SYNTHESIS: a Language for Canonical Information Modeling and Mediator Definition for Problem Solving in Heterogeneous Information Resource Environments

Moscow 2007
Kalinichenko L.A., Stupnikov S.A., Martynov D.O.

Description of the SYNTHESIS language intended for canonical information modeling and mediator definition for problem solving in application-driven distributed heterogeneous information environment is presented. The SYNTHESIS is a multipurpose language that is oriented on the following basic kinds of application:

- serving as a kernel of the canonical information model;
- providing facilities for unifying representation of heterogeneous information models of different kinds (of data, services, processes, ontologies);
- providing sufficient modeling facilities for formalized definitions of mediators for various subject domains;
- supporting mediation-based design of information systems;
- providing modeling facilities for mediator and resource specifications for the mediation architecture;
- serving for semantic reconciliation of the mediator and resource specifications to form compositions of resources refining mediator specifications;
- serving as an interface for users during problem formulation and solving in various applications.

FOR FURTHER INFORMATION PLEASE CONTACT:
Institute of Informatics Problems RAS
44-2 Vavilov str., Moscow, 119333, Russian Federation
Tel. 7 495 1292098
E-mail: leonidk@synth.ipi.ac.ru

## Contents

**Foreword** 6

**Introduction** 8

1 Motives and objectives of the language development 9
   1.1 A subject mediator concept .......................... 9
   1.2 Mediator definition and information resources registration .......................... 9
   1.3 Extensible canonical models synthesis .......................... 12
   1.4 Problem formulation applying mediator specifications .......................... 14
   1.5 SYNTHESIS language application areas .......................... 15

2 SYNTHESIS facilities for canonical representation of mediator and information resources 16
   2.1 SYNTHESIS language facilities characterization .......................... 16
   2.2 Levels and styles of the SYNTHESIS language usage .......................... 19

3 Structuring of information resource specifications in SYNTHESIS 20

4 Frame representation and manipulation facilities 22
   4.1 General characteristics of the frame language .......................... 22
   4.2 Rules for the syntactic definition of the SYNTHESIS language .......................... 24
   4.3 Basic elements of the frame language .......................... 24
      4.3.1 Frames .......................... 24
      4.3.2 Metaframes .......................... 26
      4.3.3 Binary associations .......................... 28
   4.4 Facilities supporting the temporal category of the language .......................... 29
      4.4.1 Temporal assignments of frames and of their components .......................... 29
      4.4.2 Reflection of entity evolution in the frame base .......................... 32
      4.4.3 Temporal associations .......................... 33
   4.5 Worlds and contexts .......................... 35
   4.6 Frame base manipulation facilities .......................... 36
   4.7 Formulae over the frame base .......................... 39
      4.7.1 Formulae construction rules .......................... 39
      4.7.2 Temporal assignment of formulae .......................... 39
5 Mediator and information resource specification modules

6 Type system of the SYNTHESIS language
   6.1 Principles of the type system construction
       6.1.1 A structure of the type system of the SYNTHESIS language
   6.2 Type specification
       6.2.1 Type expressions
   6.3 Function type
   6.4 Abstract data type
       6.4.1 Attribute specification in abstract data types
   6.5 Built-in types
   6.6 Built-in metatypes

7 Class representation facilities
   7.1 Collections of objects modeling
   7.2 Mediator or information resource specification section
   7.3 Metaclasses
   7.4 Association metaclasses
   7.5 Transaction classes and activities
   7.6 Script classes

8 Facilities used for expressing assertions on application domain entities
   8.1 Mediator and information resource assertions
   8.2 Built-in assertions
   8.3 Metalevel assertions
   8.4 Built-in assertional attribute categories
   8.5 Event expressions

9 Logic formulae representation
   9.1 General rules
   9.2 Formulae construction rules
   9.3 General rules of formulae interpretation
       9.3.1 Atomic formulae
       9.3.2 Compound formulae
   9.4 Interpretation of various formulae of the language
       9.4.1 Collection ordering
       9.4.2 Formulae with universal quantifiers
       9.4.3 Set functions
Foreword

This report contains description of the SYNTHESIS language reflecting its evolution comparing to the previous language descriptions known according to [10, 11]. The evolution took place due to the following significant activities that required the language application:

- development of methods and facilities for compositional information systems (IS) design;
- development of methods and tools for mediation-based IS development;
- application of extensible canonical model synthesis principles to various kinds of information models (structural, object, ontological, process-oriented);
- application of mediation-based IS development methods in e-science.

These activities provided for clarification of the language intention and helped to refine the language constructs. Brief characterization of the language objectives follows.

Continuous expansion of IT is characterized by the process of explosive growth of various information representation models. This development takes place in frame of specific distributed infrastructures (such as OMG architectures (in particular, the model driven architecture (MDA)), Semantic Web and Web service architectures, digital library architectures as collective memories of information in various subject domains, architectures of the information Grid), as well as in the standards of languages and data models (such as, for example, ODMG, SQL, UML, XML and RDF stacks of data models), process models and workflow models, semantic models (including ontological models and models of metadata), models of digital repositories of data and knowledge in particular domains. This process is accompanied by another trend – intensive development of based on such models information components and services. This growth causes the accelerating need for integration in various applications of components and services represented in heterogeneous models, as well as their reuse and composition implementing new information systems. The more variety of applied models we meet in various components and services, the more complex become problems of their integration and composition. New approaches for information systems (IS) development are needed for applications requiring problem solving over multiple pre-existing information resources (containing data, services and processes).
Two principally different approaches to the problem of integrated representation of multiple information resources for an IS are distinguished:

1. moving from resources to problems (an integrated schema of multiple resources is created independently of specific applications) and

2. moving from an application to resources (a description of an application subject domain (in terms of concepts, data structures, functions, processes) is created, in which resources relevant to the application are mapped).

The first approach driven by information resources is not scalable with respect to the number of resources, does not make semantic integration of resources in a context of specific application possible, does not lead to justifiable identification of relevant to IS resources, does not provide for enhancing of IS stability w.r.t. evolution of the relevant to IS resources. These deficiencies are inherent to the Global as View (GAV) approach studied elsewhere [8, 21]. GAV might be used as a basic technique for the information resources driven approach.

The second approach (application-driven) assumes creation of subject mediator that supports an interaction between an application and resources on the basis of the application domain definition (description of the mediator). Subject mediator approach makes possible to overcome the deficiencies of the approach driven by information resources.

The SYNTHESIS language is oriented on support of application-driven approach for mediation-based IS development. Such intention can be considered as a concretization of the language objectives formulated in previous editions of the language as modeling facilities for semantic interoperability in heterogeneous information resource environments mostly focused on compositional development of IS with re-use of pre-existing information resources.

This edition of the language description ¹ is based on the [11] report. Frame manipulation and SYNTHESIS formulae constructs are refined, subset of the formulae language is added as well as inconsistencies of the language continued to be overcome.

L.A. Kalinichenko, S.A. Stupnikov, D.O. Martynov

Moscow, September 2007

¹The work was supported by the Russian Foundation for Basic Research, grants 06-07-08072 and 06-07-89188
Introduction

This SYNTHESIS language edition is motivated by the subject mediation approach application for various subject domains. The language use in this context has been experienced due to the following activities that required the language use: development of methods and facilities for compositional IS design; development of methods and tools for mediation-based IS development; application of extensible canonical model synthesis approach to various kinds of information models (structural, object, ontological, process-oriented). Main results of these activities are summarized in the next section.

The SYNTHESIS is a multipurpose language. It is intended for the semi-formal definition of mediators for different application domains, for uniform specification of heterogeneous information resources refining mediator definitions, for mediation-based IS development, for application problem formulation in context of a mediator definition.

The model used for uniform representation of various information models in one paradigm is called the “canonical” model. The model intends to provide for uniform (canonical) representation of mediator and heterogeneous resources. The same model is intended also for description of ontological models of subject domains.

The SYNTHESIS language facilities determine the canonical information model kernel that should be sufficiently rich for refining mapping into it of various models used for information resources support. The scale of diversity of forms of information models available makes clear how hard is the problem of imposing such representation uniformity. One of the important SYNTHESIS features is that the language is hybrid, providing facilities for semi-structured as well as for typed data modeling.

The intention to apply mediation-based IS development predetermined the declarative nature of the language and its usage in different styles characterized by application of predicative specifications providing for their verifiability and for the formalization of the notion of refinement of mediator specification by a composition of resources, by application of an object calculus together with logic programming facilities and hybrid typed and frame-based modeling for mediator and resources specification as well as for the problems formulation in an application domain, by application of facilities for specification of various semantic assertions expressing constraints of an application domain and pre-existing resources, and by declarative facilities for expressing of functions, processes and queries over a mediator.

The edition of the language was prepared by means of the \LaTeX type-
setting system. In the text the syntax and the language constructs are emphasized by sans serif, examples are given in italics.

1 Motives and objectives of the language development

1.1 A subject mediator concept

Subject mediators support representation and access to various subject domains. The mediator’s layer is introduced to provide the users with the metainformation uniformly characterizing subject definitions in the canonical information model. This model is needed to express the structure and semantics of a subject domain and to map into this model heterogeneous information resources preserving information and operations.

Each mediator supports the process of systematic registration and classification of resources providing the uniform ontological knowledge and metainformation for discovery and composition of information resources relevant to mediator.

1.2 Mediator definition and information resources registration

Definition of a subject mediator and registration of information resources in mediator is based on the experience of compositional development of IS [5]. Registration of resources is a process of purposeful specification transformation including decomposition of mediator specifications into consistent fragments, search among specifications of relevant resources of data types treating as candidates for refining by them of the mediator specification types, construction of expressions defining resource classes as composition of the mediator classes. For such manipulation a specification composition calculus has been developed [9]. A principle of type specification decomposition into a set of specification reducts serving as the basic units of reuse and composition has been declared. An operation of identification of the most common reduct of resource type and mediator type specifications has been introduced. Type lattice and type algebra have also been defined. Important point in this scheme consists in implementation of the type refinement proof applying logical model of the mediator and resource type specifications in formal notation. The Abstract Machine Notation (AMN) [2] is applied for these purposes. It allows to define the model-theoretic specifications in the first order logics
and to prove the fact of specification refinement. It is said that specification \( A \) refines specification \( D \), if it is possible to use \( A \) instead of \( D \) so that the user of \( D \) does not notice this substitution. Existence of B-technology for AMN [3] provides for conducting proofs of specification refinement.

