newinsobj, newinspobj) ∧
        post_DeleteRec(γ', γ'', iwsp, iwsp) ∧ ∃ inspobj' • (stub(inspobj') ∧
        RestObjSameVal(inspobj', inspobj, { Stub }))
■
```

Definition 5: (The object: Root_module which owns all modules and to which the module iModule_module may be applied)

```
iEditableModule_type inst_of iMetaType_type
        name(iEditableModule_type) =        EditableModule_type
        supertype(iEditableModule_type) =
                                            iModule_type
        ownoperations(iEditableModule_type) =
                                            { iModule_module }

iRoot_module inst_of iEditableModule_type
        name(iRoot_module) =                Root_module
        supertype(iRoot_module) =           iWorkspaceReq_type
        oprnddescriptors(iRoot_module) =
                                            { iInspectObject', iNewInspectObject' }
        conttypes(iRoot_module) =
                                            { iFolder_type, iDataContextObject_type }
        contimpops(iRoot_module) =          { iAddToFolder, iRmFromFolder,
                                            iNewFolder, iModule_module }
        contops(iRoot_module) =             { Top_module, iStubCopy, iInspectObject,
                                            iSaveObject }
        crefops =                           { }
                iInspectObject' inst_of iOperandDescr_type
                        name(iInspectObject') =
                                                    InspectObject'
                        supertype(iInspectObject') =
                                                    iPrimalOprnd_descr
```

```
                                elttype(iInspectObject') =
                                                iFolder_type
                                minelt(iInspectObject') =
                                                1
                                maxelt(iInspectObject') =
                                                1
                                sort(iInspectObject') =
                                                in_out
                                oprvlidty(iInspectObject') =
                                                invalid
                        iNewInspectObject' inst_of iOperandDescr_type
                                name(iNewInspectObject') =
                                                NewInspectObject'
                                supertype(iNewInspectObject') =
                                                iPrimalOprnd_descr
                                elttype(iNewInspectObject') =
                                                iFolder_type                    --!!
                                minelt(iNewInspectObject') =
                                                0
                                maxelt(iNewInspectObject') =
                                                1
                                sort(iNewInspectObject') =
                                                out
                                oprvlidty(iNewInspectObject') =
                                                invalid

iRoot_workspace inst_of iRoot_module
        get(iRoot_workspace,InspectObject) = iRoot_object

iRoot_object inst_of iFolder_type
        { iRoot_module } ⊂ get(iRoot_object,Contents)

iFolder_type inst_of iObjectType_type
        name(iFolder_type) = Folder_type
        supertype(iFolder_type) =       iDataContextObject_type
        ownpropdescriptors(iFolder_type) =
                                        { iContents, iDate }
        ownoperation =          { iAddToFolder, iRmFromFolder, iNewFolder }
        iContents inst_of iComponentDescr_type
                name(iContents) =               Contents
                elttype(iContents) =            iDataContextObject_type
                minelt(iContents) =             0
                maxelt(iContents) =             ∞
```

```
iAddToFolder inst_of iOperationDescr_type
        name(iAddToFolder) =
                                        AddToFolder
        supertype(iAddToFolder) =
                                        iRequest_type
        ownpropdescriptors(iAddToFolder) =
                                        { }
        oprnddescriptors(iAddToFolder) =
                                        { iIndirFolder, iDirContObj }
        sttingdescriptors(iAddToFolder) =
                                        { }

                iIndirFolder inst_of iOperandDescr_type
                        name(iIndirFolder) =
                                                IndirFolder
                        supertype(iIndirFolder) =
                                                iPrimalOprnd_descr
                        elttype(iIndirFolder) =
                                                iFolder_type
                        minelt(iIndirFolder) =
                                                1
                        maxelt(iIndirFolder) =
                                                1
                        sort(iIndirFolder) =
                                                in_out
                        oprvlidty(iIndirFolder) =
                                                invalid
                iDirContObj inst_of iOperandDescr_type
                        name(iDirContObj) =
                                                DirContObj
                        supertype(iDirContObj) =
                                                iPrimalOprnd_descr
                        elttype(iDirContObj) =
                                                iDataContextObject_type
                        minelt(iDirContObj) =
                                                0
                        maxelt(iDirContObj) =
                                                1
                        sort(iDirContObj) =
                                                out
                        oprvlidty(iDirContObj) =
                                                invalid

iRmFromFolder inst_of iOperationDescr_type
        name(iRmFromFolder) =
                                                RmFromFolder
```

```
supertype(iRmFromFolder) =
                                        iRequest_type
ownpropdescriptors(iRmFromFolder) =
                                        { }
oprnddescriptors(iRmFromFolder) =
                                        { iIndirFolder, iDirContObj }
sttingdescriptors(iRmFromFolder) =
                                        { }
```

Definition 6: (Module, used to edit the whole super-object/ component hierarchy

```
iModule_module inst_of iCreatableModule_type
        name(iModule_module) =          Module_module
        supertype(iModule_module) =     iWorkspaceReq_type
        oprnddescriptors(iModule_module) =
                                        { iInspectObject'', iNewInspectObject'' }
        conttypes(iModule_module) =
                { iType_type, iValType_type, iBasicValType_type,
                iStructPropDescr_type, iReferenceDescr_type, iAttributeDescr_type,
                iCrossRefDescr_type, iComponentDescr_type, iOperandDescr_type,
                iObjectType_type, iCompleteObjectType_type,
                iOperationDescr_type, iModule_type, iEditableModule_type,
                iCreatableModule_type, iMetaType_type }
        contops(iModule_module) =
                { iNewObjectType, iNewOperationDescr, iNewModule,
                iDeriveProperties, iDeriveOperations, ... }

        crefops =                       {}

                iInspectObject'' inst_of iOperandDescr_type
                        name(iInspectObject'') =
                                        InspectObject
                        supertype(iInspectObject'') =
                                        iPrimalOprnd_descr
                        elttype(iInspectObject'') =
                                        iEditableModule_type
                        minelt(iInspectObject'') =
                                        1
                        maxelt(iInspectObject'') =
                                        1
                        sort(iInspectObject'') =
                                        in_out
                        oprvlidty(iInspectObject'') =
                                        invalid
```

```
iNewInspectObject'' inst_of iOperandDescr_type
        name(iNewInspectObject'') =
                NewInspectObject
        supertype(iNewInspectObject'') =
                iPrimalOprnd_descr
        elttype(iNewInspectObject'') =
                iEditableModule_type
        minelt(iNewInspectObject'') =
                0
        maxelt(iNewInspectObject'') =
                1
        sort(iNewInspectObject'') =
                out
        oprvlidty(iNewInspectObject'') =
                invalid
```

■

Definition 7: (The module: Top_module)

```
iTop_module inst_of iCreatableModule_type
        name(iTop_module) =             Top_module
        supertype(iTop_module) =        iWorkspaceReq_type
        oprnddescriptors(iTop_module) =
                                        { }
        conttypes(iTop_module) =
                                        { iObject_type, iConceptualObject_type,
                                        iDataObject_type, iRequest_type,
                                        iWorkspaceReq_type }
        contimpops(iTop_module) =
                                        { iNew }
        contops(iTop_module) =          { iStandardStructChange }
        crefops(iTop_module) =          { }
```

■

Definition 8: (Applicability of Add, Remove, Change, Delete and New)

```
iStandardStructChange inst_of iOperationDescr_type
        name(iStandardStructChange) =
                                        StandardStructChange
        supertype(iStandardStructChange) =
                                        iRequest_type
        ownpropdescriptors(iStandardStructChange) =
                                        { }
        oprnddescriptors(iStandardStructChange) =
                                        { iIndirObject, iDirObject }
        sttingdescriptors(iStandardStructChange) =
                                        { iPropertyDescr }
```

```
iIndirObject inst_of iOperandDescr_type
        name(iIndirObject) =
                IndirObject
        supertype(iIndirObject) =
                iPrimalOprnd_descr
        elttype(iIndirObject) =
                iDataObject_type
        minelt(iIndirObject) =
                1
        maxelt(iIndirObject) =
                1
        sort(iIndirObject) =
                in_out
        oprvlidty(iIndirObject) =
                invalid
iDirObject inst_of iOperandDescr_type
        name(iDirObject) =
                DirObject
        supertype(iDirObject) =
                iPrimalOprnd_descr
        elttype(iDirObject) =
                iϕ
        minelt(iDirObject) =
                0
        maxelt(iDirObject) =
                1
        sort(iDirObject) =
                out
        oprvlidty(iDirObject) =
                invalid
iPropertyDescr inst_of iCrossRefDescr_type
        name(iPropertyDescr) =
                PropertyDescr
        supertype(iPropertyDescr) =
                iPrimalCRef_descr
        elttype(iPropertyDescr) =
                iStructPropDescr_type
        minelt(iPropertyDescr) =
                1
        maxelt(iPropertyDescr) =
                1

iStandardStructChange ∈ ownoperations(iType_type)
```

```
iNew inst_of iOperationDescr_type
        name(iNew) =                    New
        supertype(iNew) =               iRequest_type
        ownpropdescriptors(iNew) =      { }
        oprnddescriptors(iNew) =        { iNewObject }
        sttingdescriptors(iNew) =       { }
                iNewObject inst_of iOperandDescr_type
                        name(iNewObject) =
                                        NewObject
                        supertype(iNewObject) =
                                        iPrimalOprnd_descr
                        elttype(iNewObject) =
                                        iDataObject_type
                        minelt(iNewObject) =
                                        0
                        maxelt(iNewObject) =
                                        1
                        sort(iNewObject) =
                                        out
                        oprvlidty(iNewObject) =
                                        invalid

--New operations below: all implemented generically, but offered as separate operations to
the user

iNewObjectType inst_of iOperationDescr_type
        iNewObjectType subtype_of iNew
        oprnddescriptors(iNewObjectType) =
                                        { iNewObject' }
                iNewObject' inst_of iOperandDescr_type
                        name(iNewObject') =
                                        NewObject
                        supertype(iNewObject') =
                                        iPrimalOprnd_descr
                        elttype(iNewObject') =
                                        iObjectType_type
                        minelt(iNewObject') =
                                        0
                        maxelt(iNewObject') =
                                        1
                        sort(iNewObject') =
                                        out
                        oprvlidty(iNewObject') =
                                        invalid

iNewObjectType ∈ ownoperations(iObjectType_type)
```

```
iNewOperationDescr inst_of iOperationDescr_type
        iNewOperationDescr subtype_of iNew
        oprnddescriptors(iNewOperationDescr) =
                                        { iNewObject'' }

                iNewObject'' inst_of iOperandDescr_type
                        name(iNewObject'') =
                                        NewObject
                        supertype(iNewObject'') =
                                        iPrimalOprnd_descr
                        elttype(iNewObject'') =
                                        iOperationDescr_type
                        minelt(iNewObject'') =
                                        0
                        maxelt(iNewObject'') =
                                        1
                        sort(iNewObject'') =
                                        out
                        oprvlidty(iNewObject'') =
                                        invalid

iNewOperationDescr ∈ ownoperations(iOperationDescr_type)

iNewModule inst_of iOperationDescr_type
        iNewModule subtype_of iNew
        oprnddescriptors(iNewModule) =
                                        { iNewObject''' }
                iNewObject''' inst_of iOperandDescr_type
                        name(iNewObject''') =
                                        NewObject
                        supertype(iNewObject''') =
                                        iPrimalOprnd_descr
                        elttype(iNewObject''') =
                                        iCreatableModule_type
                        minelt(iNewObject''') =
                                        0
                        maxelt(iNewObject''') =
                                        1
                        sort(iNewObject''') =
                                        out
                        oprvlidty(iNewObject''') =
                                        invalid

iNewModule ∈ ownoperations(iCreatableModule_type)
```

```
iNewFolder inst_of iOperationDescr_type
        iNewFolder subtype_of iNew
        oprnddescriptors(iNewFolder) =
                                        { iNewObject'''' }
                iNewObject'''' inst_of iOperandDescr_type
                        name(iNewObject'''') =
                                        NewObject
                        supertype(iNewObject'''') =
                                        iPrimalOprnd_descr
                        elttype(iNewObject'''') =
                                        iFolder_type
                        minelt(iNewObject'''') =
                                        0
                        maxelt(iNewObject'''') =
                                        1
                        sort(iNewObject'''') =
                                        out
                        oprvlidty(iNewObject'''') =
                                        invalid

Remove_Applicable : IDO'' → B
Remove_Applicable(ido) ≙
        ¬stub(ido) ∧ (∃ opr ∈ operations(type(ido)) •
                name(opr) = Remove ∨ name(opr) = StandardStructChange)

Add_Applicable : IDO'' → B
Add_Applicable(ido) ≙
        ¬stub(ido) ∧ (∃ opr ∈ operations(type(ido)) •
                name(opr) = Add ∨ name(opr) = StandardStructChange)

Change_Applicable : IDO'' → B
Change_Applicable(ido) ≙
        ¬stub(ido) ∧ (∃ opr ∈ operations(type(ido)) •
                name(opr) = Change ∨ name(opr) = StandardStructChange)

Delete_Applicable : IDO'' → B
Delete_Applicable(ido) ≙
        ¬stub(ido) ∧ (∃ opr ∈ operations(type(ido)) •
                name(opr) = Delete ∨ name(opr) = StandardStructChange)