Compositional calculus emphasizes complete type specifications and expressiveness sacrificing tractability in complex cases. Such decision is motivated by orientation of the modeling facilities on type refinement and composition. The benefits obtained include rigorous way of identification of common fragments of resource and mediator type specifications leading to justifiable mapping of resource type into mediator type. Investigations oriented on simpler data types considered functions as components modeled with relations and refinement ordering of functions [18]. But this is too restrictive for the mediation infrastructure.

A process of registration of heterogeneous information resources in a subject mediator is based on GLAV that combines two approaches – Local As View (LAV) and Global As View (GAV). According to LAV the schemas of resources being registered are considered as materialized views over virtual classes of a mediator. GAV views provide for reconciliation of various conflicts between resource and mediator specifications and provide rules for transformation of a query results from resource into mediator representation. Such registration technique provides for stability of IS application specification during any modifications of specific information resources and of their actual presence (removing, addition new ones, etc.) as well as for scalability of mediators w.r.t. the number of resources registered in them.

Application specification for IS includes definition of terminology and concepts of the subject domain of the application. They are expressed by the respective dictionaries (thesauri) and ontological definitions in the canonical model. Application definitions include also specifications of object classes corresponding to the application subject domain, specification of instance types of these classes and of their methods defining behavior of the objects, specification of processes characteristic for the application. On the early stages of design the specifications mentioned can be expressed by means of various specification languages (e.g., by UML). However, it is assumed that finally such specifications (including ontological ones) are mapped into the canonical model specifications having formal semantics. It is worth of remark that application-driven specifications are formulated independently of specific pre-existing information resources. The result of this specification activity fulfilled by a community interested in the specific IS application constitutes the mediator specification created as the result of reaching a consensus in such community. The mediator specification
activity is called the mediator consolidation phase.

During the operational phase the resources relevant to a subject mediator can be registered in it. Information resources should be defined in terms of the subject mediator, the required wrappers are assumed to be developed. Registrations of various resources are independent of each other and can be done at any time.

Thus, the basic principles of application-driven IS development over multiple heterogeneous information resources can be summarized as follows:

1. independence of application (mediator) specification of the existing information resources;

2. definition of an application mediator as a result of consolidation effort of the respective community;

3. emphasizing semantic canonical definitions for the mediator specification;

4. user interface independence of the resources registered at the mediator: application users should be only conscious of definition of the application domain (definition of mediator);

5. independence of publication of the newly developed information resources of the mediators (as a result of such publication, actions can be initiated on registration of such resources in respective mediators);

6. three stage identification of information resources relevant to mediator providing
   - resource relevance to requirements expressed by metadata;
   - resource ontological relevance;
   - resource structural and behavioral relevance;

7. semantic integration of relevant heterogeneous information resources in canonical mediator specification;

8. integrated access to the information resources registered at mediator applying the canonical model and mediator program (query) rewriting system;

9. recursive structure of mediators: each mediator can be registered as a new information resource. Such mediator capability makes possible integration of different applications that is specifically important for virtual organizations development.
1.3 Extensible canonical models synthesis

Subject mediators are specified in a canonical information model. With respect to information resources the same model serves as the common language, "Esperanto", for adequate uniform expression of semantics of various information models used in resources existing in an IS environment. For such expression of semantics it is required to define mappings of resource information models into the canonical one. Research and development of adequate methods for manipulation of various information models are required.

To prove that a definition in one language can be substituted with a definition in another one, formal specification facilities and commutative model mapping methods are provided. Initially ideas of mapping structured data models and canonical model construction for them were developed. The basic definitions of equivalence of database states, database schemas and data models were introduced to preserve operations and information while constructing of mappings of various structured data models into the canonical one [9]. According to this approach, each data model was defined by syntax and semantics of two languages – data definition language (DDL) and data manipulation language (DML). The main principle of mapping of an arbitrary resource data model into the target one (the canonical model) constituted the principle of commutative data model mapping. According to it, preserving of operations and information of a resource data model while mapping it into the canonical one could be reached under the condition, that the diagram of DDL (schemas) mapping and the diagram of DML (operators) mapping are commutative. At that time in the process of data model mappings construction the denotational semantics were used as a formalism (metamodel), allowing to prove a commutativity of the diagrams mentioned [9]. Extensions of the canonical model kernel used for construction of mapping of various structured data models were incorporated into the SYNTHESIS language as its extensions expressed as built-in assertions.

Later, for the object data models, the method of data model mapping and canonical models constructions was modified as follows. As a formalism (metamodel) of the method the Abstract Machine Notation (AMN) [2] was used instead of the denotational semantics. Instead of equivalence of respective specifications, it became possible to reason on their refinement [12]. B-technology for AMN [3] provides for conducting proofs of commutativity of model mappings semi-automatically.

The main principle of canonical model synthesis is that its extensibility is required for semantic integration and information interoperability in heterogeneous environment, including various models. A kernel of the
1.3 Extensible canonical models synthesis

canonical model is fixed. For each specific information model $M$ of the environment an extension of the kernel is defined so that this extension together with the kernel is refined by $M$. Such refining transformation of models should be provably correct. The canonical model for the environment is synthesized as the union of extensions, constructed for models $M$ of the environment. The resource schema refines the canonical model schema. The refinement of the schema mapping is formally checked. Another existing methods (e.g., used in Clio project [7]) supporting generation of mappings (queries) between resource and target schemas are applicable to limited class of structured data models and do not allow to create an application specifications providing for registration of various kinds of relevant resources (including objects, services, processes). On the contrary, the canonical data model synthesis method developed for the mediation provides a seminal role for synthesis of canonical models for various kinds of resource information models including process models [16], service models [20], ontological models [14].

Canonical model synthesis method has been applied to the process models, required for describing application activities of enterprises [16]. While mapping processes for synthesis of their canonical model, it is required to preserve the semantics of concurrency. A possibility of interpretation of concurrent process events in logics, and specifically, in the AMN has been discovered recently. Algorithms of process specifications mappings into AMN were constructed [6, 19]. This approach allows to construct provable refinements of process specifications, applying the B-technology. Simultaneously classification of the diversity of workflow models by means of workflow patterns has been obtained [1]. Due to these findings, the possibility of creation of a canonical process model kernel and construction of its extensions, refined by various workflow patterns, became possible. Finally it was shown how to synthesize the canonical process model [16].

Similarly, to work with ontologies of different resources a unified representation of ontologies is required. Again, it is provided by a canonical ontological model that includes a kernel and its extensions for every specific ontological model used in the information resources. Extensions are developed so that the ontological model of a resource should serve as a refinement of an extended canonical ontological model of the mediator. After analysis of various ontological models the kernel of canonical ontological model has been identified as a subset of the object model with logical capabilities [13, 14]. This kernel has been defined as a subset of the general mediator canonical model kernel mentioned above. This subset includes:
Motives and objectives of the language development

- concepts defined as abstract data types (ADT);
- type invariants expressing concept constraints;
- generic ontological metaclasses, instances of which are ADTs defining concepts;
- verbal definitions of concepts applying metaframes annotating types expressing concepts.

One of the popular ontological models, OWL DL has been mapped into the ontological canonical model and respective definition of the canonical model kernel extension has been defined [14]. Correctness proof of such extensions is provided by AMN tools to justify refinement of the extension by the resource ontological model.

During practical synthesis of canonical information models it is advisable to produce reversible language mappings that will be suitable for the process of heterogeneous resources registration in the mediator as well as for the run time support when wrappers (adapters) should interpret operations of the canonical level in terms of the resource model language. Taking into account the labor intensive character of development of the required mappings (compilers) it is reasonable to apply specific Meta Environment facilities providing for declarative specification of compilers and generating them according to such specifications [4]. It is such meta approach that is used in frame of the SYNTHESIS project.

The SYNTHESIS language serves as a kernel for extensible canonical models development. Basic constructs of the language are provided with formal semantics and possibilities of proof of the fact of their refinement by resource models.

1.4 Problem formulation applying mediator specifications

General approach consists in problem formulation in terms of subject mediator specifications and transformation of this formulation into set of tasks (queries) to the real information resources registered in the mediator. Such transformation in the database theory is known as view based query rewriting (information resources are treated as materialized views over virtual classes of the mediator) [8]. This is complicated problem having various (usually limited) solutions applicable to specific data models. In context of SYNTHESIS the rewriting techniques applying inverse rules are used [8, 22]. Typed object facilities of SYNTHESIS required development of specific method of the mediator programs rewriting applying the
inverse rule technique [15]. The method is based on the use of refinement relationship between mediator data types and resource data types helping to get containment of the rewritten queries in the original mediator level queries expressed in SYNTHESIS formula subset. It is important to note that a combination of LAV and GAV approaches is supported during rewriting queries as GLAV. This approach for problem formulation has been applied in the Virtual Observatories area [17].

1.5 SYNTHESIS language application areas

SYNTHESIS language basic objectives are determined by its main orientation on canonical information modeling and mediator definition for problem solving in application-driven distributed heterogeneous information environment. Due to that its basic application areas can be listed as the following:

- serving as a kernel of the canonical information model;
- providing facilities for unifying representation of heterogeneous information models of different kinds (of data, services, processes, ontologies);
- providing sufficient modeling facilities for formalized definitions of mediators for various subject domains;
- supporting mediation-based design of information systems;
- providing modeling facilities for mediator and resource specifications for the mediation architecture;
- serving for semantic reconciliation of the mediator and resource specifications to form compositions of resources refining mediator specifications;
- serving as an interface for users during problem formulation and solving in various applications.
2 SYNTHESIS facilities for canonical representation of mediator and information resources

2.1 SYNTHESIS language facilities characterization

A set of the SYNTHESIS language basic facilities used for uniform representation of mediator and information resources includes the following:

- **Frame representation facilities.** Frames are treated as a special kind of abstract values introduced mostly for description of concepts, terminological and semistructured information on an application domain. Frame representation facilities provide for expressing of arbitrary semantic associations of frames. All specifications in SYNTHESIS have a form of frames that become a part of the metainformation base. Temporal information support for all levels, maintaining of the historical data and the specific set of functions for manipulation of temporal information are incorporated through the frame model.

- **Unifying type system.** A universal constructor of arbitrary abstract data types as well as a comprehensive collection of the built-in types are included into a type system. For types a multiple subtyping relationship is defined. A possibility of the definition of generic types is provided. Types are treated as a special kind of value and may be passed as parameters of polymorphic functions. Metatypes provide for classification of the type hierarchy, to specify interfaces of the polymorphic functions. Type expressions are introduced providing for type compositions. Such expressions are required to type variables and the results of formulae application. Typing continuum is supported: from typeless values (represented by frames) to strictly typed objects. Frame, object and frame-object associations are supported.

- **Functions and collections (including sets).** Functions (predicates) are used as constituents of abstract data type specifications, for object manipulation, for constraints specification, view formation. Collections are included for creation of sets of abstract data type values being immutable as well as of objects possessing unique identity and being mutable. For collections (independently on their kinds) an algebra is defined that is closed with respect to a composition of the algebraic operations. Such algebra and a first order calculus
language make possible to define logic programs, constraints and queries to the mediator and to the information resources after mediator query rewriting.

- **Class representation.** Classes provide for representing of sets of homogeneous entities of an application domain. Class is treated as a subtype of a set. Class hierarchies subtyping mechanisms make possible to define the generalization/specialization class relationships. Class instances (objects) are typed with the abstract data types. Methods are specified in the SYNTHESIS language as functions. Declarative logic assertions and functions treated as derivation rules or consistency constraints can be associated with classes, types and their attributes. Events linked to such assertions define conditions of their enforcement. Metaclasses provide for introducing of another kind of classification relationships orthogonal to the class generalization relationship. Singular objects are also supported alongside with classes.

- **Multiactivity (process) representation.** These are used for the specification of interconnected and interdependent application activities, for the specification of declarative assertions and concurrent megaprograms (workflows) over the information resources. Main facilities for activity representation include multiactivities, scripts, assertions, event expressions, action enforcement and coupling condition facilities. These facilities provide for specification of concurrent and asynchronous behavior of applications in mediators and of interoperable resource environments.

- **Definition of assertions.** Assertions can be associated with various specification elements (attribute, type, class, collection). These facilities are based on the typed first order language formulae. To the main components of assertions belong an assertion predicate; an event expression, defining event sequences leading to assertion enforcement; a function, expressing actions that should be invoked according to the condition specified.

- **Facilities for the logical formulae expressions.** A multisorted object calculus (typed first order language) is used for program definition over a mediator as well as for specification of constraints and behaviour. The usage of the formulae goes beyond query formulation and includes: specification of rules in functions and processes; specification of assertion components; predicative specification of
functions; specification of constraints expressing properties of the mediator/resource semantics.

Information on the entities and situations observed in a real world is represented in the mediator as a collection of abstract values (immutable or mutable). In this range we can differentiate between:

- collections of self-defined objects or collections of frames (worlds);
- worlds with fixed frame associations;
- classes containing partially typed objects containing their own individual attributes that were not specified in a type of the class instance;
- strictly typed classes (a set of instance attributes is strictly fixed);
- aggregates (associations of objects and frames).

All objects and frames are assumed to belong to one and the same general type of entities – to a type of abstract values. Objects are considered to be a frame specialization providing for unique immutable identification. This means that all operations for frames are applicable also to objects (but not vice versa). A frame at any moment of its life cycle can be declared to belong to a certain class. At that moment the frame becomes an object.

Capabilities of a mediator definition are ranged from purely declarative to purely procedural way of thinking. At one end of the range a definition is considered consisting solely of a collection of logic formulae, at another – consisting of abstract values and triggers simulating application domain events in an active mediator. This range is supported by the SYNTHESIS language.

The following, more technical, language features are important for application-driven mediation:

1. Modular description of an application domain with a separation of local (resource) and global (mediator) levels of description. Modules of the local level support specifications of information resources in the canonical model. Global level modules specify the mediators. Modularity combined with parameterized specifications for subsequent concretization provides for an ability of such generic contexts specialization for specific problem solving (such specialization can be dynamic). Generic specifications are applicable to various language constructs (types, classes, modules, assertions, functions, formulae).
2. Support of active mediator definitions (actions enforced by events and (or) predicates, declaratively specified actions and constraints applied to objects and values).

3. For technical interoperability support by some middleware (e.g., CORBA, Web services) the type system of the SYNTHESIS language is designed so that there exists a possibility of refining mapping of the IDL, WSDL specifications into the SYNTHESIS specification. Due to that information resources included into the middleware-compliant environment become accessible through a mediator.

2.2 Levels and styles of the SYNTHESIS language usage

The SYNTHESIS language is oriented on different levels of specification of the mediation environment: on the unifying (canonical) definition of heterogeneous resources registered in a mediator, application domain (mediator) specification, problem formulation in terms of a mediator.

Mediator specification (a result of the consolidation phase of mediator life cycle) serves for the following:

- provides a reference model defining a structure and behavior of the information system entities;
- defines rules for the admissible evolution of the application domain entities;
- provides a conceptual definition for problem formulation over the mediator;
- serves as a conceptual model for identification of resources relevant to the mediator and construction of their refining mapping into the mediator specification;
- serves as a conceptual specification for design and implementation of a mediator specification part that is not refined by any of the pre-existing resources.

Different styles of the resource specification correspond to different levels of the interoperable environment modeling. The difference in styles characterizing a range of possible decisions can be conceived from the tentative difference between facilities of the SYNTHESIS language suggested for different layers of the resource environment specification.
3 Structuring of resource specifications in SYNTHESIS

The mediator description style. The style is characterized by high declarativity and minimum of details. Specific features of the style include:

- a usage of abstractions convenient for the application domain modeling;
- generic types and metatypes are recommended to obtain generic specifications;
- representation of functions and assertions by predicative specifications.

The problem formulation style. Specific features of the style include:

- introducing of functions and process (transaction) definitions specific for a problem;
- concretization of types and functions of the mediator;
- definition of functions and process activities by predicative specifications;
- applying formula language facilities for program (query) definition.

Information resource description style. The style is characterized by a high level of concretization of specifications. Specific features of the style include:

- high level of type concretization specifications;
- types, classes, functions, assertions and processes are encapsulated, their canonical specifications are provided.

3 Structuring of information resource specifications in SYNTHESIS

Modules of mediator description are defined on the global level, modules for information resource descriptions are defined on the local level. Modules are specified in the SYNTHESIS language providing for the definition of different kinds of module sections: frame section, type section, function section, mediator/information resource specification section, mediator/information resource assertion section. Arbitrary combinations of sections are allowed. Any module can import an arbitrary collection of
other modules containing specifications of types, classes, assertions constituting a context of a module.

In the type section specifications of abstract data types are defined. The SYNTHESIS language supports a comprehensive type system based on a recursive composition of the type constructors. Functions are used for support of methods, assertions, derivable entities. A function type specification can contain a predicative specification of function or a reference to a function implementation if required. Function implementations can be given by programs containing rules expressed by SYNTHESIS formulae or by conventional programs.

Types are organized into a multiple subtyping hierarchy. Classes combining properties of a type and of a set form a multiple hierarchy providing for a generalization/specialization relationships. Orthogonal to the subtyping relationship for types and to the generalization relationship for classes is the classification relationship. Being object themselves, types and classes can become instances of another more general classes called metatypes for type classification and metaclasses for class classification. Thus hierarchical classifications can be formed. It is essential that one and the same object can belong to several classes (one class to several metaclasses). Metaclasses are useful for introducing of generic concepts common for several attributes, types, classes. Deductive rules and assertions can be introduced in metaclasses as the generic rules for their instances. Thus metaclasses provide for better structured descriptions of the application domains (mediators) and of the information resources.

A kind of a resource (e.g., database resource, knowledge base resource, mediator resource) to which a class definition belongs can be declared by metaclasses. If required, the attributes of an object can be declared to possess all properties of classes. These classes become instances of the attribute categories introduced by means of metaclasses of associations and of generic attributes of metaclasses. Such attributes can take concrete forms in classes that are instances of a given metaclass. Assertions constraining possibilities of their concretizations can be associated with the generic attributes.

With each class, with each attribute of a class or with a resource represented by a self-defined object a collection of assertions can be associated. In particular, it is possible to apply built-in assertions, assertions represented by a formula of the SYNTHESIS language or assertions defined as predicates by means of a SYNTHESIS program. Assertions provide for:

- derivation of attribute values;
- checking of assertions realizing predetermined actions over abstract values (e.g., over the instances of a class);
• definition of an action that should be enforced on a certain event and/or condition.

A collection of assertions that are defined separately of type and class specifications can be defined in an assertion section of a module. Usually such assertions act over sets of types (classes). Modules that contain only assertions (alongside with the imported modules defining context of such assertions) are allowed. Thus a collection of assertions can be flexibly structured by means of their associations with attributes of abstract values, with types and classes, with sets of specifications defined in one or in several modules.

Specifications in the SYNTHESIS language take a form of frames. A mediator/information resource specification module is a named collection of frames (a world). Thus, metainformation can become accessible by a program. For the context formation it is possible to include into a context an arbitrary set of worlds (modules).

4 Frame representation and manipulation facilities

4.1 General characteristics of the frame language

Basic objective of the language is a representation of conceptual and semi-structured information on an application domain. In the frame language a knowledge base is considered as a collection of frames interrelated by means of binary associations. There is no means to represent classes of concepts or entities in the frame language: only individual instances can be represented. The facilities for specification of types and classes have higher level and extend the frame language upwards.

In the SYNTHESIS language frames are used also for the metadefinition of other facilities of the language. This results in the specification of all the mediator and information resources metainformation in the frame language taking a form of contexts containing mediator or information resource specification modules. In particular, such contexts are helpful in a process of integration of conceptual (ontological) specifications of various information resources in the context of a mediator.

And finally, the frame language is used also for the external representation of constants of various SYNTHESIS types (including abstract data type values (section 6)) for input/output purposes, to represent them inside of formulae and functions of the language. In particular, self-defined objects (objects that exist separately and not belong to any class) are also
4.1 General characteristics of the frame language

represented by frames. Frames can be combined into collections (worlds or contexts). These collections can include frames having arbitrary structure.