New_Applicable : IOT → B
New_Applicable(iot) ≙           iot subtype_of* iDataObject_type ∧
        (∃ opr ∈ operations(iot)) •      name(opr) = New)
■
```

Definition 9: (Cross-References in context)
RemoveCrRfinCntxt(ispd : ISPD)
ext wr: ioindir : IDO''
ext wr: iodir : IDO''
Pre: pre_RemoveCrRfinCntxt(ioindir, ispd, iodir)
Post: post_RemoveCrRfinCntxt(ioindir, ioindir, ispd, iodir, iodir)

pre_RemoveCrRfinCntxt : IDO'' × ISPD × IDO'' → B
pre_RemoveCrRfinCntxt(ioindir, ispd, iodir) ≙
pre_RemoveCrossRef(ioindir, ispd, iodir) ∧ Remove_Applicable(ioindir)

post_RemoveCrRfinCntxt : IDO'' × IDO'' × ISPD × IDO'' × IDO'' → B
post_RemoveCrRfinCntxt(ioindir, ioindir', ispd, iodir, iodir') ≙
post_RemoveCrossRef(ioindir, ioindir', ispd, iodir, iodir')

AddCrRfinCntxt(ispd : ISPD)
ext wr: ioindir : IDO''
ext wr: iodir : IDO''
Pre: pre_AddCrRfinCntxt(ioindir, ispd, iodir)
Post: post_AddCrRfinCntxt(ioindir, ioindir, ispd, iodir, iodir)

pre_AddCrRfinCntxt : IDO'' × ISPD × IDO'' → B
pre_AddCrRfinCntxt(ioindir, ispd, iodir) ≙
pre_AddCrossRef(ioindir, ispd, iodir) ∧ Add_Applicable(ioindir) ∧
if(¬ ispd inst_of iOperandDescr_type) then
context(ioindir) = context(iodir)
else true

post_AddCrRfinCntxt : IDO'' × IDO'' × ISPD × IDO'' × IDO'' → B
post_AddCrRfinCntxt(ioindir, ioindir', ispd, iodir, iodir') ≙
post_AddCrossRef(ioindir, ioindir', ispd, iodir, iodir')
■

Definition 10: (Components in context)
RemoveCmpinCntxt(ispd : ISPD)
ext wr: ioindir : IDO''
ext wr: iodir : IDO''
ext wr: iwsp : IWSP
Pre: pre_RemoveCmpinCntxt(ioindir, iwsp, ispd, iodir)
Post: post_RemoveCmpinCntxt(ioindir, ioindir, iwsp, iwsp, ispd, iodir, iodir)

pre_RemoveCmpinCntxt : IDO'' × IWSP × ISPD × IDO'' → B
pre_RemoveCmpinCntxt(ioindir, iwsp, ispd, iodir) ≙
pre_RemoveComp(ioindir, ispd, iodir) ∧
iwsp = workspace(context(ioindir)) ∧ Remove_Applicable(iodir)

post_RemoveCmpinCntxt : IDO'' × IDO'' IWSP × IWSP × ISPD × IDO'' × IDO'' → B
post_RemoveCmpinCntxt(ioindir, ioindir', iwsp, iwsp', ispd, iodir, iodir') ≙
∃ iodir'' ∈ IDO'' •
∃ iFrObjs ∈ propdescriptors(type(iwsp)) | name(iFrObjs) = FreeObjects •
post_RemoveComp(ioindir, ioindir', ispd, iodir, iodir'') ∧
post_AddComp(iwsp, iwsp', iFrObjs, iodir'', iodir')

AddCmpinCntxt(ispd : ISPD)
ext wr: ioindir : IDO''
ext wr: iodir : IDO''
ext wr: iwsp : IWSP
Pre: pre_AddCmpinCntxt(ioindir, iwsp, ispd, iodir)
Post: post_AddCmpinCntxt(ioindir, ioindir, iwsp, iwsp, ispd, iodir, iodir)

pre_AddCmpinCntxt : IDO'' × IWSP × ISPD × IDO'' → B
pre_AddCmpinCntxt(ioindir, iwsp, ispd, iodir) ≙
pre_AddComp(ioindir, ispd, iodir) ∧ Add_Applicable(ioindir) ∧
iwsp = workspace(context(ioindir)) ∧ iodir ∈ freeobjects(iwsp)

post_AddCmpinCntxt : IDO'' × IDO'' × IWSP × IWSP × ISPD × IDO'' × IDO'' → B
post_AddCmpinCntxt(ioindir, ioindir', iwsp, iwsp', ispd, iodir, iodir') ≙
∃ iodir'' ∈ IDO'' •
∃ iFrObjs ∈ propdescriptors(type(iwsp)) | name(iFrObjs) = FreeObjects •
post_AddComp(ioindir, ioindir', ispd, iodir, iodir'') ∧
post_RemoveComp(iwsp, iwsp', iFrObjs, iodir'', iodir')
■

Definition 11: (Migrating objects from one workspace to another; in general: either from an arbitrary workspace to Root_workspace or from Root_workspace to an arbitrary workspace)

Migrate
ext wr: iwspfrom, iwspto : IWSP
ext wr: iodir : IDO''
Pre: pre_Migrate(iwspfrom, iodir)
Post: post_Migrate(iwspfrom, iwspfrom, iwspto, iwspto, iodir, iodir)

pre_Migrate : IWSP × IDO'' → B
pre_Migrate(iwspfrom, iodir) ≙
iwspfrom = workspace(context(iodir)) ∧ iodir ∈ get(iwspfrom,FreeObjects) ∧
∀ ispd ∈ propdescriptors(iodir) | ispd inst_of iCrossReferenceDescr_type •
get(iodir,name(ispd)) = { } ∧ refobjs(iodir) = { }

post_Migrate: IWSP × IWSP × IWSP × IWSP × IDO'' × IDO'' → B
post_Migrate(iwspfrom, iwspfrom', iwspto, iwspto', iodir, iodir') ≙
∃ iodir'', iodir''' ∈ IDO'' •
∃ $iFrObjs_1$ ∈ propdescriptors(type(iwspfrom)) |
name($iFrObjs_1$) = FreeObjects •
∃ $iFrObjs_2$ ∈ propdescriptors(type(iwspfrom)) |
name($iFrObjs_2$) = FreeObjects •
post_RemoveComp(iwspfrom, iwspfrom', $iFrObjs_1$, iodir, iodir'') ∧
post_AddComp(iwspto, iwspto', $iFrObjs_2$, iodir'', iodir''') ∧
context(iodir') = singet(iwspto', NewIspectObject) ∧
RestObjSameVal(iodir''', iodir', { Context })

■

Definition 12: (Attributes)
RemoveAttrinCntxt(ispd : ISPD, n : ℕ+)
ext wr: ioindir : IDO''
ext rd: val : D
Pre: pre_RemoveAttrinCntxt(ioindir, ispd, n)
Post: post_RemoveAttrinCntxt(ioindir, ioindir, ispd, n, val)

pre_RemoveAttrinCntxt : IDO'' × ISPD × ℕ → B
pre_RemoveAttrinCntxt(ioindir, ispd, n) ≙
pre_RemoveAttr(ioindir, ispd, n) ∧ Remove_Applicable(ioindir)

post_RemoveAttrinCntxt : IDO'' × IDO'' × ISPD × ℕ × D → B
post_RemoveAttrinCntxt(ioindir, ioindir', ispd, n, val) ≙
post_RemoveAttr(ioindir, ioindir', ispd, n, val)

AddAttrinCntxt(ispd : ISPD)
ext wr: ioindir : IDO''
ext rd: val : D
Pre: pre_AddAttrinCntxt(ioindir, ispd, val)
Post: post_AddAttrinCntxt(ioindir, ioindir, ispd, val)

pre_AddAttrinCntxt : IDO'' × ISPD × D → B
pre_AddAttrinCntxt(ioindir, ispd, val) ≙
pre_AddAttr(ioindir, ispd, val) ∧ Add_Applicable(ioindir)

post_AddAttrinCntxt : IDO'' × IDO'' × ISPD × D → B
post_AddAttrinCntxt(ioindir, ioindir', ispd, val) ≙
post_AddAttr(ioindir, ioindir', ispd, val)

ChangeAttrinCntxt(ispd : ISPD, n : ℕ+)
ext wr: ioindir : IDO''
ext rd: val : D
Pre: pre_ChangeAttrinCntxt(ioindir, ispd, n, val)
Post: post_ChangeAttrinCntxt(ioindir, ioindir, ispd, n, val)

pre_ChangeAttrinCntxt : IDO'' × ISPD × ℕ × D → B
pre_ChangeAttrinCntxt(ioindir, ispd, n, val) ≙
pre_ChangeAttr(ioindir, ispd, n, val) ∧ Change_Applicable(ioindir)

post_ChangeAttrinCntxt : IDO'' × IDO'' × ISPD × ℕ × D → B
post_ChangeAttrinCntxt(ioindir, ioindir', ispd, n, val) ≙
post_ChangeAttr(ioindir, ioindir', ispd, n, val)
▪

Definition 13: (Delete in Context)
DeleteinCntxt
ext wr : γ ∈ IDOBJ
ext wr : iodir : IDO''
Pre: pre_DeleteRec(iodir) ∧ Delete_Applicable(iodir)
Post: post_DeleteRec(γ, γ, iodir, iodir)
▪

Definition 14: (New in Context)
NewinCntxt(iot : IOT) inewo : IDO''
ext wr : iwsp ∈ IWSP''
ext wr : γ ∈ IDOBJ
Pre: New_Applicable(iot)
Post: post_NewinCntxt(iot, iwsp, iwsp, γ, γ, inewo)

post_NewinCntxt : IOT × IWSP'' × IWSP'' × IDOBJ × IDOBJ × IO'' → B
post_NewinCntxt(iot, iwsp, iwsp' γ, γ, inewo) ≙
∃ inewoo ∈ IO'' •
∃ iFrObjs ∈ propdescriptors(type(iwsp)) |
name(iFrObjs) = FreeObjects •
post_New(iot, γ, γ, inewoo) ∧
post_AddComp(iwsp, iwsp', iFrObjs, inewoo, inewo)

--For creating new requests NewComp is used
▪

Definition 15: (Execute in Context)
post_ExecuteinCntxt : IRO × IRO × IDOBJ × IDOBJ × IWSP × IWSP → B
post_ExecuteinCntxt(iro, iro', γ, γ', iwsp, iwsp') ≙
let(ido = type(iro)) in
post_Execute(iro, iro', γ, γ') ∧

$\forall$ ioprnd $\in$ propdescriptors(ido) | ioprnd inst_of iOperandDescr_type $\wedge$
sort(ioprnd) = out • $\exists$ ido, ido` $\in$ IDO`` • ido = singet(iro`, name(ioprnd)) $\wedge$
post_AddComp(iwsp, iwsp`, iFrObjs, ido, ido`)

■

Definition 16: (Syntactic Representatives)
iSyntacticObject_type inst_of iObjectType_type
name(iSyntacticObject_type) = SyntacticObject_type
supertype(iSyntacticObject_type) =
iObject_type
ownpropdescriptors(iSyntacticObject_type) =
{ iWorkspace, iObject, iSelectable }
iWorkspace inst_of iReferenceDescr_type
name(iWorkspace) =
Workspace
supertype(iWorkspace) =
iPrimalRef_descr
elttype(iWorkspace) =
iWorkspaceReq_type
minelt(iWorkspace) =
1
maxelt(iWorkspace) =
1
iObject inst_of iReferenceDescr_type
name(iObject) =
Object
supertype(iObject) =
iPrimalRef_descr
elttype(iObject) =
iConceptualObject_type
minelt(iObject) =
1
maxelt(iObject) =
1
iSelectable inst_of iAttributeDescr_type
name(iSelectable) =
Selectable
supertype(iSelectable) =
iPrimalAttr_descr
elttype(iSelectable) =
iBool_type
minelt(iSelectable) =
1
maxelt(iSelectable) =
1

```
ISO'' =     M''(iSyntacticObject_type)
ISO' =      M'(iSyntacticObject_type)
ISO =       M(iSyntacticObject_type)

workspace : ISO → IWSP''
workspace(iso) ≙
            singet(iso,Workspace)

obj : ISO → ICO''
obj(iso) ≙
            singet(iso,Object)

selectable : ISO → B
selectable(iso) ≙
            singet(iso, Selectable)

iSyntConceptualObject_type inst_of iObjectType_type
            name(iSyntConceptualObject_type) =
                                    SyntConceptualObject_type
            supertype(iSyntConceptualObject_type) =
                                    iSyntacticObject_type
            ownpropdescriptors(iSyntConceptualObject_type) =
                                    { iPropdescs }
                  iPropdescs inst_of iReferenceDescr_type
                        name(iPropdescs) =
                                          Propdescs
                        supertype(iPropdescs) =
                                          iPrimalRef_descr
                        elttype(iPropdescs) =
                                          iStructPropDescr_type
                        minelt(iPropdescs) =
                                          0
                        maxelt(iPropdescs) =
                                          ∞
                  iToPropsAssignable inst_of iReferenceDescr_type
                        name(iToPropsAssignable) =
                                          ToPropsAssignable
                        supertype(iToPropsAssignable) =
                                          iPrimalRef_descr
                        elttype(iToPropsAssignable) =
                                          iStructPropDescr_type
                        minelt(iToPropsAssignable) =
                                          0
                        maxelt(iToPropsAssignable) =
                                          ∞
```

```
ISCO'' =  M''(iSyntConceptualObject_type)
ISCO' =   M'(iSyntConceptualObject_type)
ISCO =    M(iSyntConceptualObject_type)

propdescs : ISCO'' → ISPD-set
propdescs(isco) ≙
        get(isco,Propdescs)

iSyntDataObject_type inst_of iObjectType_type
        name(iSyntDataObject_type) =    SyntDataObject_type
        supertype(iSyntDataObject_type) =
                                        iSyntConceptualObject_type
        ownpropdescriptors(iSyntDataObject_type) =
                                        { iObject' }
                iObject' inst_of iReferenceDescr_type
                        name(iObject') =
                                        Object
                        supertype(iObject') =
                                        iPrimalRef_descr
                        elttype(iObject') =
                                        iDataObject_type
                        minelt(iObject') =
                                        1
                        maxelt(iObject') =
                                        1
ISDO'' =  M''(iSyntDataObject_type)
ISDO' =   M'(iSyntDataObject_type)
ISDO =    M(iSyntDataObject_type)

iSyntRequest_type inst_of iObjectType_type
        name(iSyntRequest_type) =       SyntRequest_type
        supertype(iSyntRequest_type) =
                                        iSyntConceptualObject_type
        ownpropdescriptors(iSyntRequest_type) =
                                        { iObject'' }
                iObject'' inst_of iReferenceDescr_type
                        name(iObject'') =
                                        Object
                        supertype(iObject'') =
                                        iPrimalRef_descr
                        elttype(iObject'') =
                                        iRequest_type
                        minelt(iObject'') =
                                        1
                        maxelt(iObject'') =
                                        1
```

```
ISRO'' =  M''(iSyntRequestObject_type)
ISRO' =   M'(iSyntRequestObject_type)
ISRO =    M(iSyntRequestObject_type)

--Separate introduction due to different behavior in protocols / dialogue

iSyntWorkspaceReq_type inst_of iObjectType_type
        name(iSyntWorkspaceReq_type) =
                                        SyntWorkspaceReq_type
        supertype(iSyntWorkspaceReq_type) =
                                        iSyntRequest_type
        ownpropdescriptors(iSyntWorkspaceReq_type) =
                                        { iObject''' }
                iObject''' inst_of iReferenceDescr_type
                        name(iObject''') =
                                                Object
                        supertype(iObject''') =
                                                iPrimalRef_descr
                        elttype(iObject''') =
                                                iWorkspaceReq_type
                        minelt(iObject''') =
                                                1
                        maxelt(iObject''') =
                                                1
--Separate introduction due to different behavior in protocols / dialogue

iSyntOperation_type inst_of iObjectType_type
        name(iSyntOperation_type) =     SyntOperation_type
        supertype(iSyntOperation_type) =
                                        iSyntacticObject_type
        ownpropdescriptors(iSyntOperation_type) =
                                        { iObject'''' }
                iObject'''' inst_of iReferenceDescr_type
                        name(iObject'''') =
                                                Object
                        supertype(iObject'''') =
                                                iPrimalRef_descr
                        elttype(iObject'''') =
                                                iOperationDescr_type
                        minelt(iObject'''') =
                                                1
                        maxelt(iObject'''') =
                                                1
--Only one syntactic representative for both operations and modules: instantiation of these
two follows the same dialogue
```

ISOD'' = M''(iSyntOperation_type)
ISOD' = M'(iSyntOperation_type)
ISOD = M(iSyntOperation_type)
■

Definition 17: (Some functions used in semantic feedback handling)
syntapplicopr : ISDO'' × IAO → ISOD-set
syntapplicopr(isdo, iaosearch) ≙ let ido = obj(isdo) in
{ isod ∈ syntobjs(iaosearch) | obj(isod) ∈ operations(type(ido)) }

syntobjoftype : IOT × ISDO''-set → ISDO''-set
syntobjoftype(iot, isdosearchset) ≙
{ isdo ∈ isdiosearchset | type(obj(isdo)) subtype_of* iot }

syntobjassignable : ISCO'' × ISPD × ISDO''-set → ISDO''-set
syntobjassignable(isco, ispd, isdosearchset) ≙
let ico = obj(isco), ielttype = elttype(ispd), ipn = name(ispd) in
if(ispd inst_of iAttributeDescr_type) then { } else
{ isdoo ∈ syntobjoftype(ielttype, isdosearchset) | ¬(obj(isdoo) ∈ get(ico,ipn))
}
--All objects of appropriate type which aren't already referenced. This incorporates uniqueness

syntobjrelassignable : ISDO'' × ISPD × ISDO''-set → ISDO''-set
--objects assignable either as component or as cross-referent
syntobjrelassignable(isdo, ispd, isdosearchset) ≙
let ido = obj(isdo) in
let iwsp = request(context(ido)) in
{ isdoo ∈ syntobjassignable(isdo, ispd, isdosearchset) | workspace(isdoo) = iwsp ∧
(if(ispd inst_of iComponentDescr_type) then
obj(isdoo) ∈ freeobjects(iwsp)
else true)
}

post_syntobjassignee : ISDO'' × ISDO''-set × ISDO''-set → B
post_syntobjassignee(isdo, isdosearchset, isdoset') ≙
let(ido = obj(isdo)) in
∀ iscoo ∈ isdosearchset | workspace(iscoo) = workspace(isdo) ∧
add_Applicable(iscoo) • ∃ iscoo' ∈ ISDO'' • (
let ispdset = { ispd ∈ propdescs(iscoo) | ispd inst_of iComponentDescr_type ∧
changeable(ispd) ∧ type(ido) subtype_of* elttype(ispd) } in
post_AddRecRef(iscoo, iscoo', iToPropsAssignable, ispdset) ∧
if(¬ ispdset = { }) then iscoo' ∈ isdoset'
)
■

Definition 18: (Dialogue manager, structure definition)

```
DIALOGSTATESET: SYMBOL-set
DIALOGSTATESET =
        { ANYTHING, ASSIGN_OBJECT, SELECT_OPERATION,
        SELECT_OBJECT }

iDialogStateSet inst_of iBasicValueType_type
        name(iDialogStateSet) = DialogStateSet
        supertype(iDialogStateSet) = BasicValue_type

domain DialogStateSet = DIALOGSTATESET

iDialogueManager_type inst_of iObjectType_type
        name(iDialogueManager_type) =
                                        DialogueManager_type
        supertype(iDialogueManager_type) =
                                        iObject_type
        ownpropdescriptors(iDialogueManager_type) =
                                        { iStatus, iAllDataObjects,
                                        iAllOperationReqs, iAllWorkspaceReqs,
                                        iAllOperations, iToObjectsAssignable,
                                        iObjectsSelectable, iOperationsSelectable,
                                        iIndirObject, iPropDesc, iDirObject }
                iStatus inst_of iAttributeDescr_type
                        name(iStatus) =
                                        Status
                        supertype(iStatus) =
                                        iPrimalAttr_descr
                        elttype(iStatus) =
                                        iDialogStateSet
                        minelt(iStatus) =
                                        0
                        maxelt(iStatus) =
                                        ∞
                iAllDataObjects inst_of iReferenceDescr_type
                        name(iAllDataObjects) =
                                        AllDataObjects
                        supertype(iAllDataObjects) =
                                        iPrimalRef_descr
                        elttype(iAllDataObjects) =
                                        iSyntacticDataObject_type
                        minelt(iAllDataObjects) =
                                        0
                        maxelt(iAllDataObjects) =
                                        ∞
```

```
iAllOperationReqs inst_of iReferenceDescr_type
        name(iAllOperationReqs) =
                                AllOperationReqs
        supertype(iAllOperationReqs) =
                                iPrimalRef_descr
        elttype(iAllOperationReqs) =
                                iSyntRequest_type
        minelt(iAllOperationReqs) =
                                0
        maxelt(iAllOperationReqs) =
                                ∞
iAllWorkspaceReqs inst_of iReferenceDescr_type
        name(iAllWorkspaceReqs) =
                                AllWorkspaceReqs
        supertype(iAllWorkspaceReqs) =
                                iPrimalRef_descr
        elttype(iAllWorkspaceReqs) =
                                iSyntWorkspaceReq_type
        minelt(iAllWorkspaceReqs) =
                                0
        maxelt(iAllWorkspaceReqs) =
                                ∞
iAllOperations inst_of iReferenceDescr_type
        name(iAllOperations) =
                                AllOperations
        supertype(iAllOperations) =
                                iPrimalRef_descr
        elttype(iAllOperations) =
                                iSyntOperation_type
        minelt(iAllOperations) =
                                0
        maxelt(iAllOperations) =
                                ∞
iToObjectsAssignable inst_of iReferenceDescr_type
        name(iToObjectsAssignable) =
                                ToObjectsAssignable
        supertype(iToObjectsAssignable) =
                                iPrimalRef_descr
        elttype(iToObjectsAssignable) =
                                iSyntDataObject_type
        minelt(iToObjectsAssignable) =
                                0
        maxelt(iToObjectsAssignable) =
                                ∞
```

```
iObjectsSelectable inst_of iReferenceDescr_type
        name(iObjectsSelectable) =
                ObjectsSelectable
        supertype(iObjectsSelectable) =
                iPrimalRef_descr
        elttype(iObjectsSelectable) =
                iSyntDataObject_type
        minelt(iObjectsSelectable) =
                0
        maxelt(iObjectsSelectable) =
                ∞
iOperationsSelectable inst_of iReferenceDescr_type
        name(iOperationsSelectable) =
                OperationsSelectable
        supertype(iOperationsSelectable) =
                iPrimalRef_descr
        elttype(iOperationsSelectable) =
                iSyntOperation_type
        minelt(iOperationsSelectable) =
                0
        maxelt(iOperationsSelectable) =
                ∞
iIndirObject inst_of iReferenceDescr_type
        name(iIndirObject) =
                IndirObject
        supertype(iIndirObject) =
                iPrimalRef_descr
        elttype(iIndirObject) =
                iConceptualObject_type
        minelt(iIndirObject) =
                0
        maxelt(iIndirObject) =
                1
iPropDesc inst_of iReferenceDescr_type
        name(iPropDesc) =
                PropDesc
        supertype(iPropDesc) =
                iPrimalRef_descr
        elttype(iPropDesc) =
                iStructPropDescr_type
        minelt(iPropDesc) =
                0
        maxelt(iPropDesc) =
                1
```

```
                    iDirObject inst_of iReferenceDescr_type
                                name(iDirObject) =
                                                    DirObject
                                supertype(iDirObject) =
                                                    iPrimalRef_descr
                                elttype(iDirObject) =
                                                    iDataObject_type
                                minelt(iDirObject) =
                                                    0
                                maxelt(iDirObject) =
                                                    1

IDM'' = M''(iDialogueManager_type)
IDM' = M'(iDialogueManager_type)
IDM = M(iDialogueManager_type)

status : IDM' → DIALOGSTATESET
status(idm) ≙
        singet(idm, Status)

indirobject : IDM' → ICO''
indirobject(idm) ≙
        singet(idm, IndirObject)

propdesc : IDM' → ISPD
propdesc(idm) ≙
        singet(idm, PropDesc)

dirobject : IDM' → IDO
dirobject(idm) ≙
        singet(idm, DirObject)

iDialogueManager inst_of iDialogueManager_type
■
```

Definition 19: (Dialogue Manager, some transition functions)

post_Reset : IDM × IDM → B

post_Reset(idm, idm') ≙ idm' = idm ∧
∀ isdo ∈ get(idm, ObjectsSelectable) • (∃ isdo' • ¬selectable(isdo') ⊙ RestObjSameVal(isdo, isdo', { Selectable })) ∧
∀ iso ∈ get(idm, OperationsSelectable) • (∃ iso' • ¬selectable(iso') ⊙ RestObjSameVal(iso, iso', { Selectable })) ∧
∀ isdo ∈ get(idm, ToObjectsAssignable) • (∃ isdo' • get(isdo', ToPropsAssignable) = { } ⊙ RestObjSameVal(isdo, isdo', { ToPropsAssignable })) ∧
get(idm', ToObjectsAssignable) = { } ∧
get(idm', ObjectsSelectable) = { } ∧
get(idm', OperationsSelectable) = { } ∧
indirobject(idm') = Nil ∧
propdesc(idm') = Nil ∧
dirobject(idm') = Nil ∧
status(idm') = ANYTHING ∧
RestObjSameVal(idm, idm', { ToObjectsAssignable, ObjectsSelectable, OperationsSelectable, IndirObject, PropDesc, DirObject, Status })

--All properties that are not listed stay the same!

post_ObjectMoved : IDM × IDM × ISDO → B

post_ObjectMoved(idm, idm', isdo) ≙
if(status(idm) = ANYTHING) then (
let iodir = obj(isdo), AllDataObjects = get(idm, AllDataObjects), iwsp = workspace(isdo) in
if(¬(iodir ∈ freeobjects(iwsp))) then (
let ioindir = superobj(iodir), ipropdesc ∈ { ispd ∈ propdescriptors(type(ioindir)) | iodir ∈ get(ioindir, name(ispd)) ∧ ispd inst_of iComponentDescr_type } in
if(pre_RemoveCmpinCntxt(ioindir, iwsp, ipropdesc, iodir)) then(
∃ iwsp' ∈ IWSP • ∃ iodir' ∈ IDO'' • ∃ ioindir' ∈ IDO'' • post_RemoveCmpinCntxt(ioindir, ioindir', iwsp, iwsp', ipropdesc, iodir, iodir') ∧ post_ObjectMoved(idm, idm', isdo))
else(RestObjSameVal(idm, idm', { }))
--do nothing
else(
∃ isdoset' ∈ ISDO''-set • post_syntobjassignee(isdo, AllDataObjects, isdoset') ∧
isdoset' = get(idm', iToObjectsAssignable)) ∧ ASSIGN_OBJECT = status(idm') ∧ iodir = dirobject(idm') ∧ RestObjSameVal(idm, idm', { ToObjectsAssignable, Status, DirObject }))
) else (
if(cancel()) then (∃ idm'' • post_Reset(idm, idm'') ∧
post_ObjectMoved(idm'', idm', isdo))
else(RestObjSameVal(idm, idm', { })))

```
post_ObjectAssigned : IDM × IDM × ISDO × ISPD → B
post_ObjectAssigned(idm, idm', isdoindir, ipropdesc) ≙
if(status(idm) = ASSIGN_OBJECT) then (
        let ioindir = obj(isdoindir), iodir = dirobject(idm), iwsp = workspace(isdoindir)
        in
        if(pre_AddCmpinCntxt(ioindir, iwsp, ipropdesc, iodir)) then(
                ∃ iwsp' ∈ IWSP • ∃ iodir' ∈ IDO'' • ∃ ioindir' ∈ IDO'' •
                post_AddCmpinCntxt(ioindir, ioindir', iwsp, iwsp', ispd, iodir,
                iodir') ∧ post_Reset(idm, idm'))
        else (post_Reset(idm, idm')))
else true                                              --error handling not incorporated

post_ObjectMigrated : IDM × IDM × ISDO × ISRO × ISRO → B
post_ObjectMigrated(idm, idm', isdodir, iswspfrom, iswspto) ≙
if(status(idm) = ASSIGN_OBJECT) then (
        let iodir = obj(isdodir), iwspfrom = obj(iswspfrom), iwspto = obj(iswspto) in
        if(pre_Migrate(iwspfrom, iodir)) then(
                ∃ iwspfrom', iwspto' ∈ IWSP • ∃ iodir' ∈ IDO'' •
                post_Migrate(iwspfrom, iwspfrom', iwspto, iwspto', iodir, iodir') ∧
                idm' = idm
        ) else (post_Reset(idm, idm')))
else true                                              --error handling not incorporated

post_ObjPropSelected : IDM × IDM × ISCO × ISPD → B
post_ObjPropSelected(idm, idm', iscoindir, ispd) ≙
if(status(idm) = ANYTHING) then (
        let ioindir = obj(iscoindir), iwsp = workspace(iscoindir), AllDataObjects =
        get(idm, AllDataObjects) in
        ∃ isdoset' ∈ ISDO''-set •
        if(¬(¬changeable(ispd) => frozen(ioindir))) then
                RestObjSameVal(idm, idm', {})
        else(
        if(ioindir inst_of iRequest_type) then
                isdoset' = syntobjassignable(iscoindir, ispd, AllDataObjects)
        else                                           --inst_of iDataObject_type
                isdoset' = syntobjrelassignable(iscoindir, ispd, AllDataObjects)
        ∧
        (∀ isdo ∈ isdoset' • ∃ isdo' ∈ ISDO'' • selectable(isdo') ∧
        RestObjSameVal(isdo, isdo', { Selectable} )) ∧ status(idm') =
        SELECT_OBJECT ∧ isdoset' = get(idm', ObjectsSelectable) ∧ ioindir =
        indirobject(idm') ∧ ispd = propdesc(idm') ∧
        RestObjSameVal(idm, idm', { Status, ObjectsSelectable, IndirObject, PropDesc
        } )
        )
) else (
```

```
        if(cancel()) then (∃ idm'' • post_Reset(idm, idm'') ∧
                                        post_ObjPropSelected(idm'', idm', iscoindir,
                                        ispd))
        else(RestObjSameVal(idm, idm', {}))
)

post_ObjectSelected : IDM × IDM × ISDO → B
post_ObjectSelected(idm, idm', isdodir) ≙
if(status(idm) = ANYTHING) then (
        let iodir = obj(isdodir), iwsp = workspace(isdodir), AllOperations = get(idm,
        AllOperations) in
        ∃ isodset' ∈ ISOD''-set •
        isodset' = syntapplicopr(isdodir, AllOperations) ∧
        (∀ isod ∈ isodset' • ∃ isod' ∈ ISOD'' • selectable(isod') ∧
        RestObjSameVal(isod, isod', { Selectable} ))
        ∧ status(idm') = SELECT_OPERATION ∧ isodset' = get(idm',
        OperationsSelectable) ∧ iodir = dirobject(idm') ∧ RestObjSameVal(idm, idm',
        { Status, OperationsSelectable, DirObject } )
) elseif(status(idm) = SELECT_OBJECT) then (
        let iodir = obj(isdodir), ioindir = indirobject(idm), ipropdesc = propdesc(idm),
        iwsp = workspace in
        if(ipropdesc inst_of iComponentDescr_type) then(
                if(pre_RemoveCmpinCntxt(ioindir, iwsp, ipropdesc, iodir)) then (
                ∃ ioindir', iodir' ∈ IDO'' • ∃ iwsp' ∈ IWSP •
                post_RemoveCmpinCntxt(ioindir, ioindir', iwsp, iwsp', ipropdesc,
                iodir, iodir') ∧ post_Reset(idm, idm''))
                elseif(pre_AddCmpinCntxt(ioindir, iwsp, ipropdesc, iodir)) then (
                ∃ ioindir', iodir' ∈ IDO'' • ∃ iwsp' ∈ IWSP •
                post_AddCmpinCntxt(ioindir, ioindir', iwsp, iwsp', ipropdesc, iodir,
                iodir') ∧ post_Reset(idm, idm''))
                else(if(cancel()) then (post_Reset(idm, idm'') ∧
                post_ObjectSelected(idm'', idm', isdodir))
                else RestObjSameVal(idm, idm', {} )))
        elseif(ipropdesc inst_of iCrossRefDescr_type) then(
                if(pre_RemoveCrRefinCntxt(ioindir, iwsp, ipropdesc, iodir)) then (
                ∃ ioindir', iodir' ∈ IDO'' • ∃ iwsp' ∈ IWSP •
                post_RemoveCrRefinCntxt(ioindir, ioindir', iwsp, iwsp', ipropdesc,
                iodir, iodir') ∧ post_Reset(idm, idm''))
                elseif(pre_AddCrRefinCntxt(ioindir, iwsp, ipropdesc, iodir)) then (
                ∃ ioindir', iodir' ∈ IDO'' • ∃ iwsp' ∈ IWSP •
                post_AddCrRefinCntxt(ioindir, ioindir', iwsp, iwsp', ipropdesc,
                iodir, iodir') ∧ post_Reset(idm, idm''))
                else(if(cancel()) then (post_Reset(idm, idm'') ∧
                post_ObjectSelected(idm'', idm', isdodir))
                else RestObjSameVal(idm, idm', {} ))
        )
```

```
) else (
        if(cancel()) then (∃ idm'' • post_Reset(idm, idm'') ∧
                                        post_ObjectSelected(idm'', idm', isdodir))
        else(RestObjSameVal(idm, idm', {} ))
)

post_OperationSelected : IDM × IDM × ISOD × IDOBJ × IDOBJ → B
post_OperationSelected(idm, idm', isod, γ, γ') ≙
let iod = obj(isod), iwsp = workspace(isod) in
if(status(idm) = ANYTHING) then (
        ∃ iroo ∈ IRO'' • ∃ iwsp' ∈ IWSP • ∃ idm'' ∈ IDM • ∃ γ'' ∈ IDOBJ •
        post_NewComp(iod, Δ, γ'', iwsp, iwsp', iOwnrequests, iroo ) ∧
        post_AddReqtoSyntLayer(idm, idm'', iroo, iwsp, γ'', γ') ∧ post_Reset(idm'',
        idm''))
elseif(status(idm) = SELECT_OPERATION) then (
        if(¬(isod ∈ get(idm, OperationsSelectable))) then (
                if(cancel()) then (∃ idm'' • post_Reset(idm, idm'') ∧
                                        post_OperationSelected(idm'', idm', isod, γ,
                                        γ'))
                else(RestObjSameVal(idm, idm', {} )))
        else (
        let iodir = dirobject(idm) in
        ∃ iroo, iro ∈ IRO'' • ∃ iwsp' ∈ IWSP • ∃ iodir' ∈ IDO'' • ∃ γ'' ∈ IDOBJ •
        post_NewComp(iod, Δ, γ'', iwsp, iwsp', iOwnrequests, iroo ) ∧
        ∃ ipropdesc ∈ { ispd ∈ propdescriptors(iod) | type(iodir) subtype_of*
        elttype(ispd) ∧ ispd inst_of iOperandDescr_type } • post_AddCrossRef(iroo,
        iro, ispd, iodir, iodir') ∧
        post_AddReqtoSyntLayer(idm, idm'', iro, iwsp, γ'', γ') ∧ post_Reset(idm'',
        idm''))
) else (
        if(cancel()) then (∃ idm'' • post_Reset(idm, idm'') ∧
                                        post_OperationSelected(idm'', idm', isod, γ,
                                        γ'))
                else(RestObjSameVal(idm, idm', {} )))
```

```
post_RequestSubmitted : IDM × IDM × ISRO × IDOBJ × IDOBJ → B
post_RequestSubmitted(idm, idm', isro, γ, γ') ≙
if(¬(status(idm) = ANYTHING)) then
        if(cancel()) then (∃ idm'' • post_Reset(idm, idm'') ∧
                                        post_RequestSubmitted(idm'', idm', isro, γ,
                                        γ'))
                else(RestObjSameVal(idm, idm', {} )))
) else (
        let iro = obj(isro), iwsp = workspace(isro), ido = type(iro) in
        if(¬synt_valid(iro)) then RestObjSameVal(idm, idm', {} )
        else (
        ∃ iro' ∈ IRO' • post_set_cond_valid(iro, iro') ∧
        if(¬valid(iro')) then RestObjSameVal(idm, idm', {} )
        else (∃ idm'', idm''' ∈ IDM • ∃ iro'' ∈ IRO • ∃ iwsp' ∈ IWSP • ∃ γ'', γ''',
        γ'''' ∈ IDOBJ •
        ∀ ioprnd ∈ propdescriptors(ido) | (ioprnd inst_of iOperandDescr_type ∧
        sort(ioprnd) = in_out ) • (∃ ido ∈ IDO'' • (ido = singet(iro', name(ioprnd)) ∧
        post_RmovCompDObjfromSyntLayer(idm, idm'', ido, γ, γ''))) ∧
                        --simplification applied here: in_out operands are single valued
        post_ExecuteinCntxt(iro', iro'', γ'', γ''', iwsp, iwsp') ∧
        ∀ ioprnd' ∈ propdescriptors(ido) | (ioprnd' inst_of iOperandDescr_type ∧
        sort(ioprnd') = out ) • (∃ ido ∈ IDO'' • (ido = singet(iro'', name(ioprnd')) ∧
        post_AddRecDObjtoSyntLayer(idm'', idm''', ido, iwsp, γ''', γ''''))) ∧
        ∀ ioprnd'' ∈ propdescriptors(ido) | (ioprnd'' inst_of iOperandDescr_type ∧
        sort(ioprnd'') = in_out ) • (∃ ido ∈ IDO'' • (ido = singet(iro', name(ioprnd''))
        ∧ post_AddCompDObjtoSyntLayer(idm''', idm', ido, γ'''', γ')))
        ))
)