Generally, frames are abstract values, their type $\text{Tframe}$ is a subtype of the abstract value type $\text{Taval}$ (subsection 6.2). Slots of the frames are considered as functions appeared as $a : C \rightarrow V$, (where $a$ is a slot name, $C$ is a context or a world name, denoting a collection of frames, $V$ is a collection of the slot values). Domains of such functions are collections of abstract values (here the collections of frames) in a context or in a world. At the same time defining a slot as a component of a frame specification a notation $a : v$ is used where $v$ is a value belonging to $V$ or to a subset of $V$.

In case of slots defined as a stored functions of the $a : T$ kind where $T$ is a type of the slot values, on usage of the function designator $a(o)$ it is implicitly assumed that the generic functions get or set are applied extracting or modifying slot $a$ value of the frame (abstract value) $o$. An internal frame representation is not fixed by the language and is considered to be encapsulated.

Each frame, its components (slots and slot values) as well as its associations with other frames are considered as existing in time. To model that at most two temporal constants for temporal modeling can be associated with each frame, any of its components and arbitrary associations. They define a time slice of existence (time duration when according to user beliefs an entity or an attribute of an application domain preserves its value) and a time slice of belief (time duration when a system "believes" into an existence of an entity or of an attribute value). Thus, the whole history of the application entities can be kept: on update of a frame or of its component. Their preceding state with the respective temporal associations is stored in the frame base. The time category support introduces an additional structuring dimension for the application domain based on temporal relationships such as before, after, during, meets.

Frame language is tightly related to the the logic programming facilities of the SYNTHESIS language used for representation of functions and intensional components of the knowledge base. This connection is reflected in the fact that frames and their components are objects of assertions and logic rules of the language alongside with other abstract values. On the other hand, collections of logic rules of the language can be used as slot values that are intended for their use as functions, assertions (e.g., consistency constraints) or for deductive inference. It is essential that such collections of rules are values themselves that exist in time analogously to other frame components.
An additional metainformation can be associated with frames, slots and values. It also takes a form of a frame. The metainformation provides for additional capabilities in specifying of the complex concepts. The frame model provides for user-defined relationships (binary associations), to keep forward and inverse relationships.

Frame base can have a modular hierarchic structure of contexts. Collections of modules (worlds) expressed in the frame language are located in nodes of this structure. Frame "visibility" relationships are expressed by associations between contexts.

A complete set of syntactic rules defining the frame language is given in the Appendix A.

4.2 Rules for the syntactic definition of the SYNTHESIS language

The syntax of the SYNTHESIS language is defined by means of the extended Backus-Naur notation. Nonterminal symbols are given in the angular brackets: <a string of characters>. To separate the left hand part of a rule from the right hand part the metasymbol "::=" is used. Alternative right hand parts of a rule are separated by the vertical bar (|). Optional fragments of the grammar are taken into square brackets. Repeating constructions are taken into square brackets with dots following them. This gives an ability to denote arbitrary (including zero) repeating of a construct.

Identifiers and variable denotations are constructed in the language according to the following schema:

\[
\begin{align*}
< \text{identifier} > &::= < \text{letter} > [ [ < \text{underscore} > ] < \text{letter} > ] \\
& \quad [ [ < \text{underscore} > ] [ < \text{digit} > ] ] ... \\
< \text{variable} > &::= < \text{identifier} > | _{ [ < \text{identifier} > ] }
\end{align*}
\]

A variable denoted by the _ character is called anonymous.

4.3 Basic elements of the frame language

4.3.1 Frames

A frame is considered to be a symbolic model of a certain entity or a concept. A frame is represented as a set of attributes called slots being used for a description of properties of an entity or a concept. An order of slots given for a frame specification is kept by the system. On update it is possible to keep a required order of the frame slots. On a data search in the frame base such order can be taken into account (if required). To each
4.3 Basic elements of the frame language

A slot is a collection of values that can be associated; each value being generally an abstract value of arbitrary type defined in the SYNTHESIS language (e.g., another frame) or a collection of the logic language formulae.

Any frame can acquire a unique identifier to which the built-in slot self corresponds. The value of this slot cannot be modified. At the same time the self slot can be used in the predicates and formulae of the language as other ordinary slots.

The frame is an important concept of the language: any language entity (including objects) is uniformly represented by means of frames. It means that each such entity possesses properties of frames given by the type \( T_{\text{frame}} \) (subsection 6.2). The behavior of instances of the type is given by functions as for the other abstract values.

Frame is always represented in figure brackets \{ and \}. Identifiers of nested frames can be omitted. Slots and their values are separated by a colon. The values of a slot are separated by a comma. Semicolon terminates the listing of values of a slot. It is important that any frame component (identifier, slots, values) can be omitted. In case when a value of a slot is absent, semicolon is provided immediately after a colon. Values in frames containing no slots belong to implicit universal built-in slot.

The following forms represent well-formed frames (an external representation of a frame as a certain constant is given):

- a frame having no slots:
  
  \[
  \{ \text{< frame identifier >};
  \}
  \]

- a frame having no name:
  
  \[
  \{
  \text{< slot1 >:< list of slot1 values >};
  \text{< slot2 >:< list of slot2 values >};
  ...
  \text{< slotM >:< list of slot1 values >};
  \}
  \]

- a frame having no values:
  
  \[
  \{ \text{< frame identifier >};
  \text{< slot1 >:;}
  \text{< slot1 >:;}
  ...
  \text{< slotM >:;}
  \}
  \]
• a frame containing values without any slots:

\[
\{( \text{frame identifier} ); \quad \text{value}; \quad \text{value}; \quad \text{value}; \quad \text{value} \}
\]

In frames containing values without any slots, a blank is allowed as a separator between values (alongside with a semicolon). Such frames can be conveniently used for representation of texts, hierarchical and other similar structures. In such structures, nested frames are also frames without any slots. These forms alongside with the complete form of frames can be arbitrarily intermixed for a frame representation. Besides that, these forms can also be used in metaframes (subsection 4.3.2).

Variables in frames are used for indication that content of their certain component can be arbitrarily chosen. Variables can get values of arbitrary types. Variables in frames are treated locally (inside of the frame and of all its enclosed frames). Variables denoted by an identifier starting with the underscore character are allowed as frame or slot identifiers and can be used for slots values as well. A frame identifier accompanied by parameters is used for a concretization of a frame by passing the parameters to it.

### 4.3.2 Metaframes

Any frame component (a frame itself, a slot or a value) can have additional metainformation associated to it. The metainformation is contained in the values of slots of a metaframe associated with a certain frame, a slot or a value. Depending on a component, such metacomponents are called metaframes, metaslots and metavales respectively. In the sequel, the term “metaframe” will be used generically when it is not important with which part of a frame the metainformation is associated.

Metaframes, metaslots and metavales are denoted as it is shown below:

```
metaframe            metaframe       end
metaslot             metaslot       end
metavalue            metavalue      end
```

Metaframes can be associated with a frame or with a frame components in two ways.

**First way.** The identifier of a metaframe defined separately is inserted between the key words in the respective construction of another frame. In
4.3 Basic elements of the frame language

this case the first frame will play role of metframe, metslot or metavalue. For example,

- metaframe:
  
  \{ a;  
  metframe b ; end  
  s : v;  
  \} ;  

- metaslot:

  \{ a;  
  s : v;  
  metslot b; end  
  \} ;  

- metavalue:

  \{a;  
  s : v metavalue b; end;  
  \} 

Second way. A complete definition of metframe, metslot or metavalue is embedded into a given frame. In this case a metframe may not contain an explicit identifier, or it can be named if it is referred from another frame base components.

A structure of an embedded metframe representation looks as follows:

\{ [ < frame identifier > ];  
metaframe [ < frame identifier > ];  
< slots and values of metaframe slots >  
end  
< slots and values of frame slots >  
\}

An example of a metframe content:

metframe < frame identifier >;  
in : ;  
history : ;  
end  

A slot in defines names of classes and is used when a frame on creation or as a result of certain transformations becomes an object of certain classes (section 7). In a metframe any subset of the slots listed above
is allowed as well as any other slot. A representation of the temporal components in metaframes (slot history) is considered in subsection 4.4.

Metaslots are defined analogously to metaframes. Built-in slots that can be used in a metaslot definition alongside with any other slots (e.g., slots representing application domain attributes) are shown in the following structure:

```
metaslot < frame identifier >;
    history : ;
    range : ;
    cardinality : ;
    default : ;
end
```

A value of the slot range defines a set of values of a slot with which the metaslot is associated. On setting of a value of this slot the system checks if this value belongs to the indicated set. If not, a value is not included into the slot.

The cardinality slot contains as its value an expression that defines admissible maximal number of values of the slot. When new value is added to the slot, system checks if the number of values of the slot does not exceed the value of the expression given by the slot cardinality. If this is not the case, a value is not included into the slot.

The default slot defines a value of the primary slot that should be set by default. If the slot range is specified then the default value must be included into a set of the admissible primary slot values.

An example of a content of a metavalue follows (an arbitrary frame can also be associated as a metavalue frame).

```
metavalue < frame identifier >;
    history : ;
end
```

### 4.3.3 Binary associations

Separate frames in the frame base can be linked with each other by means of binary associations. For example, if a concept is a refinement of another concept then the frames representing those concepts should be kept in a association that reflects the respective relationship.

To set an association between the frame A and the frame B a slot is included into the frame A with a name of an association required. An identifier of the frame B should become a value of this slot.
Association names are assumed to be unique in a context.

In the frame language it is allowed to define arbitrary binary associations filling in the following frame:

\[
\{ <\text{frame identifier}>; \\
in: \text{association}; \\
domain: ; \\
range: ; \\
inverse: ; \\
association\_type: ; \\
\}
\]

In the slot domain a collection of frames is specified that constitutes an association domain. In the slot range a collection of frames is specified that constitutes an association range. A collection of frames can be specified by means of a formula over the frame base that defines a collection of frame values: \(\{<\text{formula}>\}\). An absence of the slot domain or the slot range means that the respective collection can include arbitrary frames.

An association identifier specified in the slot inverse is considered to be an inverse association to those being defined. An identifier of the inverse association can be used equally with the identifier of a direct association.

The association\_type slot defines a kind of the association fixing ranges - pairs of numbers defining a range (minimal and maximal admissible value) of the number of frames in the association range that can be associated with every frame in the association domain (the first range) and of the number of frames in the inverse association range that can be associated with every frame in the inverse association domain (the second range).