post_OpenRequested : IDM × IDM × ISRO × IDOBJ × IDOBJ → B
post_OpenRequested(idm, idm', isro, γ, γ') ≙
        let iro = obj(isro), iwsp = workspace(isro), iod = type(iro), AllDataObjects =
        get(idm, AllDataObjects), AllOperations = get(idm, AllOperations),
        AllRequests = get(idm, AllOperationReqs) ∪ get(idm, AllWorkspaceReqs) in
        if(¬synt_valid(iro)) then RestObjSameVal(idm, idm', {} )
        else (
        ∃ iro' ∈ IRO' • post_set_cond_valid(iro, iro') ∧
        if(¬valid(iro')) then RestObjSameVal(idm, idm', {} )
        elseif(¬pre_Open(iro')) then RestObjSameVal(idm, idm', {} )
        else(∃ isro' ∈ ISRO • ∃ γ',γ'' ∈ IDOBJ • ∃ idm'', idm''', idm'''' ∈ IDM •
        post_Open(isro, isro', γ, γ') ∧
        ∃ icontext ∈ IDO'' • icontext = singet(isro', NewInspectObject) ∧
        post_AddRecDObjtoSyntLayer(idm, idm'', icontext, γ', γ'') ∧
        ∃ operations ∈ IOD-set • operations = contimpops(ido) ∪ crefops(ido) ∧
        post_AddGroupOpDesctoSyntLayer(idm'', idm''', operations, iwsp, γ'', γ') ∧
        post_Reset(idm''', idm'''') ∧
```

if(status(idm) = SELECT_OBJECT) then (let ioindir = indirobject(idm), ispd = propdesc(idm) in
let iscoindir ∈ { isco ∈ AllDataObjects ∪ AllOperationReqs | obj(isco) = ioindir } in
post_ObjPropSelected(idm'''', idm', iscoindir, ispd))
elseif(status(idm) = SELECT_OPERATION) then (let iodir = dirobject(idm) in
let isdodir ∈ { isdo ∈ get(idm, AllDataObjects) | obj(isdo) = dirobject(idm) } in
post_ObjectSelected(idm'''', idm', isdodir))
elseif(status(idm) = ASSIGN_OBJECT) then (let iodir = dirobject(idm) in
let isdodir ∈ { isdo ∈ get(idm, AllDataObjects) | obj(isdo) = dirobject(idm) } in
post_ObjectMoved(idm'''', idm', isdodir))
)

■

Definition 20: (Auxiliary functions)

post_AddReqtoSyntLayer : IDM × IDM × IRO × IWSP × IDOBJ × IDOBJ → B
post_AddReqtoSyntLayer(idm, idm', iro, iwsp, γ, γ') ≙
let iod = type(iro) in
∃ isro, isroo ∈ ISRO'' •
if(iro inst_of iWorkspaceReq_type) then (
post_New(iSyntWorkspaceReq_type, γ, γ', isro) ∧ isro ∈ get(idm', AllWorkspaceReqs))
else (
post_New(iSyntRequest_type, γ, γ', isro) ∧ isro ∈ get(idm', AllOperationReqs))
∧
iro = obj(isroo) ∧ iwsp = workspace(isroo) ∧ propdescs(isroo) = { ispd ∈ propdescriptors(iod) | ispd inst_of iCrossRefDescr_type } ∧
RestObjSameVal(isro, isroo, { Object, Workspace, Propdescs })

post_AddDObjtoSyntLayer : IDM × IDM × IDO × IWSP × IDOBJ × IDOBJ → B
post_AddDObjtoSyntLayer(idm, idm', ido, iwsp, γ, γ') ≙
let iot = type(ido) in
∃ isdo ∈ ISDO'' • post_New(iSyntDataObject_type, γ, γ', isdo) ∧ isdo ∈ get(idm', AllDataObjects) ∧
∃ isdo' ∈ ISDO'' •
ido = obj(isdo') ∧ iwsp = workspace(isdo') ∧ propdescs(isdo') = { ispd ∈ propdescriptors(iot) | (¬changeable(ispd) => ¬frozen(ido)) ∧ ¬(ispd inst_of iReferenceDescr_type) } ∧ RestObjSameVal(isdo, isdo', { Workspace, PropDescs })

post_AddRecDObjtoSyntLayer : IDM × IDM × IDO × IWSP × IDOBJ × IDOBJ → B
post_AddRecDObjtoSyntLayer(idm, idm', ido, iwsp, γ, γ') ≙
∃ allobjects ∈ IDO-set • AllobjectsRec(ido, allobjects) ∧
post_AddGroupDObjtoSyntLayer(idm, idm', allobjects, iwsp, γ, γ')

post_AddCompDObjtoSyntLayer :
IDM × IDM × IDO × IWSP × IDOBJ × IDOBJ → B
post_AddCompDObjtoSyntLayer(idm, idm', ido, iwsp, γ, γ') ≙
∃ allobjects, allcomps ∈ IDO-set • AllobjectsRec(ido, allobjects) ∧ allcomps =
allobjects \ { ido } ∧
post_AddGroupDObjtoSyntLayer(idm, idm', allcomps, iwsp, γ, γ')

post_AddGroupDObjtoSyntLayer :
IDM × IDM × IDO-set × IWSP × IDOBJ × IDOBJ → B
post_AddGroupDObjtoSyntLayer(idm, idm', allobjects, iwsp, γ, γ') ≙
∃ ido ∈ allobjects • ∃ idm'' ∈ IDM • ∃ γ'' ∈ IDOBJ •
post_AddDObjtoSyntLayer(idm, idm'', ido, iwsp, γ, γ'') ∧
∃ allobjects' ∈ IDO-set • allobjects' = allobjects \ { ido } ∧
post_AddGroupDObjtoSyntLayer(idm'', idm', allobjects', iwsp, γ'', γ')

post_RmovCompDObjfromSyntLayer : IDM × IDM × IDO × IDOBJ × IDOBJ → B
post_RmovCompDObjfromSyntLayer(idm, idm', ido, γ, γ') ≙
∃ syntobjs ∈ ISO-set • ∃ allobjects, allcomps ∈ IDO-set •
AllobjectsRec(ido, allobjects) ∧ allcomps = allobjects \ { ido } ∧
syntobjs = { iso ∈ get(idm, AllObjects) | obj(iso) ∈ allcomps } ∧
get(idm', AllObjects) = get(idm, AllObjects) \ { syntobjs } ∧
DelGroup(syntobjs, γ, γ')

post_AddOpDesctoSyntLayer : IDM × IDM × IOD × IWSP × IDOBJ × IDOBJ → B
post_AddOpDesctoSyntLayer(idm, idm', iod, iwsp, γ, γ') ≙
∃ isod ∈ ISOD'' • post_New(iSyntOperation_type, γ, γ', isod) ∧ isod ∈
get(idm', AllOperations) ∧
iod = obj(isod) ∧ iwsp = workspace(isod)

post_AddGroupOpDesctoSyntLayer :
IDM × IDM × IOD-set × IWSP × IDOBJ × IDOBJ → B
post_AddGroupOpDesctoSyntLayer(idm, idm', operations, iwsp, γ, γ') ≙
∃ iod ∈ operations • ∃ idm'' ∈ IDM • ∃ γ'' ∈ IDOBJ •
post_AddOpDesctoSyntLayer(idm, idm'', iod, iwsp, γ, γ'') ∧
∃ operations' ∈ IOD-set • operations' = operations \ { iod } ∧
post_AddGroupOpDesctoSyntLayer(idm'', idm', operations', iwsp, γ'', γ')

▪

Chapter 4

Executing operation requests

Formalization

Recapitulation

--Recapitulation of the way operations are linked to types, plus one extra requirement:

req_{ot2} : IOT' $\rightarrow$ B

$req_{ot2}(iot) \triangleq$

$\forall\ iod_1 \in operations(iot) \bullet$

$\neg(\exists\ iod_2 \in operations(iot) \bullet$

$iod_1 \neq iod_2 \wedge name(iod_1) = name(iod_2))$

--Chapter 2, def. 23c

req_{ot4} : IOT' $\rightarrow$ B

$req_{ot4}(iot) \triangleq$

$\forall\ iod_2 \in operations(supertype(iot)) \bullet$

$iod_2 \in operations(iot)$

--Chapter 2, def. 23c

mk_operations : IOT' $\rightarrow$ IOD'-set

mk_operations(iot) $\triangleq$

ownoperations(iot) $\cup$ operations(supertype(iot)) $\setminus$

{ $iopd_1 \in$ ownoperations(iot) | ($\exists\ iopd_2 \in$

operations(supertype(iot)) $\bullet$ $iopd_1$ subtype_of* $iopd_2$) }

--Chapter 2, def. 25

req_{ot6} : IOT' $\rightarrow$ B

$req_{ot6}(iot) \triangleq$

($\exists\ iod_1 \in$ ownoperations(iot) $\bullet\ \exists\ iod_2 \in$ ownoperations(iot) $\bullet$

$iod_1 \neq iod_2 \wedge name(iod_1) = name(iod_2)) \Rightarrow$

($\exists\ iod_3 \in$ operations(iot) $\bullet$ iod_1 subtype_of* $iod_3 \wedge iod_2$ subtype_of* iod_3

$\wedge\ iod_1$ subtype_of* $iod_2 \vee iod_2$ subtype_of* iod_1)

$req_{ot6} \in$ erequirements(iObjectType_type)

-- If two or more similar operations are defined in the property ownoperations, one (each of the one) must be subtype of the other. This is used in the search mechanism.

Definition 1: (Find the lowest operation, subtype of the selected operation (i.e. type(ireq)), directly applicable to the first operand, which is also applicable to the other operands)

best_applicable : IRO × IOD → B
best_applicable(ireq, iod) ≙
let itod = type(ireq) in
let ifrstoprnddesc = singet(itod, OperandDescriptors) in
if(ifrstoprnddes = Nil) then
iod = itod
else (
let maxtype = elttype(ifrstoprnddesc),
frstoprndnam = name(ifrstoprnddesc) in
∃ ifrstoprndtype ∈ IOT •
if(maxelt(ifrstoprnddesc) > 1) then
maxtypeof_eltofset(get(ireq, frstoprndnam), ifrstoprndtype) else
ifrstoprndtype = type(singet(ireq, frstoprndnam))--type of first oprnd
•
∃ iot ∈ IOT • frstoprndtype subtype_of* iot ∧
iot subtype_of* maxtype • iod ∈ ownoperations(iot) ∧ name(iod) =
name(itod) ∧ appl_toothers(ireq, iod) ∧ ¬(∃ iod' ∈
ownoperations(iot) • iod' ≠ iod ∧ iod' subtype_of* iod ∧
appl_toothers(ireq, iod'))
•
¬(∃ iot' ∈ IOT • ifrstoprndtype subtype_of* iot' ∧
iot' subtype_of* iot ∧ iot' ≠ iot ∧ ∃ iod'' ∈ ownoperations(iot') •
name(iod'') = name(itod) ∧ appl_toothers(ireq, iod'') }
)
)

typeof_eltofset : IO''-set × IOT → B
typeof_eltofset(ioset, iot) ≙
∃ io ∈ ioset • iot = type(io)

maxtypeof_eltofset : IO''-set × IOT → B
maxtypeof_eltofset(ioset, iot) ≙
typeof_eltofset(ioset, iot) ∧
¬(∃ iot' ∈ IOT • typeof_eltofset(ioset, iot') ∧
iot subtype_of$^+$ iot')

appl_toothers : IRO × IOD → B
appl_toothers(ireq, iod) ≙
let itod = type(ireq) in
let ifrstoprnddesc = singet(itod, OperandDescriptors) in
∀ ioprnddesc ∈ { iopdd ∈ oprnddescriptors(itod) | iopdd ≠ ifrstoprnddesc } •
let oprndnam = name(ioprnddesc) in
let oprndtype = --type of the actual operand

```
                    if(maxelt(ioprnddesc) > 1) then
                    maxtypeof_eltofset(get(ireq, oprndnam), oprndtype) else
                    type(singet(ireq, oprndnam)),
                iopd ∈ { iopd' ∈ propdescriptors(iod) | name(iopd') =
                name(ioprnddesc) } in
                --The property descriptor with this name for iod
                oprndtype subtype_of* elttype(iopd)
```

■

Definition 2: (Some extra definitions in the kernel schema to allow execution of the request!)

```
iImplObjects ∈ ownpropdescriptors(iConceptualObject_type)
        iImplObjects inst_of iComponentDescr_type
                name(iImplObjects) =            ImplObjects
                supertype(iImplObjects) =       iPrimalComp_descr
                elttype(iImplObjects) =         iObject_type
                --Conceptual objects may serve as implementation object as well
                minelt(iImplObjects) =          0
                --Conceptual object may be the only implementation
                maxelt(iImplObjects) =          ∞
                changeable(iImplObjects) =      true

implobjects : ICO''→ IIO''-set
implobjects(ico) ≙
        get(ico, ImplObjects)

iImplOperations ∈ ownpropdescriptors(iOperationDescr_type)
        iImplOperations inst_of iCrossRefDescr_type
                name(iImplOperations) =         ImplOperations
                supertype(iImplOperations) =    iPrimalCrRef_descr
                elttype(iImplOperations) =      iOperationDescr_type
                --for simplicity: no special subtypes
                minelt(iImplOperations) =       1
                maxelt(iImplOperations) =       ∞
                changeable(iImplOperations) =   false

imploperations : IOD → IOD-set
imploperations(iod) ≙
        get(iod, ImplOperations)

iImplTypes ∈ ownpropdescriptors(iObjectType_type)
                name(iImplTypes) =              ImplTypes
                supertype(iImplTypes) =         iPrimalCrRef_descr
                elttype(iImplTypes) =           iImplObjectType_type
                minelt(iImplTypes) =            0
                maxelt(iImplTypes) =            ∞
```

changeable(iImplTypes) = false

impltypes : IOT → IOT-set
impltypes(iot) ≜
get(iot, ImplTypes)
■

Definition 3: (Schema extentions to incorporate transformation between implementation types, part 1)

iTransformOperationDescr_type inst_of iMetaType_type
name(iTransformOperationDescr_type) =
TransformOperationDescr_type
supertype(iTransformOperationDescr_type) =
iOperationDescr_type
ownpropdescriptors(iTransformOperationDescr_type) =
{ iOperandDescriptors' }

iOperandDescriptors' inst_of iComponentDescr_type
name(iOperandDescriptors') =
OperandDescriptors
supertype(iOperandDescriptors') =
iPrimalComp_descr
elttype(iOperandDescriptors') =
iOperandDescr_type
minelt(iOperandDescriptors') =
2
maxelt(iOperandDescriptors') =
2
changeable(iOperandDescriptors') =
false

ITOD = M(iTransformOperationDescr_type)

iPrimalTransform_descr inst_of iTransformOperationDescr_type
name(iPrimalTransform_descr) =
PrimalTransform_descr
supertype(iPrimalTransform_descr) =
iRequest_type
oprnddescriptors(iPrimalTransform_descr) =
{ iInput, iOutput }

iInput inst_of iOperandDescr_type
name(iInput) = Input
supertype(iInput) = iPrimalOprnd_descr
elttype(iInput) = iObject_type
minelt(iInput) = 1

```
                    maxelt(iInput) =        1
                    sort(iInput) =          in
                    oprvlidty(iInput) =     synt_valid
          iOutput inst_of iOperandDescr_type
                    name(iOutput) =         Output
                    supertype(iOutput) =    iPrimalOprnd_descr
                    elttype(iOutput) =      iObject_type
                    minelt(iOutput) =       0
                    maxelt(iOutput) =       1
                    sort(iOutput) =         out
                    oprvlidty(iOutput) =    synt_valid

ITRO = M(iPrimalTransform_descr)

inoperand : ITOD' → IOPD
inoperand(itod) ≜
          let iopd ∈ { iopd' ∈ opernddescriptors(itod) |
                    name(iopd') = Input } in
          iopd

intype : ITOD' → IOT
intype(itod) ≜
          elttype(inoperand(itod))

outoperand : ITOD' → IOPD
outoperand(itod) ≜
          let iopd ∈ { iopd' ∈ opernddescriptors(itod) |
                    name(iopd') = Output } in
          iopd

outtype : ITOD' → IOT
outtype(itod) ≜
          elttype(outoperand(itod))

req_tod1 : ITOD' → B
req_tod1(itod) ≜
          intype(itod) ≠ outtype(itod)

inobject : ITRO → IO''
inobject(itro) ≜
          singet(itro, Input)