4.4 Facilities supporting the temporal category of the language

4.4.1 Temporal assignments of frames and of their components

The objective of support of the temporal category in the frame base is a representation of an information on evolution of the real world entities. This objective determines a requirement to keep in the frame base the entities and attribute values that are not valid further. This means that the frame base stores an information that is associated with the application domain states in different discrete moments of time. An information for each of such moment should be consistent. Entity data are kept only in those discrete moments of time that correspond to a certain event in the entity "life" that leads to a change of its state. The entity data that
are not explicitly stored in the interval between the events should be in-
ferrable.

For the representation of time moments and intervals the temporal
constants are included into the frame language, such as a constant of a
time moment and a constant of a time slice:

\[
< \text{constant of a moment} > ::= < \text{year} > # < \text{month} > # < \text{date} > \\
\quad [ < \text{hour} > : < \text{minutes} > : < \text{seconds} > : [ . < \text{fractions of second} > ] ] | \\
\quad < \text{hour} > : < \text{minutes} > : < \text{seconds} > : [ . < \text{fractions of second} > ] | \\
\quad \text{current} | \inf | -\inf \\
\]

\[
< \text{constant of interval} > ::= \text{interval}( < \text{years} > # < \text{months} > ) | \\
\quad \text{interval}( < \text{days} > [ < \text{hours} > : < \text{minutes} > : < \text{seconds} > ], < \text{fractions of second} > ] ) | \\
\quad \text{interval}( < \text{hours} > : < \text{minutes} > : < \text{seconds} > [ . < \text{fractions of seconds} > ] ) | \inf \\
\]

\[
< \text{constant of a time slice} > ::= \\
\quad < \text{constant of initial moment} > / \\
\quad < \text{constant of terminating moment} > | \\
\quad < \text{constant of initial moment} > / < \text{empty} > | \\
\quad < \text{empty} > / < \text{constant of terminating moment} > \\
\]

Here metavariables year(s), month(s), date, days, hour(s), minutes,
seconds, fractions of second denote integers representing corresponding
moment of time or an interval. In the constant of a moment or of an
interval it is allowed to use an arbitrary sequence of adjacent fields in ac-
cordance with the syntactic rules. For proper interpretation of a constant
it is necessary to provide explicitly the separators ‘#’, ‘,’ or ‘.’ surrounding
the constant from the left and from the right.

For instance, constants of a moment are: 10# # (the tenth year),
#10# (the tenth month), # #10 (the tenth date), 10: (ten hours), :10:
(ten minutes), ::10 (ten seconds); constants of an interval are: interval
(10#) (ten years), interval (#10) (ten months). The constant current
denotes current time moment, the constants inf and -inf denote infinite
time moment in the future and in the past respectively. The constant
inf can be used also to denote an infinite time interval. Empty values
in a constant of a time slice mean that the respective components of the
constant are not defined. This allows to denote by means of time slice
constants the discrete moments of time.

The built-in functions (start(< time slice >) and fin(< time slice >))
where time slice denotes a constant of a time slice or a variable define
the initial and terminating moments of the time slice. With a frame or a frame component (a slot or a value) a temporal information can be associated that is kept in a specific history slot of a metaframe, metaslot or a metavalue. A value of the history slot is a temporal frame or generally a collection of temporal frames. The structure of a temporal frame follows:

\[
< \text{temporal frame} > ::= \{ \text{valid} : <\text{constant of a time slice} > ; \\
\text{belief} : <\text{constant of a time slice} > \}
\]

A temporal frame contains two temporal constants: valid and belief, that define respectively a time slice of existence (during this time an application entity or its attribute corresponding to a frame or its component according to the user beliefs preserves its value) and a time slice of belief (during which the system “believes” into the existence of an entity or of its attribute).

The value of the first time slice is provided by a user’s temporal assignment during frame specification or modification; the value of the second time slice is provided by the system. If a frame or its component corresponds to a point event, a constant of a terminating moment of the valid temporal constant is empty. At the same time it is assumed that properties created by the point events are persistent. It means that their interval of existence is extended to the right maximally in the range of the temporal constraints provided. If there are no constraints, a constant of a terminating moment is assumed to be equal inf. This assumption remains valid until the system will not get clarifying data.

Syntactically a temporal assignment is expressed as follows:

\[
< \text{temporal assignment} > ::= ( \text{at} <\text{constant of a time slice} > )
\]

Temporal assignment of a frame:

\[
< \text{frame} > ::= \{ [<\text{frame identifier} > ; | <\text{temporal assignment} > ; ] \\
<\text{metaframe} >| <\text{frame body} > \} 
\]

Temporal assignment of a slot:

\[
< \text{frame body} > ::= <\text{slot} > : <\text{value} > ; \\
<\text{temporal assignment} >| <\text{metaslot} > | ...
\]

Temporal assignment of a value:

\[
< \text{entire value} > ::= <\text{value} > | <\text{temporal assignment} >
\]

Defaults:
• if the time slices of existence for slots and values are not given on creation of a frame then they are assumed to be equal to that of the frame;

• if on creation of a value its time slice of existence is not given then it is assumed to be equal to that of the respective slot.

4.4.2 Reflection of entity evolution in the frame base

On a temporal assignment the system provides a value for the slot belief using the following procedure. start(belief) is set equal to a transaction start time. fin(belief) is set equal to inf (the system assumes an interval of belief to be an infinite half-interval due to the hypothesis of persistency of entity properties). fin(belief) is set by the system equal to the starting time of another transaction that updates the frame component or its time slice of existence. It means that the system does not "believe" further into the respective state of the component. If the component is being deleted or updated by a user, no other modifications are attempted (the component is just kept in the system). On update of a component its new value is created with the respective temporal assignment. Therefore, belief generally means:

• for a frame: a period of time from frame creation until its deletion or update of the frame time slice of existence by a user;

• for a slot: a period of time from inclusion of the slot into a frame until its exclusion from the frame or update of the slot time slice of existence by a user;

• for a value: a period of time from the inclusion of the value into a collection of slot values until its erasure or update of the value time slice of existence by a user.

It is necessary to note that on update of the time slice of existence of any component or on repeated creation of a frame or a slot with the same identifier or on a repeated inclusion into a slot of a value that existed earlier it is sufficient (alongside with the modifications defined above) to add a new temporal frame to the history slot of the respective component.

Thus every component (a frame, a slot, a value) can have a collection of values in the slot history. However, only in one of the temporal frames the value fin(belief) can be equal to inf. This temporal frame defines current state of a component. Other temporal frames refer to the history.
4.4 Facilities supporting the temporal category of the language

On setting of any association between frames, with each particular frame relationship as with a slot value a metavalue containing temporal frames in the slot history is associated. It indicates in the temporal frame:

- a time slice of existence of a particular relationship provided by a user;
- a time slice of belief given by the system.

One and the same relationship can be set and erased many times: all history of the association is kept in the history slot.

### 4.4.3 Temporal associations

Assume that \( x \) and \( y \) provide for unique identification of temporal frames (that can be associated to a frame, a slot or a value) and at the same time define what time slices (of existence or of belief) are considered. For such identification it is sufficient to use names valid or belief qualified by an identifier of a variable referencing a frame, a slot or a slot containing values (subsection 4.7). The temporal predicates listed below provide for the definition of associations between discrete moments of time and between time slices referenced by \( x \) and \( y \). The predicates become true when the following conditions will be satisfied:

- **before\((x,y)\)**: a discrete time moment \( x \) precedes a discrete time moment \( y \), or a discrete time moment \( x \) precedes a time slice \( y \), or time slices \( x \) and \( y \) are not overlapped and \( x \) precedes \( y \);
- **equals\((x,y)\)**: \( x \) and \( y \) denote the same discrete time moments or the same time slices;
- **after\((x,y)\)**: a discrete time moment \( x \) goes after a discrete time moment \( y \), or a discrete time moment \( x \) appears later than a time slice \( y \), or time slices \( x \) and \( y \) are not overlapped and \( x \) appears later than \( y \);
- **during\((x,y)\)**: a time slice \( y \) includes a discrete time moment \( x \), or a time slice \( y \) includes a time slice \( x \) (inclusion means in this case that starting and terminating moments of \( x \) and \( y \) do not coincide);
- **starts\((x,y)\)**: a discrete time moment \( x \) coincides with a starting moment of a time slice \( y \), or time slices \( x,y \) are initiated simultaneously and a terminating moment of \( x \) is included into \( y \);
• finishes\((x, y)\) : a terminating moment of a time slice \(y\) coincides with a discrete time moment \(x\), or terminating moments of time slices \(x, y\) coincide and a starting moment of \(x\) is included into a time slice \(y\);

• meets\((x, y)\) : a time slice \(x\) is initiated before time slice \(y\) and terminating model of \(x\) coincides with an initial moment of \(y\);

• overlaps\((x, y)\) : a time slice \(x\) is initiated before a terminating moment of a time slice \(y\) and a time slice \(y\) is initiated before a terminating moment of a time slice \(x\);

• before_starts\((x, y)\) : a time slice \(x\) is initiated before a time slice \(y\) (the time slices may be overlapped);

• before_ends\((x, y)\) : a terminating moment of a time slice \(x\) appears before a terminating moment of a time slice \(y\) (the time slices may be overlapped).

By means of the predicates introduced more complex predicates can be constructed:

\[
\text{in}(x, y) : \neg \text{starts}(x, y) \lor \text{during}(x, y) \lor \text{finishes}(x, y)
\]

\[
\text{ge}(x, y) : \neg \text{after}(x, y) \lor \text{equals}(x, y)
\]

\[
\text{le}(x, y) : \neg \text{before}(x, y) \lor \text{equals}(x, y)
\]

\[
\text{ne}(x, y) : \neg \text{after}(x, y) \lor \text{before}(x, y)
\]

Predicates in, ge, gt, le, lt, eq, ne can be used for the comparison of two time slice values. Temporal associations are assumed to be declared similarly to any other frame associations. Setting of temporal associations is allowed for frames, metaframes, metaslots or metavalue. Interpretation of temporal associations set on any level is assumed to be the same.

Temporal associations express certain facts (before, after, during, ...) for the interrelated entities meaning that the respective time slices or their components should be interrelated by this association. It is essential that temporal frames denoting the respective temporal constants (valid, belief) are used for establishing of the temporal associations.