outobject : ITRO → IO''
outobject(itro) ≜
          singet(itro, Output)
```

TRANSFORMTABLE = (IOT × IOT) $\xrightarrow{m}$ ITOD*
TRANSFORMTABLE ∈ Domains

iTransformtable_type inst_of iBasicValueType_type
name(iTransformtable_type) = Transformtable
supertype(iTransformtable_type) = iBasicValue_type

Transformtable ∈ BVTNAMES
domain Transformtable = TRANSFORMTABLE

iTransformTable ∈ ownpropdescriptors(iObjectType_type)
iTransformTable inst_of iAttributeDescr_type
name(iTransformTable) = TransformTable
supertype(iTransformTable) = iPrimalAttr_descr
elttype(iTransformTable) = iTransformtable_type
minelt(iTransformTable) = 1
maxelt(iTransformTable) = 1
changeable(iTransformTable) = false

transformtable : IOT → TRANSFORMTABLE
transformtable(iot) ≜
singet(iot, TransformTable)

req_{ot7} : IOT* → B
req_{ot7}(iot) ≜
∀ comp_transf ∈ transformtable(iot) •
let in_impl_type = comp_transf[1],
out_impl_type = comp_transf[2],
transf_seq = transformtable(comp_transf) in
∀ i | 2 ≤ i ≤ len transf_seq•
(outtype(transf_seq(i-1)) = intype(transf_seq(i)) ∧
outtype(transf_seq(i-1)) ∈ impltypes(iot))
∧
in_impl_type = intype(transf_seq(1)) ∧
in_impl_type ∈ impltypes(iot) ∧
out_impl_type = outtype(transf_seq(len transf_seq)) ∧
out_impl_type ∈ impltypes(iot)
■

Definition 4: (Extention of the schema to incorporate updates)
iUptodate ∈ ownpropdescriptors(iObject_type)
iUptodate inst_of iAttributeDescr_type
name(iUptodate) = Uptodate
supertype(iUptodate) = iPrimalAttr_descr
elttype(iUptodate) = iBool_type
minelt(iUptodate) = 1

maxelt(iUptodate) = 1
changeable(iUptodate) = true

uptodate : IOT → B
uptodate(iot) ≙
singet(iot, Uptodate)
■

Definition 5: (Exention of the schema to distinguish between the internal representation -conceptual types- and other implementation types)

iImplObjectType_type inst_of iMetaType_type
name(iImplObjectType_type) = ImplObjectType_type
iImplObjectType_type subtype_of* iObjectType_type
ownpropdescriptors(iImplObjectType_type) =
{ }
ownoperations(iImplObjectType_type) =
{ }
primaltype(iImplObjectType_type) =
iImplObject_type

IIOT`` = M``(iImplObjectType_type)
IIOT` = M`(iImplObjectType_type)
IIOT = M(iImplObjectType_type)

iImplObject_type inst_of iImplObjectType_type
name(iImplObject_type) =
ImplObject_type
supertype(iImplObject_type) = iObject_type
ownpropdescriptors(iImplObject_type) =
{ iAddress }
ownoperations(iImplObject_type) =
{ }
iAddress inst_of iComponentDescr_type
name(iAddress) =
Address
supertype(iAddress) =
iPrimalComp_descr
elttype(iAddress) =
iObject_type
minelt(iAddress) =
1
maxelt(iAddress) =
1
changeable(iAddress) =
false

IIO'' = M''(iImplObject_type)
IIO' = M'(iImplObject_type)
IIO = M(iImplObject_type)
■

Definition 6: (Completeness of the transformation table)

req_{iot8} : IT' → B
req_{iot8}(iot) ≙
impltypes(supertype(iot)) ⊂ impltypes(iot)

req_{iot9} : IOT' → B
req_{iot9}(iot) ≙
transformtable(supertype(iot)) ⊂ transformtable(iot)

req_{iot10} : IOT' → B
req_{iot10}(iot) ≙ let transftable = transformtable(iot)
∀ iot_1, iot_2, iot_3 ∈ IOT | ∃ $comp\_transf_1$, $comp\_transf_2$ ∈ dom transftable •
($comp\_transf_1$ [1] = iot_1 ∧ $comp\_transf_1$ [2] = iot_2) ∧ ($comp\_transf_2$ [1] = iot_2 ∧
$comp\_transf_2$ [2] = iot_3) •
∃ $comp\_transf_3$ ∈ dom transftable • ($comp\_transf_3$ [1] = iot_1 ∧ $comp\_transf_3$ [2]
= iot_3)

{ req_{iot8}, req_{iot9}, req_{iot10} } ⊂ erequirements(iObjectType_type)
■

Definition 7: (The complete transformation mechanism)

BEGSTRANSFORMS = IO ⇸ (IOT × IOT)

post_Transform : ICO'' × ICO'' × IOT × IO → B
post_Transform(ico, ico', iiot, γ, γ', iend) ≙
let allimplobjects = implobjects(ico) ∪ ico, icot = type(ico), transformtable =
transformtable(type(ico)) in
if(∃ iio ∈ allimplobjects • uptodate(iio) ∧ outputtypecompatible(type(object),
iiot, transformtable)) then (ico' = ico ∧ γ' = γ ∧ iend = iio)
else (
∃ ibegin ∈ allimplobjects • ∃ comp_transf ∈ dom transformtable •
optimaltransform(ibegin, comp_transf, allimplobjects, transformtable, iiot) ∧
∃ transfseq ∈ rng transformtable • ∃ iend ∈ implobjects(ico') ∪ ico' •
transfseq = transformtable(comp_transf) ∧
transformroute(ico, ico', allimplobjects, ibegin, iend, comp_transf, transfseq, 1,
len transfseq, γ, γ') ∧
if(iiot = type(ico)) then ico' = iend
)

transformroute : ICO'' × ICO'' × IO-set × IO × IO × ITOD* × $\mathbb{N}_1$ × $\mathbb{N}_1$ × IDOBJ × IDOBJ → B

transformroute(ico, ico', allimplobjects, ibegin, iend, transfseq, i, len transfseq, γ, γ') ≙
let thistransf = transfseq(i) in
∃ inext, inext' ∈ IO'' • ∃ γ1, γ2, γ3, γ4 ∈ IDOBJ •
if(∃ iob ∈ allimplobjects • type(iob) = outtype(thistransf)) then (inext = iob ∧ γ1 = γ ∧ ico'' = ico ∧ inext' = inext)
else (post_New(outtype(thistransf), γ, γ1, inext) ∧ post_Addcomp(ico, ico'', iImplObjects, inext, inext'))
∧
∃ implreq, implreq' ∈ IRO • ∃ ibegin', inext'' ∈ IO'' •
post_New(thistransf, γ1, γ2, implreq) ∧ post_AddCrossRef(implreq, implreq', iInput, ibegin, ibegin') ∧ post_AddCrossRef(implreq, implreq', iOutput, inext, inext') ∧
∃ implreq'' ∈ IRO • inext'' ∈ IO'' • postcond(implreq', implreq'', γ2, γ3) ∧ inext'' = outobject(implreq'') ∧ uptodate(inext'') ∧
post_Delete(γ3, γ4, implreq'') ∧
if(i = len) then (iend = inext'' ∧ γ' = γ4)
else transformroute(ico'', ico', allimplobjects, inext'', iend, transfseq, i+1 , len transfseq, γ4, γ')

optimaltransform : IO'' × (IOT × IOT) × IO''-set × TRANSFORMTABLE × IOT → B
optimaltransform(
ibegin, comp_transf, allimplobjects, transformtable, iiot) ≙
∃ comp_transfset ∈ (IOT × IOT)-set • ∃ begstrnsfrmset ∈ BEGSTRANSFORMS •
findcomp_transfs_onoutputtype(comp_transfset, transformtable, iiot) ∧
findbegstrnsfrms(begstrnsfrmset, allimplobjects, comp_transfset) ∧
findshortest_comp_transf(ibegin, comp_transf, begstrnsfrmset)

findcomp_transfs_onoutputtype : (IOT × IOT)-set × TRANSFORMTABLE × IOT → B
findcomp_transfs_onoutputtype(comp_transfset, transformtable, iiot) ≙
∀ comp_transf ∈ comp_transfset • comp_transf ∈ TRANSFORMTABLE ∧
outputtypecompatible(comp_transf [2], iiot, transformtable)

outputtypecompatible : IOT × IOT × TRANSFORMTABLE → B
outputtypecompatible(endcomp_transf, iiot, transformtable) ≙
endcomp_transf subtype_of* iiot ∧
if(¬(endcomp_transf inst_of iImplObjectType_type)) then
¬(∃ comp_transf ∈ dom transformtable • comp_transf [2] subtype_of* iiot ∧
endcomp_transf subtype_of+ comp_transf [2])
else true

findbegstrnsfrms : BEGSTRANSFORMS × IO-set × (IOT × IOT)-set → B
findbegstrnsfrms(begstrnsfrmset, allimplobjects, comp_transfset) ≙
∀ comp_transf ∈ rng begstrnsfrmset • ∃ object ∈ dom begstrnsfrmset •
begstrnsfrmset(object) = comp_transf
comp_transf ∈ comp_transfset ∧ object ∈ { io ∈ allimplobjects | uptodate(io)
} ∧ inputtypecompatible(comp_transf [1], type(object), transformtable)

inputtypecompatible : IOT × IOT × TRANSFORMTABLE → B
inputtypecompatible(begcomp_transf, iiot, transformtable)
iiot subtype_of* begcomp_transf ∧
if(¬(begcomp_transf inst_of iImplObjectType_type)) then
¬(∃ comp_transf ∈ dom transformtable • comp_transf [1] subtype_of*
begcomp_transf ∧ iiot subtype_of* comp_transf [1])
else true

findshortest_comp_transf :
IO × (IOT × IOT) × BEGSTRANSFORMS × TRANSFORMTABLE
findshortest_comp_transf(ibegin, comp_transf, begstrnsfrmset, transformtable)
comp_transf ∈ rng begstrnsfrmset ∧ ibegin ∈ dom begstrnsfrmset ∧
begstrnsfrmset(ibegin) = comp_transf ∧
let len1 = len transformtable(comp_transf) in
¬(∃ comp_transf2 ∈ rng begstrnsfrmset •
let len2 = len transformtable(comp_transf2) in
len2 < len1)
■

Definition 8: (The complete execute mechanism)
postcond : IRO × IRO × IDOBJ × IDOBJ → B
--Description of the post condition after execution of a certain operation

post_Execute : IRO × IRO × IDOBJ × IDOBJ → B
post_Execute(ireq, ireq', γ, γ') ≙
let itod = type(ireq) in
let outpropdescriptors = { iopd ∈ oprnddescriptors(itod) | sort(iopd) = out } in
∃ ibestod ∈ IOD • best_applicable(ireq, ibestod) •
∃ γ'' ∈ IDOBJ • createalloutputobjs(ireq, ireq', outpropdescriptors, γ, γ'') ∧
∃ implopseq ∈ IOD* • implopseq = value(ibestod)(ImplOperations) ∧
execute_all_implops(ireq, ireq', implopseq, 1, len implopseq, γ'', γ')

```
execute_all_implops :
        IRO × IRO × IOD* × ℕ1 × ℕ1 × IDOBJ × IDOBJ → B
execute_all_implops(ireq, ireq', implopseq, i, len, γ, γ')
        let implod = implopseq(i) in
        ∃ γ1, γ2, γ3, γ4, γ5 ∈ IDOBJ •
        ∃ ireq1, ireq2, ireq3, ireq4 ∈ IRO •
        ∃ impl_req, impl_req', impl_req'', impl_req''' ∈ IRO • post_New(implod, γ,
        γ1, impl_req) ∧
        --Add the implementation objects needed as input
        (∀ iopd ∈ oprnddescriptors(implod) | sort(iopd) ≠ out •
                let ieltiot = elttype(iopd) in
                ∀ ico ∈ get(ireq, name(iopd)) • ∃ iend, iend' ∈ IO'' •
                        (∃ ico' ∈ get(ireq1, name(iopd)) •
                                post_Transform(ico, ico', ieltiot, γ1, γ2,
                                iend)) ∧
                                post_AddCrossRef(impl_req, impl_req',
                                iopd, iend, iend')
                        )
        ) ∧
        let allops = { ∀ iopd ∈ oprnddescriptors(implod) | sort(iopd) ≠ out } in
        RestObjSameVal(ireq, ireq1, allops ) ∧
        --Propagate the values of the settings
        let sttngs = sttngdescriptors(implod) in
        post_PropagtRecProps(ireq, sttngs, impl_req, impl_req') ∧
        --Create output implementation objects and make implementation objects
        invalid
        (∀ iopd ∈ oprnddescriptors(implod) | sort(iopd) ≠ in •
                let ieltiot = elttype(iopd) in
                ∀ ico' ∈ get(ireq1, name(iopd)) •
                        (∃ ico'' ∈ get(ireq2, name(iopd)) •
                                ico'' = ico' ∧
                                if(sort(iopd) = out) then
                                        ∃ iionew, iionew', iionew'' ∈ IO
                                        •
                                        post_New(ieltiot, γ2, γ3, iionew)
                                        ∧
                                        post_AddComp(ico', ico'',
                                        iImplObjects, iionew, iionew') ∧
                                        post_AddCrossRef(impl_req',
                                        impl_req'', iopd, iionew',
                                        iionew'')
                                else (
                                        γ3 = γ2 ∧ RestObjSameVal(ico,
                                        ico', {} ) ∧
                                        RestObjSameVal(impl_req',
                                        impl_req'', {} ))
```

```
                                        ∧
                                        ∀ iio ∈ implobjects(ico'') ∪ ico'' • ∃ iio' ∈
                                        IO • ¬uptodate(iio') ∧ RestObjSameVal(iio,
                                        iio', { Uptodate } )
                        )
) ∧
let allops = { ∀ iopd ∈ oprnddescriptors(implod) | sort(iopd) ≠ in } in
RestObjSameVal(ireq1, ireq2, allops ) ∧
postcond(impl_req'', impl_req''', γ3, γ4) ∧
--Update implementation objects
(∀ iopd ∈ oprnddescriptors(implod) | sort(iopd) ≠ in •
        let ieltiot = elttype(iopd) in
        ∀ ico'' ∈ get(ireq2, name(iopd)) •
                (∃ ico''' ∈ get(ireq3, name(iopd)) •
                                ico''' = ico'' ∧
                                ∀ iio ∈ get(impl_req''', name(iopd)) • ∃ iio'
                                ∈ IO'' • uptodate(iio') ∧
                                RestObjSameVal(iio, iio', { Uptodate }) ∧
                                iio' ∈ implobjects(ico''') ∧
                                RestObjSameVal(ico'', ico''', { ImplObjects
                                })
                        )
) ∧
let allops = { ∀ iopd ∈ oprnddescriptors(implod) | sort(iopd) ≠ in } in
RestObjSameVal(ireq2, ireq3, allops ) ∧
--Read the internal representation
(∀ iopd ∈ oprnddescriptors(implod) | sort(iopd) ≠ in •
        ∀ ico''' ∈ get(ireq3, name(iopd)) •
                (∃ ico''' ∈ get(ireq4, name(iopd)) •
                                ico'''' = ico''' ∧
                                let icotype = type(ico''') in
                                post_Transform(ico''', ico'''', icotype, γ4,
                                γ5, ico'''') ∧ uptodate(ico'''')
) ∧
let allops = { ∀ iopd ∈ oprnddescriptors(implod) | sort(iopd) ≠ in } in
RestObjSameVal(ireq3, ireq4, allops ) ∧
if(i = len) then (
        if(synt_valid(ireq4)) then post_set_cond_valid(ireq4, ireq')
        else RestObjSameVal(ireq4, ireq', {})
        ∧ γ' = γ5
)
else
        execute_all_implops(ireq4, ireq', implopseq, i+1, len, γ5, γ')
```

createalloutputobjs : IRO'' × IRO'' × IOPD-set × IDOBJ × IDOBJ → B
createalloutputobjs(ireq, ireq', outpropdescriptors, γ, γ')
∃ iopd ∈ outpropdescriptors • ∃ γ'' ∈ IDOBJ • ∃ outpropdescriptors' ∈ IOPD-set • ∃ inewdo, inewdo' ∈ IDO • ∃ ireq'' ∈ IRO •
let elttype = elttype(iopd) in
post_New(elttype, γ, γ'', inewdo) ∧
post_AddCrossRef(ireq, ireq'', iopd, inewdo, inewdo') ∧ outpropdescriptors' = outpropdescriptors \ { iopd } ∧
if(outpropdescriptors' = {}) then RestObjSameVal(ireq'', ireq', {})
else createalloutputobjs(ireq'', ireq', outpropdescriptors', γ'', γ')

selectimplobj : ICO'' × IOT × IIO'' → B
selectimplobj(ico, iiot, iio) ≙
iio ∈ (impobjects(ico) ∪ { ico }) ∧ type(iio) = iiot

∎

Chapter 5

Application of the YANUS model to the reference problem

Formalization

Definition 1: (Preparations: Metatypes that define special metaproperties)

iAtomicObjectType_type inst_of iMetaType_type
name(iAtomicObjectType_type) =
AtomicObjectType_type
supertype(iAtomicObjectType_type) =
iObjectType_type
ownpropdescriptors(iAtomicObjectType_type) =
{ iAttrDescriptors }
ownoperations(iAtomicObjectType_type) =
{ iDeriveProperties'' }
primaltype(iAtomicObjectType_type) =
iAtomicObject_type

iAttrDescriptors inst_of iComponentDescr_type
name(iAttrDescriptors) =
AttrDescriptors
supertype(iAttrDescriptors) =
iPrimalComp_descr
elttype(iAttrDescriptors) =
iAttributeDescr_type
minelt(iAttrDescriptors) =
0
maxelt(iAttrDescriptors) =
∞
changeable(iAttrDescriptors) =
false
unique(iAttrDescriptors) =
true

iAtomicObjectType_type ∈ conttypes(Module_module)

IAOT'' = M''(iAtomicObjectType_type)
IAOT' = M'(iAtomicObjectType_type)
IAOT = M(iAtomicObjectType_type)

attrdescriptors : IAOT' → IOPD-set
attrdescriptors(iaot) ≙ get(iaot, iAttrDescriptors)

post_DeriveProperties'' : IAOT' × IAOT' × IDOBJ × IDOBJ → B
post_DeriveProperties''(iaot, iaot', γ, γ') ≙
ownpropdescriptors(iaot') = attrdescriptors(iaot) ∧
propdescriptors(iaot') = mk_propdescriptors(iaot') ∧
RestObjSameVal(iaot, iaot', { PropDescriptors, OwnPropDescriptors }) ∧ γ' =
γ

iDeriveProperties'' ∈ contops(Module_module)

req_{iaot1} : IAOT' → B
req_{iaot1}(iaot) ≙
ownpropdescriptors(iaot) = attrdescriptors(iaot) ∧
propdescriptors(iaot) = mk_propdescriptors(iaot)

req_{iaot1} ∈ erequirements(iAtomicObjectType_type)

iWholeObjectType_type inst_of iMetaType_type
name(iWholeObjectType_type) =
WholeObjectType_type
supertype(iWholeObjectType_type) =
iObjectType_type
ownpropdescriptors(iWholeObjectType_type) =
{ iAttrDescriptors, iCompDescriptors }
ownoperations(iWholeObjectType_type) =
{ iDeriveProperties''' }
primaltype(iWholeObjectType_type) =
iWholeObject_type

iCompDescriptors inst_of iComponentDescr_type
name(iCompDescriptors) =
CompDescriptors
supertype(iCompDescriptors) =
iPrimalComp_descr
elttype(iCompDescriptors) =
iComponentDescr_type
minelt(iCompDescriptors) =
0
maxelt(iCompDescriptors) =
∞
changeable(iCompDescriptors) =
false
unique(iCompDescriptors) =
true

iWholeObjectType_type ∈ conttypes(iModule_module)

IWOT'' = M''(iWholeObjectType_type)
IWOT' = M'(iWholeObjectType_type)
IWOT = M(iWholeObjectType_type)

attrdescriptors : IWOT' → IOPD-set
attrdescriptors(iwot) ≙ get(iwot, iAttrDescriptors)

compdescriptors : IWOT' → IOPD-set
compdescriptors(iwot) ≙ get(iwot, iCompDescriptors)

post_DeriveProperties''' : IWOT' × IWOT' × IDOBJ × IDOBJ → B
post_DeriveProperties'''(iwot, iwot', γ, γ') ≙
ownpropdescriptors(iwot') = attrdescriptors(iwot) ∪
compdescriptors(iwot) ∧
propdescriptors(iwot') = mk_propdescriptors(iwot') ∧
RestObjSameVal(iwot, iwot', { PropDescriptors, OwnPropDescriptors }) ∧ γ' =
γ

iDeriveProperties''' ∈ contops(Module_module)

req_{wotI} : IOD' → B
req_{wotI}(iwot) ≙
ownpropdescriptors(iwot) = attrdescriptors(iwot) ∪
compdescriptors(iwot) ∧
propdescriptors(iwot) = mk_propdescriptors(iwot)

iSimpleAtomicObjectType_type inst_of iMetaType_type
name(iSimpleAtomicObjectType_type) =
SimpleAtomicObjectType_type
supertype(iSimpleAtomicObjectType_type) =
iAtomicObjectType_type
ownpropdescriptors(iSimpleAtomicObjectType_type) =
{ iAttrDescriptors' }
ownoperations(iSimpleAtomicObjectType_type) =
{ }
primaltype(iSimpleAtomicObjectType_type) =
iSimpleAtomicObject_type

iAttrDescriptors' inst_of iComponentDescr_type
name(iAttrDescriptors') =
AttrDescriptors
supertype(iAttrDescriptors') =
iAttrDescriptors

```
                                elttype(iAttrDescriptors') =
                                                iAttributeDescr_type
                                minelt(iAttrDescriptors') =
                                                1
                                maxelt(iAttrDescriptors') =
                                                1
                                changeable(iAttrDescriptors') =
                                                false
                                unique(iAttrDescriptors') =
                                                true

iSimpleAtomicObjectType_type ∈ conttypes(iModule_module)

iSimpleWholeObjectType_type inst_of iMetaType_type
        name(iSimpleWholeObjectType_type) =
                                        SimpleWholeObjectType_type
        supertype(iSimpleWholeObjectType_type) =
                                        iWholeObjectType_type
        ownpropdescriptors(iSimpleWholeObjectType_type) =
                                        { iCompDescriptors' }
        ownoperations(iSimpleWholeObjectType_type) =
                                        { }
        primaltype(iSimpleWholeObjectType_type) =
                                        iSimpleWholeObject_type

                iCompDescriptors' inst_of iComponentDescr_type
                        name(iCompDescriptors') =
                                                CompDescriptors
                        supertype(iCompDescriptors') =
                                                iCompDescriptors
                        elttype(iCompDescriptors') =
                                                iComponentDescr_type
                        minelt(iCompDescriptors') =
                                                1
                        maxelt(iCompDescriptors') =
                                                1
                        changeable(iCompDescriptors') =
                                                false
                        unique(iCompDescriptors') =
                                                true

iSimpleWholeObjectType_type ∈ conttypes(iModule_module)

iAtomicObject_type subtype_of iDataObject_type
iAtomicObject_type ∈ conttypes(iTop_module)
iWholeObject_type subtype_of iDataObject_type
```

```
iWholeObject_type ∈ conttypes(iTop_module)
iSimpleAtomicObject_type subtype_of iAtomicObject_type
iSimpleAtomicObject_type ∈ conttypes(iTop_module)
iSimpleWholeObject_type subtype_of iWholeObject_type
iSimpleWholeObject_type ∈ conttypes(iTop_module)

iSimpleAtomicCrossRefDescr_type inst_of iMetaType_type
        name(iSimpleAtomicCrossRefDescr_type) =
                                        SimpleAtomicCrossRefDescr_type
        supertype(iSimpleAtomicCrossRefDescr_type) =
                                        iCrossRefDescr_type
        ownpropdescriptors(iSimpleAtomicCrossRefDescr_type) =
                                        { iEltType' }
        ownoperations(iSimpleAtomicCrossRefDescr_type) =
                                        { }
        primaltype(iSimpleAtomicCrossRefDescr_type) =
                                        iPrimalCrRef_descr
iSimpleAtomicCrossRefDescr_type ∈ conttypes(iTop_module)

iSimpleAtomicComponentDescr_type inst_of iMetaType_type
        name(iSimpleAtomicComponentDescr_type) =
                                        SimpleAtomicComponentDescr_type
        supertype(iSimpleAtomicComponentDescr_type) =
                                        iComponentDescr_type
        ownpropdescriptors(iSimpleAtomicComponentDescr_type) =
                                        { iEltType' }
        ownoperations(iSimpleAtomicComponentDescr_type) =
                                        { }
        primaltype(iSimpleAtomicComponentDescr_type) =
                                        iPrimalComp_descr

                iEltType' inst_of iStructPropDescr_type
                        name(iEltType') =       EltType
                        supertype(iEltType') =
                                                iEltType
                        elttype(iEltType') =    iSimpleAtomicObjectType_type
                        minelt(iEltType') =     1
                        maxelt(iEltType') =     1
                        changeable(iEltType') =
                                                false
                        unique(iEltType') =     true

iSimpleAtomicComponentDescr_type ∈ conttypes(iTop_module)
■
```

Definition 2: (Meta types for defining table and record types)

```
iTableObjectType_type inst_of iMetaType_type
        name(iTableObjectType_type) =
                                        TableObjectType_type
        supertype(iTableObjectType_type) =
                                        iSimpleWholeObjectType_type
        ownpropdescriptors(iTableObjectType_type) =
                                        { }
        ownoperations(iTableObjectType_type) =
                                        { }
        primaltype(iTableObjectType_type) =
                                        iTableObject_type

iTableObjectType_type ∈ conttypes(iModule_module)

iRecordType_type inst_of iMetaType_type
        name(iRecordType_type) =
                                        RecordType_type
        supertype(iRecordType_type) =
                                        iObjectType_type
        ownpropdescriptors(iRecordType_type) =
                                        { iFieldDescriptors, iKeyDescriptors }
        ownoperations(iRecordType_type) =
                                        { iDeriveProperties'''' }
        primaltype(iRecordType_type) =
                                        iRecord_type

                iFieldDescriptors inst_of iComponentDescr_type
                        name(iFieldDescriptors) =
                                                FieldDescriptors
                        supertype(iFieldDescriptors) =
                                                iPrimalComp_descr
                        elttype(iFieldDescriptors) =
                                                iStructPropDescr_type
                        minelt(iFieldDescriptors) =
                                                0
                        maxelt(iFieldDescriptors) =
                                                ∞
                        changeable(iFieldDescriptors) =
                                                true
                        unique(iFieldDescriptors) =
                                                true
                iKeyDescriptors inst_of iComponentDescr_type
                        name(iKeyDescriptors) =
                                                KeyDescriptors
```

supertype(iKeyDescriptors) =
iPrimalComp_descr
elttype(iKeyDescriptors) =
iStructPropDescr_type
minelt(iKeyDescriptors) =
0
maxelt(iKeyDescriptors) =
∞
changeable(iKeyDescriptors) =
true
unique(iKeyDescriptors) =
true

iRecordType_type ∈ conttypes(iModule_module)

IRT'' = M''(iRecordType_type)
IRT' = M'(iRecordType_type)
IRT = M(iRecordType_type)

fielddescriptors : IRT' → IOPD-set
fielddescriptors(irt) ≙ get(irt, iFieldDescriptors)

keydescriptors : IRT' → IOPD-set
keydescriptors(irt) ≙ get(irt, iKeyDescriptors)

post_DeriveProperties'''' : IRT' × IRT' × IDOBJ × IDOBJ → B
post_DeriveProperties''''(irt, irt', γ, γ') ≙
ownpropdescriptors(irt') =
fielddescriptors(irt) ∪ keydescriptors(irt) ∧
propdescriptors(irt') = mk_propdescriptors(irt') ∧
RestObjSameVal(irt, irt', { PropDescriptors, OwnPropDescriptors }) ∧ γ' = γ

iDeriveProperties'''' ∈ contops(iModule_module)

req_{irt} : IRT' → B
req_{irt}(irt) ≙
ownpropdescriptors(irt) =
fielddescriptors(irt) ∪ keydescriptors(irt) ∧
propdescriptors(irt) = mk_propdescriptors(irt)
req_{irt} ∈ erequirements(iRecordType_type)

iOriginalRecordType_type inst_of iMetaType_type
name(iOriginalRecordType_type) =
OriginalRecordType_type
supertype(iOriginalRecordType_type) =
iRecordType_type

```
        ownpropdescriptors(iOriginalRecordType_type) =
                                        { iFieldDescriptors', iKeyDescriptors' }
        ownoperations(iOriginalRecordType_type) =
                                        { }
        primaltype(iOriginalRecordType_type) =
                                        iOriginalRecordObject_type

                iFieldDescriptors' inst_of iComponentDescr_type
                        name(iFieldDescriptors') =
                                        FieldDescriptors
                        supertype(iFieldDescriptors') =
                                        iFieldDescriptors
                        elttype(iFieldDescriptors') =
                                iSimpleAtomicComponentDescr_type
                        minelt(iFieldDescriptors') =
                                        0
                        maxelt(iFieldDescriptors') =
                                        ∞
                        changeable(iFieldDescriptors') =
                                        false
                        unique(iFieldDescriptors') =
                                        true
                iKeyDescriptors' inst_of iComponentDescr_type
                        name(iKeyDescriptors') =
                                        KeyDescriptors'
                        supertype(iKeyDescriptors') =
                                        iKeyDescriptors
                        elttype(iKeyDescriptors') =
                                iSimpleAtomicComponentDescr_type
                        minelt(iKeyDescriptors') =
                                        0
                        maxelt(iKeyDescriptors') =
                                        ∞
                        changeable(iKeyDescriptors') =
                                        false
                        unique(iKeyDescriptors') =
                                        true

iOriginalRecordType_type ∈ conttypes(iModule_module)

iDerivedRecordType_type inst_of iMetaType_type
        name(iDerivedRecordType_type) =
                                DerivedRecordType_type
        supertype(iDerivedRecordType_type) =
                                iRecordType_type
        ownpropdescriptors(iDerivedRecordType_type) =
```

```
                                        { iFieldDescriptors'', iKeyDescriptors'' }
        ownoperations(iDerivedRecordType_type) =
                                        { }
        primaltype(iDerivedRecordType_type) =
                                        iDerivedRecord_type

                iFieldDescriptors'' inst_of iComponentDescr_type
                        name(iFieldDescriptors'') =
                                                FieldDescriptors
                        supertype(iFieldDescriptors'') =
                                                iFieldDescriptors
                        elttype(iFieldDescriptors'') =
                                        iSimpleAtomicCrossRefDescr_type
                        minelt(iFieldDescriptors'') =
                                                0
                        maxelt(iFieldDescriptors'') =
                                                ∞
                        changeable(iFieldDescriptors'') =
                                                false
                        unique(iFieldDescriptors'') =
                                                true
                iKeyDescriptors'' inst_of iComponentDescr_type
                        name(iKeyDescriptors'') =
                                                KeyDescriptors
                        supertype(iKeyDescriptors'') =
                                                iKeyDescriptors
                        elttype(iKeyDescriptors'') =
                                        iSimpleAtomicCrossRefDescr_type
                        minelt(iKeyDescriptors'') =
                                                0
                        maxelt(iKeyDescriptors'') =
                                                ∞
                        changeable(iKeyDescriptors'') =
                                                false
                        unique(iKeyDescriptors'') =
                                                true

iDerivedRecordType_type ∈ conttypes(iModule_module)
■
```

Definition 3: (Primal types for defining tables and records)

```
iTable_module inst_of iCreatableModule_type
        name(iTable_module) =
                                        Table_module
        supertype(iTable_module) =
                                        iWorkspaceReq_type
```

```
        oprnddescriptors(iTable_module) =
                                        { iInspectObject''', iNewInspectObject''' }

                iInspectObject''' inst_of iOperandDescr_type
                        name(iInspectObject''') =
                                        InspectObject
                        supertype(iInspectObject''') =
                                        iPrimalOprnd_descr
                        elttype(iInspectObject''') =
                                        iTableObject_type
                        minelt(iInspectObject''') =
                                        1
                        maxelt(iInspectObject''') =
                                        1
                        sort(iInspectObject''') =
                                        in_out
                        oprvlidty(iInspectObject''') =
                                        invalid
                iNewInspectObject''' inst_of iOperandDescr_type
                        name(iNewInspectObject''') =
                                        NewInspectObject
                        supertype(iNewInspectObject''') =
                                        iPrimalOprnd_descr
                        elttype(iNewInspectObject''') =
                                        iTableObject_type
                        minelt(iNewInspectObject''') =
                                        0
                        maxelt(iNewInspectObject''') =
                                        1
                        sort(iNewInspectObject''') =
                                        out
                        oprvlidty(iNewInspectObject''') =
                                        invalid

iTableObject_type inst_of iTableObjectType_type
        name(iTableObject_type) =
                                TableObject_type
        supertype(iTableObject_type) =
                                iSimpleWholeObject_type
        ownpropdescriptors(iTableObject_type) =
                                { iRecords }
        ownoperations(iTableObject_type) =
                                { iQuery, iMkPrsnlCpy,
                                iStandardStructChange }
```

```
                iRecords inst_of iComponentDescr_type
                        name(iRecords) =
                                                Records
                        supertype(iRecords) =
                                                iPrimalComp_descr
                        elttype(iRecords) =
                                                iRecordObject_type
                        minelt(iRecords) =
                                                0
                        maxelt(iRecords) =
                                                ∞
                        changeable(iRecords) =
                                                true
                        unique(iRecords) =
                                                true

iTableObject_type ∈ conttypes(iTable_module)

iOriginalTableObject_type inst_of iOriginalTableObjectType_type
        name(iOriginalTableObject_type) =
                                        OriginalTableObject_type
        supertype(iOriginalTableObject_type) =
                                        iTableObject_type
        ownpropdescriptors(iOriginalTableObject_type) =
                                        { iRecords' }
        ownoperations(iOriginalTableObject_type) =
                                        { iQuery', iMkPrsnlCpy' }

                iRecords' inst_of iComponentDescr_type
                        name(iRecords') =
                                                Records
                        supertype(iRecords') =
                                                iRecords
                        elttype(iRecords') =
                                                iOriginalRecordObject_type
                        minelt(iRecords') =
                                                0
                        maxelt(iRecords') =
                                                ∞
                        changeable(iRecords') =
                                                false
                        unique(iRecords') =
                                                true

iOriginalTableObject_type ∈ conttypes(iTable_module)
```

```
iDerivedTableObject_type inst_of iDerivedTableObjectType_type
        name(iDerivedTableObject_type) =
                                        DerivedTableObject_type
        supertype(iDerivedTableObject_type) =
                                        iTableObject_type
        ownpropdescriptors(iDerivedTableObject_type) =
                                        { iRecords'' }
        ownoperations(iDerivedTableObject_type) =
                                        { iQuery'', iMkPrsnlCpy'' }

                iRecords'' inst_of iComponentDescr_type
                        name(iRecords'') =
                                                Records
                        supertype(iRecords'') =
                                                iRecords
                        elttype(iRecords'') =
                                        iDerivedRecordObject_type
                        minelt(iRecords'') =
                                                0
                        maxelt(iRecords'') =
                                                ∞
                        changeable(iRecords'') =
                                                false
                        unique(iRecords'') =
                                                true

iDerivedTableObject_type ∈ conttypes(iTable_module)

iRecordObject_type inst_of iRecordType_type
        name(iRecordObject_type) =
                                        RecordObject_type
        supertype(iRecordObject_type) =
                                        iDataObject_type
        ownpropdescriptors(iRecordObject_type) =
                                        { }
        ownoperations(iRecordObject_type) =
                                        { iStandardStructChange }

iRecordObject_type ∈ conttypes(iTable_module)

iOriginalRecordObject_type inst_of iOriginalRecordType_type
        name(iOriginalRecordObject_type) =
                                        OriginalRecordObject_type
        supertype(iOriginalRecordObject_type) =
                                        iRecordObject_type
```

```
        ownpropdescriptors(iOriginalRecordObject_type) =
                                        { }
        ownoperations(iOriginalRecordObject_type) =
                                        { }

iOriginalRecordObject_type ∈ conttypes(iTable_module)

iDerivedRecordObject_type inst_of iDerivedRecordType_type
        name(iDerivedRecordObject_type) =
                                        DerivedRecordObject_type
        supertype(iDerivedRecordObject_type) =
                                        iRecordObject_type
        ownpropdescriptors(iDerivedRecordObject_type) =
                                        { }
        ownoperations(iDerivedRecordObject_type) =
                                        { }

iDerivedRecordObject_type ∈ conttypes(iTable_module)

PREDICATE = ISPD-set → B
PREDICATE ∈ Domains

iPredicate_type inst_of iBasicValueType_type
        name(iPredicate_type) =         Predicate
        supertype(iPredicate_type) =    iBasicValue_type

iPredicate_type ∈ conttypes(iTable_module)

Predicate ∈ BVTNAMES
domain Predicate = PREDICATE

iQuery inst_of iOperationDescr_type
        name(iQuery) =
                                        Query
        iQuery subtype_of iRequest_type
        ownpropdescriptors(iQuery) =
                                        { }
        otherpropdescriptors(iQuery) =
                                        { }
        oprnddescriptors(iQuery) =
                                        { iInTable, iOutTable }
        sttngdescriptors(iQuery) =
                                        { iFields, iPredicate }
```

```
iInTable inst_of iOperandDescr_type
        name(iInTable) =
                InTable
        supertype(iInTable) =
                iPrimalOprnd_descr
        elttype(iInTable) =
                iTableObject_type
        minelt(iInTable) =
                1
        maxelt(iInTable) =
                1
        changeable(iInTable) =
                true
        unique(iInTable) =
                false
        sort(iInTable) =
                in
        oprvlidty(iInTable) =
                syntvalid
iOutTable inst_of iOperandDescr_type
        name(iOutTable) =
                OutTable
        supertype(iOutTable) =
                iPrimalOprnd_descr
        elttype(iOutTable) =
                iDerivedTableObject_type
        minelt(iOutTable) =
                0
        maxelt(iOutTable) =
                1
        changeable(iOutTable) =
                true
        unique(iOutTable) =
                false
        sort(iOutTable) =
                out
        oprvlidty(iOutTable) =
                syntvalid
iFields inst_of iCrossRefDescr_type
        name(iFields) =
                Fields
        supertype(iFields) =
                iPrimalCrRef_descr
        elttype(iFields) =
                iStructPropDescr_type
```

minelt(iFields) =
0
maxelt(iFields) =
∞
changeable(iFields) =
true
unique(iFields) =
true
iPredicate inst_of iAttributeDescr_type
name(iPredicate) =
Predicate
supertype(iPredicate) =
iPrimalAttr_descr
elttype(iPredicate) =
iPredicate_type
minelt(iPredicate) =
1
maxelt(iPredicate) =
1
changeable(iPredicate) =
true
unique(iPredicate) =
true

iQuery ∈ contimpops(iTable_module)

IODQ' = M'(iQuery)
req_{iodq1} : IODQ' → B
req_{iodq1}(iodq) ≙
get(iodq, iFields) ⊆
get(elttype(get(type(get(iodq, iInTable)), iRecords)), iFieldDescriptors)

req_{iodq1} ∈ erequirements(iQuery)

iQuery' inst_of iOperationDescr_type
name(iQuery') =
Query
iQuery' subtype_of iQuery
ownpropdescriptors(iQuery') =
{ }
otherpropdescriptors(iQuery') =
{ }
oprnddescriptors(iQuery') =
{ iInTable' }
sttngdescriptors(iQuery') =
{ }