It is essential that temporal associations can be used for constraining of other binary frame associations. For that it is not obligatory that the constants defining a period of existence of a frame or of its component would be defined in a temporal frame at the moment of setting of a temporal association. However, on setting of the association as well as afterwards on update of the values of the components of the time slices of existence the respective constraint imposed by the temporal association
4.5 Worlds and contexts

Frames that are contained in the frame base are subdivided into the subcollections (worlds). Every world can be seen as separate and relatively independent section of the frame base that is used for the formation of the frame collections and for their manipulation. A world is represented by a frame having the predetermined structure:

\{
  < world identifier >;
  in : world;
  < collection of frames >
\}

Generally a world is a unit (a module) for representation, compilation, storage and manipulation of the frame collections. If variables are used in a world, the respective type definitions should be included into the same context the world belongs to. Such context is used for compilation and interpretation of the world. Each frame and all its nested frames should belong to one and the same world. Collections of worlds can form contexts. A context is also represented by a frame having the predetermined structure:

\{
  < context identifier >;
  in : context;
  member : < list of world or context identifiers >
\}

The association member (member_of is its inverse association) is used for establishing of a membership of frames (or contexts) in contexts. Contexts in their turn can participate in various associations to form hierarchical or network structures. Context (world) names can be used as parameters of functions and predicates providing for localization of the respective functions in the context (or in the world) given. For data search a context (or a world) explicitly indicates a space for the search by means of a certain formula.
Frame base manipulation facilities

In this subsection the functions for frame base manipulation are defined. The functions are presented in a form suitable for their use in conventional programming languages. Variables used in functions have types defined in section 6. Functions applicable to frames are also applicable to objects that can be considered as frame specializations (section 6).

1. \(<\text{define}_{\text{world}}>:\text{define}_{\text{world}}(\text{<world identifier}>)\)
defines a new world.

2. \(<\text{delete}_{\text{world}}>:\text{delete}_{\text{world}}(\text{<world identifier}>)\)
deletes the world given.

3. \(<\text{define}_{\text{frame}}>:\text{define}_{\text{frame}}(\text{<new frame>})\)
   \([\text{<temporal assignment>}, \text{<world>}, \text{<variable>}}]\)
\(<\text{new frame}>:=(\text{<frame>\mid<variable>})\)
defines a new frame in a given world, the variable takes frame identifier as its value.

4. \(<\text{define}_{\text{slot}}>:\text{define}_{\text{slot}}(\text{<frame selector>},\text{<variable>})\]
\([\text{<temporal assignment>}, \text{<manner>}}]\)
\(<\text{frame selector}>:=(\text{<formula>\mid<variable>})\)
defines a new slot in frames relevant to the frame selector. If the selector is given by a variable, the variable value should be a frame in which a slot should be directly included. The variable used as the function parameter should provide a name of a new slot as its value. The formula should identify frames into which new slot should be directly included.

5. \(<\text{define}_{\text{value}}>:\text{define}_{\text{value}}(\text{<frame selector>}, \text{<variable>})\]
\([\text{<value>\mid<temporal assignment>\mid<manner>}}]\)
\(<\text{manner}>:=(\text{append} | \text{overwrite})\)
defines a new value in a slot in frames relevant to the frame selector. The variable provides the slot name as its value. The frame selector should identify frames into which the required slot is directly included. The parameter \(<\text{manner}>\) defines a manner for the value definition: value overwrite or append to the existing ones.
6. \(< \text{assign} \text{.value} > ::= \text{assign}(< \text{variable} > (< \text{list of parameters} >))\)

initializes variables in a frame defined by the variable, is used for the concretization of frames. The number of elements in the list of parameters and their order should correspond to those of the list of formal parameters related to a frame identifier.

7. \(< \text{delete.frame} > ::= \text{delete.frame}(< \text{frame selector} >)\)

deletes frames relevant to the frame selector.

8. \(< \text{delete.slot} > ::= \text{delete.slot}(< \text{frame selector} >, < \text{variable} >)\)

deletes from frames relevant to a formula a slot name of which is defined by the variable. The frame selector should identify frames into which the required slot is directly included.

9. \(< \text{delete.value} > ::= \text{delete.value}(< \text{frame selector} >, < \text{variable} >, < \text{value} >)\)

deletes a given value of the slot name of which is given by a variable from the frames relevant to the frame selector. The frame selector should identify frames into which the required slot is directly included.

10. \(< \text{iterator} > ::= < \text{value iterator} >|< \text{frame iterator} >\)

\(< \text{value iterator} > ::= \text{find}(< \text{variable} >, < \text{formula} >)\)

\(< \text{frame iterator} > ::= \text{findframe}(< \text{variable} >, < \text{formula} >)\)

\(< \text{iterator advancing function} > ::= \text{getnext}(< \text{variable} >)\)

\(< \text{frame exhausting predicate} > ::= \text{eoc}(< \text{variable} >)\)

There are two variants of iterators. The first variant is used for iteration of a collection of values of the product type (section 6) components of which have names and types defined by names and types of the resulting variables of the formula. The second variant is used for iteration of a collection of frames of the frame base that are relevant to a formula. The variable for the first variant takes for its value current value of the product type and for the second variant takes for its value current frame. Several iterators can co-exist in a program; the iterators being distinguished by their names. In the iterator advancing function the same variable name is used as in the iterator. The frame exhausting predicate becomes true if during the iterator execution or during the last execution of \text{getnext} a relevant
frame (or a value) has been retrieved. In this case the variable that is the first parameter of the iterator (or of the iterator advancing function) takes as its value the first (next) frame (or a value of the product type). When relevant frames (values) are exhausted, \texttt{eoc} becomes false.

11. Functions within frame slot manipulation

\[
<\text{slot iterator}> ::= \text{first_slot}(<\text{iterator variable}>, <\text{variable}>)
\]

Slot iterator makes possible to browse through slots of a frame given by the variable having the frame as its value. The iterator advancing function and the slot exhausting predicate are the same as for the frame iterator. The iterator variable gets as its value the following sequence:

\[
<\text{slot identifier}>, <\text{list of values}>, <\text{temporal assignment}>
\]

For analysis of the components of this sequence the following set of functions can be used:

\[
\begin{align*}
\text{slot\_name} & (<\text{variable}>, <\text{iterator variable}>) \\
\text{slot\_value} & (<\text{variable}>, <\text{iterator variable}>) \\
\text{slot\_time} & (<\text{variable}>, <\text{iterator variable}>)
\end{align*}
\]

To insert new slot immediately before those just found by an iterator the following functions can be applied:

\[
\text{insert\_positional}(<\text{iterator variable}>, <\text{variable}>, <\text{entire value}>)
\]

The variable has a slot name as its value. For further analysis of values obtained by the function \texttt{slot\_value} the slot value iterator is required:

\[
<\text{slot value iterator}> ::= \\
\text{first\_value}(<\text{iterator variable}>, <\text{variable}>)
\]

Parsing of the obtained pair (slot value, temporal assignment) is realized by the functions:

\[
\begin{align*}
\text{value} & (<\text{variable}>, <\text{iterator variable}>) \\
\text{time\_of\_value} & (<\text{variable}>, <\text{iterator variable}>)
\end{align*}
\]
For analysis of a kind of a value (a frame, a value, a temporal constant) the following function is used:

\[
\text{type_of_value}(\langle \text{type variable identifier} \rangle, \langle \text{variable} \rangle)
\]

To modify a current value the following function is used:

\[
\text{replace_value}(\langle \text{iterator variable} \rangle, \langle \text{entire value} \rangle)
\]

12. Context manipulation functions

Functions:

\[
\begin{align*}
\text{create_ctx}(\langle \text{variable} \rangle [\langle \text{list of world identifiers} \rangle]), \\
\text{extend_ctx}(\langle \text{variable} \rangle, \langle \text{list of world identifiers} \rangle), \\
\text{narrow_ctx}(\langle \text{variable} \rangle, \langle \text{list of world identifiers} \rangle)
\end{align*}
\]

are used for creation of a context, for extending (narrowing) a context. The variable has an identifier of a context as its value. The list of world identifiers is defined as follows:

\[
\langle \text{list of world identifiers} \rangle ::= \langle \text{variable} \rangle | \\
\{ \langle \text{world identifier} \rangle [; \langle \text{world identifier} \rangle] \ldots \}
\]

4.7 Formulae over the frame base

4.7.1 Formulae construction rules

Predicates and atoms (section 9) defined by means of contexts (Appendix A) are used as components of formulae over the frame base. The frame predicates linked with each other and with an atom by the symbols & , | ,  form a subformula. Every predicate in a subformula acts over collection of frames determined by the atom (a world or a context). A subformula can be satisfied by a subcollection of frames determined by the atom. Logic programs over the frame base can be specified in accordance with the rules of the section 9.

4.7.2 Temporal assignment of formulae

On specifying of the formulae it is necessary to indicate the temporal constraints defining a temporal slice of the frame base that should be considered in a given formula. Current state of the frame base (a collection of frames having terminating moments for belief equal to inf ) can
be an example of such slice. Such slices can be defined by means of tem-
poral predicates formulated with respect to the attributes valid and belief
and associated with various predicates in a formula (with each predicate
a condition determining necessary temporal slice should be associated).
Usually the temporal slice is one and the same for all the predicates of
a formula. In such cases it is inconvenient to repeat one and the same
temporal predicate for all predicates of a formula.
To specify temporal slice for the whole formula special constructs are
included into the language:

< formula > |on < temporal selector >|as_of < temporal selector >|

< temporal selector > ::= < constant of a moment >|
| < constant of a time slice >|
| < variable >| < temporal function >|
| < temporal association identifier > (< parameter >)
< temporal function identifier ::= year | month | day | hour |
| minute | second | fraction | start | fin | intersect |
| union | extend | sp_convert

A construct on defines a temporal slice for the attribute of existence
valid, a construct as_of defines a temporal slice for the belief attribute. A
selector defines a value of a temporal constant (of a discrete time moment
or of a time slice). Temporal functions are defined in the Appendix B.
The constructs on and as_of that do not use temporal associations in
the temporal selectors act so that in subformulae where temporal predi-
cates associated with the attributes of existence or belief are not defined,
the predicate
in(valid, < temporal selector >)
or
in(belief, < temporal selector >)
are assumed if the selector has a form of a time slice or the predicates
in(< temporal selector >, valid)
or
in(< temporal selector >, belief)
are assumed if the selector has a form of a discrete time moment. By
default the selector is assumed to be defined as current for as_of as well as
for on

If temporal associations are used, the parameter corresponds to the
second argument of the associations defined in subsection 4.4.3. A tem-
poral assignment of a frame or of its component relevant to the corre-
sponding subformula is assumed to be the first argument.