```
                    iInTable' inst_of iOperandDescr_type
                                name(iInTable') =
                                                    InTable
                                supertype(iInTable') =
                                                    iPrimalOprnd_descr
                                elttype(iInTable') =
                                                    iOriginalTableObject_type
                                minelt(iInTable') =
                                                    1
                                maxelt(iInTable') =
                                                    1
                                changeable(iInTable') =
                                                    true
                                unique(iInTable') =
                                                    false
                                sort(iInTable') =
                                                    in
                                oprvlidty(iInTable') =
                                                    syntvalid

iQuery' ∈ contops(iTable_module)

iQuery'' inst_of iOperationDescr_type
        name(iQuery'') =
                                        Query
        iQuery'' subtype_of iQuery
        ownpropdescriptors(iQuery'') =
                                        { }
        otherpropdescriptors(iQuery'') =
                                        { }
        oprnddescriptors(iQuery'') =
                                        { iInTable'' }
        sttngdescriptors(iQuery'') =
                                        { }

                    iInTable'' inst_of iOperandDescr_type
                                name(iInTable'') =
                                                    InTable
                                supertype(iInTable'') =
                                                    iPrimalOprnd_descr
                                elttype(iInTable'') =
                                                    iDerivedTableObject_type
                                minelt(iInTable'') =
                                                    1
                                maxelt(iInTable'') =
                                                    1
```

```
                                changeable(iInTable'') =
                                                true
                                unique(iInTable'') =
                                                false
                                sort(iInTable'') =
                                                in
                                oprvlidty(iInTable'') =
                                                syntvalid

iQuery'' ∈ contops(iTable_module)

iMkPrsnlCpy inst_of iOperationDescr_type
        name(iMkPrsnlCpy) =
                                        MkPrsnlCpy
        iMkPrsnlCpy subtype_of iRequest_type
        ownpropdescriptors(iMkPrsnlCpy) =
                                        { }
        otherpropdescriptors(iMkPrsnlCpy) =
                                        { }
        oprnddescriptors(iMkPrsnlCpy) =
                                        { iInTable, iOutCTable }
        sttngdescriptors(iMkPrsnlCpy) =
                                        { iDBname, iTablename }

                --iInTable as defined earlier

                iOutCTable inst_of iOperandDescr_type
                        name(iOutCTable) =
                                                OutCTable
                        supertype(iOutCTable) =
                                                iPrimalOprnd_descr
                        elttype(iOutCTable) =
                                                iOriginalTableObject_type
                        minelt(iOutCTable) =
                                                0
                        maxelt(iOutCTable) =
                                                1
                        changeable(iOutCTable) =
                                                true
                        unique(iOutCTable) =
                                                false
                        sort(iOutCTable) =
                                                out
                        oprvlidty(iOutCTable) =
                                                syntvalid
```

```
                    iDBname inst_of iAttributeDescr_type
                            name(iDBname) =
                                                    DBname
                            supertype(iDBname) =
                                                    iPrimalAttr_descr
                            elttype(iDBname) =
                                            iString_type
                            minelt(iDBname) =
                                                    1
                            maxelt(iDBname) =
                                                    1
                            changeable(iDBname) =
                                                    true
                            unique(iDBname) =
                                                    true
                    iTablename inst_of iAttributeDescr_type
                            name(iTablename) =
                                                    Tablename
                            supertype(iTablename) =
                                                    iPrimalAttr_descr
                            elttype(iTablename) =
                                                    iString_type
                            minelt(iTablename) =
                                                    1
                            maxelt(iTablename) =
                                                    1
                            changeable(iTablename) =
                                                    true
                            unique(iTablename) =
                                                    true

iMkPrsnlCpy ∈ contimpops(iTable_module)

iMkPrsnlCpy' inst_of iOperationDescr_type
        name(iMkPrsnlCpy') =
                                            MkPrsnlCpy
        iMkPrsnlCpy' subtype_of iMkPrsnlCpy
        ownpropdescriptors(iMkPrsnlCpy') =
                                            { }
        otherpropdescriptors(iMkPrsnlCpy') =
                                            { }
        oprnddescriptors(iMkPrsnlCpy') =
                                            { iInTable' }
        sttngdescriptors(iMkPrsnlCpy') =
                                            { }
```

```
                    iInTable' inst_of iOperandDescr_type
                                name(iInTable') =
                                                    InTable
                                supertype(iInTable') =
                                                    iPrimalOprnd_descr
                                elttype(iInTable') =
                                                    iOriginalTableObject_type
                                minelt(iInTable') =
                                                    1
                                maxelt(iInTable') =
                                                    1
                                changeable(iInTable') =
                                                    true
                                unique(iInTable') =
                                                    false
                                sort(iInTable') =
                                                    in
                                oprvlidty(iInTable') =
                                                    syntvalid

iMkPrsnlCpy' ∈ contops(iTable_module)

iMkPrsnlCpy'' inst_of iOperationDescr_type
            name(iMkPrsnlCpy'') =
                                                MkPrsnlCpy
            iMkPrsnlCpy'' subtype_of iMkPrsnlCpy
            ownpropdescriptors(iMkPrsnlCpy'') =
                                                { }
            otherpropdescriptors(iMkPrsnlCpy'') =
                                                { }
            oprnddescriptors(iMkPrsnlCpy'') =
                                                { iInTable'' }
            sttngdescriptors(iMkPrsnlCpy'') =
                                                { }

                    iInTable'' inst_of iOperandDescr_type
                                name(iInTable'') =
                                                    InTable
                                supertype(iInTable'') =
                                                    iPrimalOprnd_descr
                                elttype(iInTable'') =
                                                    iDerivedTableObject_type
                                minelt(iInTable'') =
                                                    1
                                maxelt(iInTable'') =
                                                    1
```

changeable(iInTable'') =
true
unique(iInTable'') =
false
sort(iInTable'') =
in
oprvlidty(iInTable'') =
syntvalid

iMkPrsnlCpy'' ∈ contops(iTable_module)
■

Definition 4: (Implementation types)
GRAMMAR ∈ Domains

iGrammar_type inst_of iBasicValueType_type
name(iGrammar_type) = Grammar
supertype(iGrammar_type) = iBasicValue_type

Grammar ∈ BVTNAMES
domain Grammar = GRAMMAR

iImplOriginalTable_type inst_of iImplObjectType_type
name(iImplOriginalTable_type) =
ImplOriginalTable_type
supertype(iImplOriginalTable_type) =
iImplObject_type
ownpropdescriptors(iImplOriginalTable_type) =
{ iAddress', iGrammar }
ownoperations(iImplOriginalTable_type) =
{ }

iAddress' inst_of iComponentDescr_type
name(iAddress') =
Address
supertype(iAddress') =
iPrimalComp_descr
elttype(iAddress') =
iDatabaseAddress_type
minelt(iAddress') =
1
maxelt(iAddress') =
1
changeable(iAddress') =
false

```
                    iGrammar inst_of iAttributeDescr_type
                                name(iGrammar) =
                                                Grammar
                                supertype(iGrammar) =
                                                iPrimalAttr_descr
                                elttype(iGrammar) =
                                                iGrammar_type
                                minelt(iGrammar) =
                                                1
                                maxelt(iGrammar) =
                                                1
                                changeable(iGrammar) =
                                                true
                                unique(iGrammar) =
                                                true

iImplOriginalTable_type ∈ conttypes(iTable_module)
iImplOriginalTable_type ∈ impltypes(iOriginalTableObject_type)

iDatabaseAddress_type inst_of iAtomicObjectType_type
        name(iDatabaseAddress_type) =
                                DatabaseAddress_type
        supertype(iDatabaseAddress_type) =
                                iObject_type
        ownpropdescriptors(iDatabaseAddress_type) =
                                { iDBname, iTablename, iQueryText }
        ownoperations(iDatabaseAddress_type) =
                                { }
                --Definition of iDBname and iTablename: see definition 3

                iQueryText inst_of iAttributeDescr_type
                                name(iQueryText) =
                                                QueryText
                                supertype(iQueryText) =
                                                iPrimalAttr_descr
                                elttype(iQueryText) =
                                                iString_type
                                minelt(iQueryText) =
                                                1
                                maxelt(iQueryText) =
                                                1
                                changeable(iQueryText) =
                                                true
                                unique(iQueryText) =
                                                true
```

```
iDatabaseAddress_type ∈ conttypes(iTable_module)

iImplDerivedTable_type inst_of iImplObjectType_type
        name(iImplDerivedTable_type) =
                                        ImplDerivedTable_type
        supertype(iImplDerivedTable_type) =
                                        iImplObject_type
        ownpropdescriptors(iImplDerivedTable_type) =
                                        { iAddress'', iGrammar }
        ownoperations(iImplDerivedTable_type) =
                                        { }

                iAddress'' inst_of iComponentDescr_type
                        name(iAddress'') =
                                                Address
                        supertype(iAddress'') =
                                                iPrimalComp_descr
                        elttype(iAddress'') =
                                                iViewDescr_type
                        minelt(iAddress'') =
                                                1
                        maxelt(iAddress'') =
                                                1
                        changeable(iAddress'') =
                                                false

iImplDerivedTable_type ∈ impltypes(iDerivedTableObject_type)
iImplDerivedTable_type ∈ conttypes(iTable_module)

iViewDescr_type inst_of iObjectType_type
        name(iViewDescr_type) =
                                        ViewDescr_type
        supertype(iViewDescr_type) =
                                        iDatabaseAddress_type
        ownpropdescriptors(iViewDescr_type) =
                                        { iQueryprop, iRemovObjs }
        ownoperations(iViewDescr_type) =
                                        { }

                iQueryprop inst_of iCrossRefDescr_type
                        name(iQueryprop) =
                                                Queryprop
                        supertype(iQueryprop) =
                                                iPrimalCrRef_descr
                        elttype(iQueryprop) =
                                                iQuery
```

```
                    minelt(iQueryprop) =
                                        1
                    maxelt(iQueryprop) =
                                        1
                    changeable(iQueryprop) =
                                        true
                    unique(iQueryprop) =
                                        true
            iRemovObjs inst_of iCrossRefDescr_type
                    name(iRemovObjs) =
                                        RemovObjs
                    supertype(iRemovObjs) =
                                        iPrimalCrRef_descr
                    elttype(iRemovObjs) =
                                iRecordKeys_type
                    minelt(iRemovObjs) =
                                        0
                    maxelt(iRemovObjs) =
                                        ∞
                    changeable(iRemovObjs) =
                                        true
                    unique(iRemovObjs) =
                                        true

iViewDescr_type ∈ conttypes(iTable_module)

iImplIspahanPopulation_type inst_of iImplObjectType_type
        name(iImplIspahanPopulation_type) =
                                ImplIspahanPopulation_type
        supertype(iImplIspahanPopulation_type) =
                                iImplObject_type
        ownpropdescriptors(iImplIspahanPopulation_type) =
                                { iAddress''' }
        ownoperations(iImplIspahanPopulation_type) =
                                { }

            iAddress''' inst_of iComponentDescr_type
                    name(iAddress''') =
                                        Address
                    supertype(iAddress''') =
                                        iPrimalComp_descr
                    elttype(iAddress''') =
                                        iFileAddress_type
                    minelt(iAddress''') =
                                        1
```

```
                                    maxelt(iAddress''') =
                                                        1
                                    changeable(iAddress''') =
                                                        false

iImplIspahanPopulation_type ∈ conttypes(iPopulationBrowse_module)
--See appendix A5.2
iImplIspahanPopulation_type ∈ impltypes(iPersonalCopyPopulation_type)

iFileAddress_type inst_of iAtomicObjectType_type
        name(iFileAddress_type) =
                                            FileAddress_type
        supertype(iFileAddress_type) =
                                            iObject_type
        ownpropdescriptors(iFileAddress_type) =
                                            { iFname }
        ownoperations(iFileAddress_type) =
                                            { }

                iFname inst_of iAttributeDescr_type
                        name(iFname) =
                                                    Fname
                        supertype(iFname) =
                                                    iPrimalAttr_descr
                        elttype(iFname) =
                                                    iString_type
                        minelt(iFname) =
                                                    1
                        maxelt(iFname) =
                                                    1
                        changeable(iFname) =
                                                    true
                        unique(iFname) =
                                                    true

iFileAddress_type ∈ conttypes(iPopulationBrowse_module)
--See appendix A5.2

iImplRecord_type inst_of iImplObjectType_type
        name(iImplRecord_type) =
                                            ImplRecord_type
        supertype(iImplRecord_type) =
                                            iImplObject_type
        ownpropdescriptors(iImplRecord_type) =
                                            { iAddress'''' }
```

```
        ownoperations(iImplRecord_type) =
                                        { }

                iAddress'''' inst_of iComponentDescr_type
                        name(iAddress'''') =
                                        Address
                        supertype(iAddress'''') =
                                        iPrimalComp_descr
                        elttype(iAddress'''') =
                                        iRecordKeys_type
                        minelt(iAddress'''') =
                                        1
                        maxelt(iAddress'''') =
                                        1
                        changeable(iAddress'''') =
                                        false

iImplRecord_type ∈ impltypes(iRecordObject_type)
iImplRecord_type ∈ conttypes(iTable_module)

iRecordKeys_type inst_of iAtomicObjectType_type
        name(iRecordKeys_type) =
                                RecordKeys_type
        supertype(iRecordKeys_type) =
                                iObject_type
        ownpropdescriptors(iRecordKeys_type) =
                                { iKeys }
        ownoperations(iRecordKeys_type) =
                                { }

                iKeys inst_of iCrossRefDescr_type
                        name(iKeys) =
                                        Keys
                        supertype(iKeys) =
                                        iPrimalCrRef_descr
                        elttype(iKeys) =
                                        iSimpleAtomicObject_type
                        minelt(iKeys) =
                                        0
                        maxelt(iKeys) =
                                        ∞
                        changeable(iKeys) =
                                        true
                        unique(iKeys) =
                                        true
```

```
iRecordKeys_type ∈ conttypes(iTable_module)

iIngresReal4_type inst_of iImplObjectType_type
        name(iIngresReal4_type) =
                                        IngresReal4_type
        supertype(iIngresReal4_type) =
                                        iImplObject_type
        ownpropdescriptors(iIngresReal4_type) =
                                        { }
        ownoperations(iIngresReal4_type) =
                                        { }

iIngresReal4_type ∈ impltypes(iRealObject_type)
iIngresReal4_type ∈ conttypes(iNumber_module)

iIngresReal8_type inst_of iImplObjectType_type
        name(iIngresReal8_type) =
                                        IngresReal8_type
        supertype(iIngresReal8_type) =
                                        iImplObject_type
        ownpropdescriptors(iIngresReal8_type) =
                                        { }
        ownoperations(iIngresReal8_type) =
                                        { }

iIngresReal8_type ∈ impltypes(iRealObject_type)
iIngresReal8_type ∈ conttypes(iNumber_module)
■
```

Definition 5: (Deriving new type information for a newly derived table)

```
imploperations(iQuery') =
        [ iDeriveType, iSetOutputImplObject, iDeriveQueryText, iDeriveGrammar,
        iDatabaseQuery ]

iDeriveType inst_of iOperationDescr_type
        name(iDeriveType) =
                                        DeriveType
        iDeriveType subtype_of iRequest_type
        ownpropdescriptors(iDeriveType) =
                                        { }
        otherpropdescriptors(iDeriveType) =
                                        { }
        oprnddescriptors(iDeriveType) =
                                        { iInTable, iOutTable }
        sttngdescriptors(iDeriveType) =
                                        { iFields }
```

```
iInTable inst_of iOperandDescr_type
        name(iInTable) =
                InTable
        supertype(iInTable) =
                iPrimalOprnd_descr
        elttype(iInTable) =
                iOriginalTableObject_type
        minelt(iInTable) =
                1
        maxelt(iInTable) =
                1
        changeable(iInTable) =
                true
        unique(iInTable) =
                false
        sort(iInTable) =
                in
        oprvlidty(iInTable) =
                syntvalid
iOutTable inst_of iOperandDescr_type
        name(iOutTable) =
                OutTable
        supertype(iOutTable) =
                iPrimalOprnd_descr
        elttype(iOutTable) =
                iDerivedTableObject_type
        minelt(iOutTable) =
                0
        maxelt(iOutTable) =
                1
        changeable(iOutTable) =
                true
        unique(iOutTable) =
                false
        sort(iOutTable) =
                out
        oprvlidty(iOutTable) =
                syntvalid
iFields inst_of iCrossRefDescr_type
        name(iFields) =
                Fields
        supertype(iFields) =
                iPrimalCrRef_descr
        elttype(iFields) =
                iStructPropDescr_type
```

minelt(iFields) =
0
maxelt(iFields) =
∞
changeable(iFields) =
true
unique(iFields) =
true

iDeriveType ∈ contypes(iTable_module)

ITP = M(iDeriveType)
ITO = M(iTableObject_type)
ITOT = M(iTableObjectType_type)

post_DeriveType : ITP × ITP × IDOBJ × IDOBJ → B
post_DeriveType(impl_req, impl_req', γ, γ') ≙
let intable = singet(impl_req, iInTable), outtable = singet(impl_req, iOutTable),
fields = get(impl_req, iFields) in
let intype = type(intable), outtype = type(outtable) in
∃ outtype' ∈ ITOT • ∃ outtable' ∈ ITO •
outtable' = outtable ∧ post_New(iDerivedTableObjectType_type, γ, γ',
outtype') ∧ ObjCopy(outtype, outtype', { PropDescriptors,
OwnPropDescriptors, KeyDescriptors, FieldDescriptors }) ∧
post_TypeProject(intype, outtype', fields) ∧ outtype' = type(outtable')

post_TypeProject : ITOT × ITOT × ISPD-set → B
post_TypeProject(intype, outtype, fields) ≙
keydescriptors(outtype) = keydescriptors(intype) ∧
fielddescriptors(outtype) = { fielddescriptor ∈ fields | fielddescriptor ∈
fielddescriptors(intype) }

iSetOutputImplObject inst_of iOperationDescr_type
name(iSetOutputImplObject) =
SetOutputImplObject
iSetOutputImplObject subtype_of iRequest_type
ownpropdescriptors(iSetOutputImplObject) =
{ }
otherpropdescriptors(iSetOutputImplObject) =
{ }
oprnddescriptors(iSetOutputImplObject) =
{ iInImplTable, iOutImplTable }
sttngdescriptors(iSetOutputImplObject) =
{ iFields }

```
iInImplTable inst_of iOperandDescr_type
        name(iInImplTable) =
                InTable
        supertype(iInImplTable) =
                iPrimalOprnd_descr
        elttype(iInImplTable) =
                iImplOriginalTableObject_type
        minelt(iInImplTable) =
                1
        maxelt(iInImplTable) =
                1
        changeable(iInImplTable) =
                true
        unique(iInImplTable) =
                false
        sort(iInImplTable) =
                in
        oprvlidty(iInImplTable) =
                syntvalid
iOutImplTable inst_of iOperandDescr_type
        name(iOutImplTable) =
                OutTable
        supertype(iOutImplTable) =
                iPrimalOprnd_descr
        elttype(iOutImplTable) =
                iImplDerivedTableObject_type
        minelt(iOutImplTable) =
                0
        maxelt(iOutImplTable) =
                1
        changeable(iOutImplTable) =
                true
        unique(iOutImplTable) =
                false
        sort(iOutImplTable) =
                out
        oprvlidty(iOutImplTable) =
                syntvalid
```

iSetOutputImplObject ∈ contops(iTable_module)

ISOIO : M(iSetOutputImplObject)

post_SetOutputImplObject : ISOIO × ISOIO × IDOBJ × IDOBJ → B
post_SetOutputImplObject(impl_req, impl_req', γ, γ') ≜
let inimpltable = singet(impl_req, iInImplTable), outimpltable' =
singet(impl_req', iOutImplTable) in
let iaddressin = address(inimpltable), iaddressout = address(outimpltable') in
get(iaddressin, DBname) = get(iaddressout, DBname) ∧
get(iaddressin, Tablename) = get(iaddressout, Tablename) ∧
get(iaddressout, Queryprop) = superobj(impl_req) ∧
get(iaddressout, RemovObjs) = { } ∧ γ' = γ

--other implementation operations of iQuery' are not further specified

{ iDeriveQueryText, iDeriveGrammar, iDatabaseQuery } ⊂ contops(iTable_module)

--imploperations(iQuery'') and imploperations(iMkPrsnlCpy) are not further specified
■

Definition 6: (Transformations for tables and populations)
(iImplOriginalTable_type, iOriginalTable_type) ∈
dom transformtable(iOriginalTable_type)
transformtable(iOriginalTable_type) (iImplOriginalTable_type, iOriginalTable_type) =
[iCreateOriginalInternTablefromDB]

iCreateOriginalInternTablefromDB inst_of iTransformOperationDescr_type
name(iCreateOriginalInternTablefromDB) =
CreateOriginalInternTablefromDB
supertype(iCreateOriginalInternTablefromDB) =
iPrimalTransform_descr
oprnddescriptors(iCreateOriginalInternTablefromDB) =
{ iInput, iOutput }

iInput inst_of iOperandDescr_type
name(iInput) =
Input
supertype(iInput) =
iPrimalOprnd_descr
elttype(iInput) =
iImplOriginalTable_type
minelt(iInput) =
1
maxelt(iInput) =
1
sort(iInput) =
in
oprvlidty(iInput) =
synt_valid

```
                    iOutput inst_of iOperandDescr_type
                            name(iOutput) =
                                    Output
                            supertype(iOutput) =
                                    iPrimalOprnd_descr
                            elttype(iOutput) =
                                    iOriginalTable_type
                            minelt(iOutput) =
                                    0
                            maxelt(iOutput) =
                                    1
                            sort(iOutput) =
                                    out
                            oprvlidty(iOutput) =
                                    synt_valid

iCreateOriginalInternTablefromDB ∈ contops(iTable_module)

(iOriginalTable_type, iImplIspahanPopulation_type) ∈ dom
transformtable(iOriginalTable_type)
transformtable(iOriginalTable_type) (iOriginalTable_type, iImplIspahanPopulation_type) =
        [ iCreateFilefromOriginalInternTable ]

iCreateFilefromOriginalInternTable inst_of iTransformOperationDescr_type
        name(iCreateFilefromOriginalInternTable) =
                CreateFilefromOriginalInternTable
        supertype(iCreateFilefromOriginalInternTable) =
                iPrimalTransform_descr
        oprnddescriptors(iCreateFilefromOriginalInternTable) =
                { iInput, iOutput }

                    iInput inst_of iOperandDescr_type
                            name(iInput) =
                                    Input
                            supertype(iInput) =
                                    iPrimalOprnd_descr
                            elttype(iInput) =
                                    iOriginalTable_type
                            minelt(iInput) =
                                    1
                            maxelt(iInput) =
                                    1
                            sort(iInput) =
                                    in
                            oprvlidty(iInput) =
                                    synt_valid
```

iOutput inst_of iOperandDescr_type
name(iOutput) =
Output
supertype(iOutput) =
iPrimalOprnd_descr
elttype(iOutput) =
iImplIspahanPopulation_type
minelt(iOutput) =
0
maxelt(iOutput) =
1
sort(iOutput) =
out
oprvlidty(iOutput) =
synt_valid

iCreateFilefromOriginalInternTable ∈ contops(iTable_module)

(iImplOriginalTable_type, iImplIspahanPopulation_type) ∈
dom transformtable(iOriginalTable_type)
transformtable(iOriginalTable_type) (iImplOriginalTable_type,
iImplIspahanPopulation_type) =
[iCreateOriginalInternTablefromDB, iCreateFilefromOriginalInternTable]
■

Definition 7: (Implementation type for Ispahan trees)
iImplIsphanTree_type inst_of iImplObjectType_type
name(iImplIspahanTree_type) =
ImplIspahanTree_type
supertype(iImplIspahanTree_type) =
iImplObject_type
ownpropdescriptors(iImplIspahanTree_type) =
{ iAddress````` }
ownoperations(iImplIspahanTree_type) =
{ }

iAddress````` inst_of iComponentDescr_type
name(iAddress`````) =
Address
supertype(iAddress`````) =
iPrimalComp_descr
elttype(iAddress`````) =
iNamedObjAddress_type
minelt(iAddress`````) =
1

```
                                maxelt(iAddress'''''') =
                                                1
                                changeable(iAddress'''''') =
                                                false

iImplIspahanTree_type ∈ impltypes(iTree_type)
iImplIspahanTree_type ∈ conttypes(iPatternRec_module)

iNamedObjAddress_type inst_of iAtomicObjectType_type
        name(iNamedObjAddress_type) =
                                        NamedObjAddress_type
        supertype(iNamedObjAddress_type) =
                                        iObject_type
        ownpropdescriptors(iNamedObjAddress_type) =
                                        { iName }
        ownoperations(iNamedObjAddress_type) =
                                        { }

                iName inst_of iAttributeDescr_type
                        name(iName) =
                                                Name
                        supertype(iName) =
                                                iPrimalAttr_descr
                        elttype(iName) =
                                        iString_type
                        minelt(iName) =
                                                1
                        maxelt(iName) =
                                                1
                        changeable(iName) =
                                                true
                        unique(iName) =
                                                true

iNamedObjAddress_type ∈ conttypes(iPatternRec_module)

iImplIspahanNode_type inst_of iImplObjectType_type
        name(iImplIspahanNode_type) =
                                        ImplIspahanNode_type
        supertype(iImplIspahanNode_type) =
                                        iImplObject_type
        ownpropdescriptors(iImplIspahanNode_type) =
                                        { iAddress'''''' }
        ownoperations(iImplIspahanNode_type) =
                                        { }
```

```
                    iAddress`````` inst_of iComponentDescr_type
                            name(iAddress``````) =
                                                    Address
                            supertype(iAddress``````) =
                                                    iPrimalComp_descr
                            elttype(iAddress``````) =
                                                    iNodeAddress_type
                            minelt(iAddress``````) =
                                                    1
                            maxelt(iAddress``````) =
                                                    1
                            changeable(iAddress``````) =
                                                    false

iIspahanImpl_type ∈ impltypes(iTree_type)
iIspahanImpl_type ∈ conttypes(iPatternRec_module)

iNodeAddress_type inst_of iAtomicObjectType_type
        name(iNodeAddress_type) =
                                            NodeAddress_type
        supertype(iNodeAddress_type) =
                                            iObject_type
        ownpropdescriptors(iNodeAddress_type) =
                                            { iNodeName, iTreeName }
        ownoperations(iNodeAddress_type) =
                                            { }

                    iNodeName inst_of iAttributeDescr_type
                            name(iNodeName) =
                                                    NodeName
                            supertype(iNodeName) =
                                                    iPrimalAttr_descr
                            elttype(iNodeName) =
                                                    iString_type
                            minelt(iNodeName) =
                                                    1
                            maxelt(iNodeName) =
                                                    1
                            changeable(iNodeName) =
                                                    true
                            unique(iNodeName) =
                                                    true
                    iTreeName inst_of iAttributeDescr_type
                            name(iTreeName) =
                                                    TreeName
```

supertype(iTreeName) =
iPrimalAttr_descr
elttype(iTreeName) =
iString_type
minelt(iTreeName) =
1
maxelt(iTreeName) =
1
changeable(iTreeName) =
true
unique(iTreeName) =
true

iNodeAddress_type ∈ conttypes(iPatternRec_module)
■

Definition 8: (Implementation operations for a specific Ispahan operation)
imploperations(iTest) =
[iSetOutputObject, iSetOutputImplObject, iTestInvoke]

iSetOutputObject inst_of iOperationDescr_type
name(iSetOutputObject) =
SetOutputObject
iSetOutputObject subtype_of iRequest_type
ownpropdescriptors(iSetOutputObject) =
{ }
otherpropdescriptors(iSetOutputObject) =
{ }
oprnddescriptors(iSetOutputObject) =
{ iNewtre }
sttngdescriptors(iSetOutputObject) =
{ }

iNewtre inst_of iOperandDescr_type
name(iNewtre) =
Newtre
supertype(iNewtre) =
iPrimalOprnd_descr
elttype(iNewtre) =
iTree_type
minelt(iNewtre) =
1
maxelt(iNewtre) =
1
changeable(iNewtre) =
true

```
                                    unique(iNewtre) =
                                                        false
                                    sort(iNewtre) =
                                                        in_out
                                    oprvlidty(iNewtre) =
                                                        syntvalid

iSetOutputImplObject inst_of iOperationDescr_type
        name(iSetOutputImplObject) =
                                                SetOutputImplObject
        iSetOutputImplObject subtype_of iRequest_type
        ownpropdescriptors(iSetOutputImplObject) =
                                                { }
        otherpropdescriptors(iSetOutputImplObject) =
                                                { }
        oprnddescriptors(iSetOutputImplObject) =
                                                { iImplNewtre }
        sttngdescriptors(iSetOutputImplObject) =
                                                { }

                iImplNewtre inst_of iOperandDescr_type
                        name(iImplNewtre) =
                                                        Newtre
                        supertype(iImplNewtre) =
                                                        iPrimalOprnd_descr
                        elttype(iImplNewtre) =
                                                        iImplIspahanTree_type
                        minelt(iImplNewtre) =
                                                        1
                        maxelt(iImplNewtre) =
                                                        1
                        changeable(iImplNewtre) =
                                                        true
                        unique(iImplNewtre) =
                                                        false
                        sort(iImplNewtre) =
                                                        in_out
                        oprvlidty(iImplNewtre) =
                                                        syntvalid

iTestInvoke inst_of iOperationDescr_type
        name(iTestInvoke) =
                                                TestInvoke
        iTestInvoke subtype_of iRequest_type
        ownpropdescriptors(iTestInvoke) =
                                                { }
```

```
otherpropdescriptors(iTestInvoke) =
                                { }
oprnddescriptors(iTestInvoke) =
                                { iImplPnode, iImplDnode, iImplNewtre }
sttngdescriptors(iTestInvoke) =
                                { iNewtrenam, iPrior? }

        iImplPnode inst_of iOperandDescr_type
                name(iImplPnode) =
                                Pnode
                supertype(iImplPnode) =
                                iPrimalOprnd_descr
                elttype(iImplPnode) =
                                iImplIspahanNode_type
                minelt(iImplPnode) =
                                1
                maxelt(iImplPnode) =
                                1
                changeable(iImplPnode) =
                                true
                unique(iImplPnode) =
                                false
                sort(iImplPnode) =
                                in
                oprvlidty(iImplPnode) =
                                syntvalid
        iImplDnode inst_of iOperandDescr_type
                name(iImplDnode) =
                                Dnode
                supertype(iImplDnode) =
                                iPrimalOprnd_descr
                elttype(iImplDnode) =
                                iImplIspahanNode_type
                minelt(iImplDnode) =
                                1
                maxelt(iImplDnode) =
                                1
                changeable(iImplDnode) =
                                true
                unique(iImplDnode) =
                                false
                sort(iImplDnode) =
                                in
                oprvlidty(iImplDnode) =
                                syntvalid
```

Definition 9: (Transformation for trees)
(iImplIspahanTree_type, iTree_type) ∈ dom transformtable(iImplIspahanTree_type)
transformtable(iTree_type) (iImplIspahanTree_type, Tree_type) =
[CreateInternTreefromIspTree]

iCreateInternTreefromIspTree inst_of iTransformOperationDescr_type
name(iCreateInternTreefromIspTree) =
CreateInternTreefromIspTree
supertype(iCreateInternTreefromIspTree) =
iPrimalTransform_descr
oprnddescriptors(iCreateInternTreefromIspTree) =
{ iInput, iOutput }

iInput inst_of iOperandDescr_type
name(iInput) =
Input
supertype(iInput) =
iPrimalOprnd_descr
elttype(iInput) =
iImplIspahanTree_type
minelt(iInput) =
1
maxelt(iInput) =
1
sort(iInput) =
in
oprvlidty(iInput) = synt_valid
iOutput inst_of iOperandDescr_type
name(iOutput) =
Output
supertype(iOutput) =
iPrimalOprnd_descr
elttype(iOutput) =
iTree_type
minelt(iOutput) =
1
maxelt(iOutput) =
1
sort(iOutput) =
in_out
oprvlidty(iOutput) =
synt_valid

■

Definition 10: (Example of implementation operation for Addition)

```
imploperations(iAdd) =
        [ iImplAdd ]

iImplAdd inst_of iOperationDescr_type
        name(iImplAdd) =
                                        ImplAdd
        iImplAdd subtype_of iRequest_type
        ownpropdescriptors(iImplAdd) =
                                        { }
        otherpropdescriptors(iImplAdd) =
                                        { }
        oprnddescriptors(iImplAdd) =
                                        { iOp1, iOp2, iOtp }
        sttngdescriptors(iImplAdd) =
                                        { }

                iOp1 inst_of iOperandDescr_type
                        name(iOp1) =
                                                Op1
                        supertype(iOp1) =
                                                iPrimalOprnd_descr
                        elttype(iOp1) =
                                                iRealObject_type
                        minelt(iOp1) =
                                                1
                        maxelt(iOp1) =
                                                1
                        changeable(iOp1) =
                                                true
                        unique(iOp1) =
                                                false
                        sort(iOp1) =
                                                in
                        oprvlidty(iOp1) =
                                                syntvalid
                iOp2 inst_of iOperandDescr_type
                        name(iOp2) =
                                                Op2
                        supertype(iOp2) =
                                                iPrimalOprnd_descr
                        elttype(iOp2) =
                                                iRealObject_type
                        minelt(iOp2) =
                                                1
```

maxelt(iOp2) =
1
changeable(iOp2) =
true
unique(iOp2) =
false
sort(iOp2) =
in
oprvlidty(iOp2) =
syntvalid
iOtp inst_of iOperandDescr_type
name(iOtp) =
Otp
supertype(iOtp) =
iPrimalOprnd_descr
elttype(iOtp) =
iRealObject_type
minelt(iOtp) =
0
maxelt(iOtp) =
1
changeable(iOtp) =
true
unique(iOtp) =
false
sort(iOtp) =
out
oprvlidty(iOtp) =
syntvalid

IODIA = M(iImplAdd)

post_ImplAdd : IODIA × IODIA × IDOBJ × IDOBJ → B
post_ImplAdd(impl_req, impl_req', γ, γ') ≙
let op1 = get(impl_req, iOp1), op2 = get(impl_req, iOp2), otp' = get(impl_req', iOtp) in
singet(otp', iValue) = singet(op1, iValue) + singet(op2, iValue) ∧ γ' = γ

■