5 Mediator and information resource specification modules

A mediator or information resource specification module (further "module" for short) is the basic unit of specification, import and compilation in the SYNTHESIS language. Each module contains either a mediator or a canonical information resource specification. A specification given by a module can be considered belonging to one of the levels: local or global.

\[
\text{< module definition } ::= \{ \text{< header } \text{< module specification }> \}
\]

\[
\text{< header } ::= \{ \text{< module identifier }> ; \\
\text{in : module, < metaclass name list > ; }
\text{[params : \{< formal parameter list >\} ; ] }
\text{[module_level :< module level >; ] }
\text{[schema :< identifier >; ] }
\]\n
\[
\text{< module level } ::= < \text{identifier }> \\
\text{< formal parameter list } ::= < \text{formal parameter } > |
\text{< formal parameter > ; < formal parameter list > }
\]

A module kind that should be given in the metaclass name list defines a built-in metaclass to which a module belongs. A resource module kind indicates that a module contains canonical resource component specifications. A mediator module kind indicates that a module contains mediator component specifications.

A module can be parameterized. Such specification denotes a generic module that makes possible to form concrete modules as a result of actual parameter substitution dynamically during the module import or in the process of compilation. Usually type variables are applied as formal parameters that get types as their values for the module concretization (section 6). The parameters are allowed that take values of arbitrary types of the SYNTHESIS language.

An identifier of a declared entity everywhere in the module specification is treated as its explicit identifier or as an identifier of a formal parameter of the module that is used for subsequent concretization. A name of a resource schema is defined if the module is combined with other modules into the mediator or information resource schema. The schema
identifier in the module header defines the schema into which the module should be included.

\[
\text{< schema >::= \{< schema name >; in : schema; }
\]
\[
\text{[params : \{< formal parameter list >\}; ]}
\]
\[
\text{< module specification >}
\]
\[
\text{[: < module body >]}
\]

A specification of a module can contain up to five different sections. Any combination of sections is allowed. Any section can be omitted.

\[
\text{< module specification >::=}
\]
\[
\text{[import :< import list >; ]}
\]
\[
\text{[rename : \{< renaming list >\}; ]}
\]
\[
\text{< frame section >}
\]
\[
\text{< type section >}
\]
\[
\text{< function section >}
\]
\[
\text{< resource component specification section >}
\]
\[
\text{< resource component assertion section >}
\]
\[
\text{< import list >::=< list element >|}
\]
\[
\text{< list element >, < import list >}
\]
\[
\text{< list element >::=< module name >|< module concretization >}
\]
\[
\text{< module name >::=< module identifier >|}
\]
\[
\text{< class identifier >}
\]
\[
\text{< module concretization >::=< frame name >|}
\]
\[
\text{\{< module identifier > [:< module identifier >];}
\]
\[
\text{params :< actual parameter list >\}}}
\]
\[
\text{< actual parameter list >::=< actual parameter >|}
\]
\[
\text{< actual parameter >, < actual parameter list >}
\]
\[
\text{< actual parameter >::=< value >|< typed variable >| \{\{< expression >\}\} |}
\]
\[
\text{\{\{< logic program >\}\} |< actual type >}
\]
\[
\text{< actual type >::=< type name >}
\]
\[
\text{< type name >::=< type identifier >|< type variable identifier >|}
\]
\[
\text{< metatype identifier >|< formal parameter identifier >|}
\]
\[
\text{< class name > . < component - kind > |.nonobject}
\]
\[
\text{< component - kind >::= own | inst | instinst}
\]
\[
\text{< type identifier >::= [< module identifier > . ] < identifier >}
\]
\[
\text{< class name >::=< class identifier >|< type variable identifier >|}
\]
\[
\text{< metatype identifier >|< formal parameter identifier >|}
\]
\[
\text{< metaclass identifier >}
\]
\[
\text{< metatype name >::=< class identifier >|< type variable identifier >}
\]
\[
\text{< class identifier >::= [< module identifier > . ] < identifier >}
\]
A module can import an arbitrary collection of other modules, each of them containing any set of sections. It is allowed to import specifications of separate classes from a module (with their concretization, if required). It is essential that in a process of import a concretization of the imported module can take place, so that objects of the module in which import is indicated can be passed as parameters into the module that should be imported and made concrete. In the concretization expression places of formal parameters that are not subjected to concretization should be taken by anonymous variables and places of formal parameters that are subjected to concretization should be taken by actual parameters. It is important to note that import is a function that eventually leads to a formation of the frame context (section 4).

In a module concretization <module name> after a colon a name of a module is given that should be used as the result of the concretization. If such name is not provided, a name of the concretization result coincides with the name of a module (or a class) that is subjected to a concretization. A renaming list provides for changing of names of entities specified in a module in process of import. A name given to the left of ':' in the list defines new entity name. Definition of a type designator will be given in the next section.

For convenience it is allowed to locate frames defining components of module specification (such as types, classes, assertions and their subcomponents) in any place of the module. An arbitrary collection of frames can be defined in the frame section of a module. A module body contains a specification of implementation of the module entities. Facilities for the module body definition are not fixed by the SYNTHESIS language and should be defined in process of the language implementation.

Class and type specifications in a module are considered to be abstract and completely separated from the implementation. A collection of different implementations can correspond to such abstract specifications.

When a class name is used for the type name, this name (without a qualifier own, inst, instinst) indicates a type of a set of objects corresponding to the class. A qualifier inst indicates an instance type specified for the class. A qualifier own means a type of a class itself as an object.
An instance type or a type of a class as an object defines a complete type of the corresponding object taking into account its participation in metaclasses. A qualifier `instinst` is applicable to metaclasses and denotes instance instance type specified for this metaclass. In all cases independently of qualification by the component-kind a name of the type associated to a class means the corresponding mutable type (section 7) if the qualifier `nonobject` is not provided. If the latter qualifier is given, the type is treated subsequently as immutable.

Rules for composition of other sections of modules (function section, resource component specification section, resource component assertion section) are given in the further sections of the SYNTHESIS language definition.

### 6 Type system of the SYNTHESIS language

#### 6.1 Principles of the type system construction

The type system in the language constitutes a basis for the object model playing key role in specifying of the mediation environment.

Abstract data types (ADT) constitute the basis for the type system of the language providing for construction of arbitrary data types. ADT definition should include a specification defining a behavior of the type values (by means of the operation signatures and of their abstract descriptions applying a formal language). Such type interface encapsulates the type implementation. Description of an implementation also can be given, though in the SYNTHESIS language it is (generally) not obligatory. Definition of a mapping gives a correspondence of operations and parameters to specific functions and their parameters. Parametric specifications of ADT are possible.

A set of built-in data types is included into the language alongside with ADT. The built-in types are defined by their interfaces. Built-in and arbitrary types (given by ADT) are subdivided into concrete and generic (parametric) types. Generic types are treated as functions of free variables contained in type specifications that deliver types as their resulting values. An application of such type functions provides for concretization of generic types. Usage of generic types as parameters for the function specifications is allowed for static and for dynamic type concretization. For the latter case a technique for partial compilation of functions is assumed.

The multilevel type system of the language is organized as follows. A set of all values expressible in the language (including frames and objects that are not type objects themselves) is located on the zero level (the
level of values). At this level various computations on the values can be defined.

On the first level (the level of types) type objects are located (as well as type functions and expressions) that provides for the definition of the concrete and generic types and for the concretization of the latter ones. On the third level (the level of "types of types") the metatype objects are located that include as their instances the types of the first level. On the third level the metatypes objects are located that include the metatypes of the second level as their instances, and so on.

Thus the multilevel type system that sets a classification relationship on the data types is defined. For instance, it is natural to introduce type objects that denote arbitrary set types, sequence types, function types, etc. Metatypes are metaclasses (section 7) having types or metatypes as their instances.

Introducing of such hierarchy makes possible to specify generic programs with formal parameters typed with the metatypes. A concretization of such functions (static or dynamic) consists in a substitution of the formal parameter metatype with the respective concrete (or generic) types. To reach that for specifying of formal parameters of modules, types, classes as well as for specifying of type variables and for typing of variables the type designator is introduced:

\[
\ast\text{instance_of}(<\text{type name}>)
\]

It is assumed that <type name> denotes a metatype. The function instance_of obtains as its result a set of types (metatypes) being instances of this metatype. Usage of \(\ast\) leads to a calculation of a transitive closure of a function instance_of that includes types (metatypes).

Alongside with the type classification relationship, in the language facilities are introduced for establishing of the multiple "type-subtype" relationship. In the subtyping hierarchy a subtype inherits (or overloads) specifications of the attributes of a supertype. Any value of a type can be used everywhere where a value of the supertype can be used. Multiple subtyping hierarchy is allowed. The classification relationship that is being set on types is orthogonal to the subtyping relationship. It is essential that only types (metatypes) belonging to one and the same classification level can participate in the subtyping relationship.

### 6.1.1 A structure of the type system of the SYNTHESIS language

A structure of the type system of the SYNTHESIS language is shown on figure 6.1. The Taval type is used as the root of the lattice, the Tnone type – as its bottom. The Tnone type represents abstract values with an
"empty" semantics. For instance, an object none belongs to Tnone and can be returned as a result of functions creating an empty result of any type.

Links between types in the figure denote a subtyping relationship having the following meaning:

1. a value of a subtype can be used in all cases where a supertype value is expected;

2. to each attribute function of a supertype a function of a subtype should correspond such that a type of each its input parameter can coincide with or be a supertype of the type of respective parameter of the supertype function, a type of each its output parameter can coincide with or be a subtype of the type of respective parameter of the supertype function, a type of its input/output parameter should coincide with the type of respective parameter of the supertype function, its specification should refine the specification of the supertype function and conjunction of all subtype predicates (subtype invariant) should imply the conjunction of all predicates of the supertype (supertype invariant);

3. attribute functions of a subtype to which attribute functions of a supertype correspond can be renamed in case of multiple subtyping (subsection 6.4);

4. multiple subtyping is allowed (for a subtype a set of supertypes can be defined).

In the type system of the SYNTHESIS language mutable and immutable types are distinguished emphasizing the mutability of their values. Objects having unique identifiers possess a mutability property. Due to that they exist in the system as uniquely identified entities. On modification of the mutable values they preserve their entity, their identity. A modification of an immutable value leads to a creation of a new value, the old one ceases to exist. Unique identifiers themselves are immutable values that are kept by the system unmodified during all the life cycle of an object.

At the same time any mutable type is considered as a subtype of the respective immutable type. It means that abstract data types correspond to objects defining admissible collections of immutable values. An object type is a subtype of such ADT. Specialization of such subtype is reached by an introduction of the additional attribute self defined on the set of unique identifier values (all mutable types are subtypes of the Tsynth_object type.
Figure 6.1. A structure of the type system
in which an attribute self is defined). Therefore an object can be used in all cases where values of the respective ADT are used. All immutable types possess properties of the Taval type, all mutable values possess properties of the Tsynth_object type.

Values of Taval type are characterized by possessing an interface defining function signatures with typed parameters. The main property of an abstract value is an ability to run the interface functions requiring an abstract value itself for the first parameter. Values of Tframe type are abstract values possessing additional properties including a capability of introducing new or modifying existing properties given in their type. Values of Tsynth_object type specialize abstract values and frames with additional capabilities: to have unique identifiers and a potential ability to belong to a certain class (classes).

A class combines properties of a type and of a set. A class supports a set of objects of a given type that constitutes an extent of the class.

Sets in the language (alongside with collections, bags, sequences) are considered to be a general mechanism of grouping of abstract values. Sets resemble classes in a sense that they represent sets of ADT values or sets of objects. One and the same object can belong at the same time to several sets and to change dynamically its participation in them. A set is distinguished from a class in the following:

1. an object cannot be created by sets;

2. for a set there is no difference between the own and total extent as for a class (section 7).

At the same time in the SYNTHESIS language a class is considered as a subtype of a set type. Due to that these generally different constructs can be used quite uniformly: a class can be used everywhere where a set can be used. For instance, for the logic language formulae the starting and resulting data are represented as sets of ADT values (or of objects). Classes can be used for such sets as specializations of the set type. Due to that a calculus and algebra on which the SYNTHESIS formula language is based (section 9) are closed with respect to the composition of the algebraic operations.

To each class an ADT specification corresponds for its instances. For a class this ADT is assumed to be mutable even if in ADT specification this is not stated explicitly (subsection 6.4).

---

2The Taval, Tsynth_object, Type are types of the language implementation and are not specified by the language
6.2 Type specification

In a mediator or an information resource specification module types are specified in the type section.

\[
\text{< type section > ::= < empty >| type ::= < type specification list >|}
\]

\[
\text{< type specification list > ::= < type specification >|}
\text{< type specification list >, < type specification >|}
\]

\[
\text{< type specification > ::= < frame name >| < type declarator >|}
\text{\{ < type variable identifier >; < type declarator > \} |}
\text{\{ < type variable identifier >; < formal parameter identifier > \} |}
\text{\{ < type variable identifier >; \texttt{[*]}\text{instance_of( < type name > )} \}}
\]

\[
\text{< type declarator > ::= < built – in type >| < metatype specification >|}
\text{\{ < abstract type >|}
\text{\{ < parameterized type concretization >|}
\text{\texttt{< parameterized type concretization > ::= < frame name >|}}
\text{\{ < type identifier >;}
\text{\texttt{params ::= < actual parameter list >|}
\texttt{}}
\]

\[
\text{< metatype specification > ::= < metaclass specification >|}
\text{< built – in type > ::= < function description >| < simple built in type >|}
\text{\{ < collection type >| < set type >| < sequence type >|}
\text{< bag type >| < array type >|}
\text{< enumeration type >| < range type >|}
\text{< union type >| < product type >|}
\text{< frame type >|}
\text{< simple built in type > ::= integer | short | long | unsigned short |}
\text{unsigned_long | real | Boolean | char | octet | string | bitstring |}
\text{time | timeslice | interval}
\]

For a type specification the type declarators of the following kinds are allowed: built-in type declarators, ADT declarators, metatype declarators, parameterized type (generic types providing for their further concretization) declarators, declarators that are the concretizations of parameterized types. There is a possibility of creating of new types in the language by assignment of values of concretization of generic types to type variables. A type variable is introduced by a type declarator. Types are values of the type variable. In particular, types that are concretizations of a parameterized type or instances of a metatype can become values of a type variable.
A metatype specification is given similarly to the specification of a metaclass (subsections 7.2, 7.3). In the specification it is necessary to indicate in: metatype. A frame name in a type specification denotes a frame containing a type object.

Identifiers of the types being specified should be unique in a given module. During an import an identifier of a type can be qualified for uniqueness by an identifier of a module that contains the imported declarator.

Type designators are components of type expressions (subsection 6.2.1):

\[
\begin{align*}
&\text{< type designator > ::= < type name >| < attribute name >|} \\
&\quad\quad\quad [\ast]\text{instance_of( < type name >) | < reduct >|} \\
&\quad\quad\quad <\text{ simple built in type >} \\
&\text{< reduct > ::= < type name > " [ " reduct element list " ] " } \\
&\text{< reduct element list > ::= < reduct element > [, < reduct element >]...} \\
&\text{< reduct element > ::= < attribute identifier > [ / < type expression >] } \\
&\quad\quad\quad [::<\text{ path }> ] \\
&\text{< path > ::= < attribute identifier > [ / < type expression >] } \\
&\quad\quad\quad [<\text{ attribute identifier > [ / < type expression >]]...}
\end{align*}
\]

Usage of an attribute name for type designator means a reference to the attribute type.

Reducts (type projections) are allowed as type designators. Conventional case of a reduct is a type with a reduced set of attributes corresponding to the original type. In this case in the type hierarchy a reduct is an immediate or a transitive supertype of an original type. So reduct Person[name, degree] specifies a supertype of the type Person including only name and degree attributes of the type Person.

The reduct specification consists of a type designator for a source type and a set of reduct elements. In the general case the reduct specification defines a new type and a transformation from instances of the source ADT to instances of this reduct. According to the syntactic rule for < reduct > in the general case the reduct specification has the following form:

\[T[e_1, \ldots, e_n]\]

Here \(T\) is a type name for a source type and \(e_1, \ldots, e_n\) are reduct elements. Every reduct element specifies one attribute of the reduct. In the general case the reduct element consists of three parts:

- name;
• type;
• path.

According to the syntactic rule < reduct element > the reduct element specification has the following full form:

\[ a/T : t \]

Here \( a \) is identifier defining a name of the reduct element, \( T \) is a type designator defining a type of the reduct element, \( t \) is a path of the reduct element. The simplest case of path is a dot-delimited list of attribute identifiers having the following form:

\[ a_1.a_2.\ldots.a_k \]

Here \( a_1 \) is an attribute of the source type, \( a_2 \) is an attribute found in the type of an attribute \( a_1 \), etc. The path can be considered as a term and its type equals to a type of the last attribute \( a_k \). The value returned by the path is defined recursively: the value of \( a_1.a_2.\ldots.a_k \) equals to the value of the attribute \( a_k \) in the path \( a_1.a_2.\ldots.a_{k-1} \). General case of the path is discussed in subsection 9.4.6.

There are few simplest forms of the reduct element specification:

• \( a \) (a reduct element where the name and the type are not specified and the path consists of one attribute). The reduct element \( a \) is equivalent to \( a/T : a \) where \( T \) is a type of \( a \).

• \( a : t \) (a reduct element where the type is not specified). The reduct element \( a : t \) is equivalent to \( a/T : t \) where \( T \) is a type of the path \( t \).

Every reduct element defines an attribute as follows: the name of the attribute is a name of the reduct element and the type of the attribute is a type of the reduct element.

The path in a reduct element can contain an attribute of a set type. In the language a flattening takes place when an attribute in a reduct element path has been declared as a set (bag, sequence) type \{ set; type_of_element : T; \} and is typed in the path with type \( T \). So flattening is a type construction operation used in reducts: if an attribute identifier \( a \) in a path of a reduct element is a name of a set (bag, sequence) then its interpretation depends whether \( a \) is typed in the path or not. Without typing, \( a \) is interpreted as a respective set (bag, sequence). Otherwise, if \( a \) is typed
52 6 Type system of the SYNTHESIS language

with \( T \) - a type of element of the set (bag, sequence) \( a \) then \( a/T \) means flattening.

For example, in the following rule the resulting set for each repository will contain multiple instances, each one for a different author. So sets identified by attributes \( \text{collections} \) and \( \text{contains} \) are flattened.

\[
\text{repository} \\
(r/\text{Repository} \\
 \text{name, author}/\text{Creator}[\text{name, generalInfo}], \\
 \text{collections}/) \\
 \quad \{
 \quad \quad \{\text{set; type_of_element : Collection}[\text{name, location}, \\
 \quad \quad \quad \text{contains}/(\{\text{set; type_of_element : Painting; }\})];
 \quad \quad \}
 \quad ) : - \\
\text{museum} \\
m(name, \text{author}/\text{Creator}[\text{name, generalInfo}]: \\
 \text{collections}/\text{Collection.contains}/\text{Painting.created_by,} \\
 \text{collections}/ \\
 \quad \{
 \quad \quad \{\text{set; type_of_element : Collection}[\text{name, location}, \\
 \quad \quad \quad \text{contains}/(\{\text{set; type_of_element : Painting; }\})];
 \quad \quad \}
 \quad )
\]
\& m.name = 'Uffizi'

In the following rule the resulting set for each repository will contain one instance with a set of different authors of paintings contained in collections belonging to a repository: no flattening occurs.

\[
\text{repository} \\
(r/\text{Repository} \\
 \text{name,} \\
 \text{collections}/) \\
\]
6.2 Type specification

A type in the SYNTHESIS language is a value represented by an object. A type as the value is defined in the specifications, is determined in typed variables, is produced (inferred) by logic programs (formulae) and can be produced as the result of type expressions evaluation. Type functions can be formed in the language.

Main operations that are used for a construction of type expressions (explicit or implicit (constructed dynamically during logic program evaluation)) are operations of a type composition (meet and join) and a type product.

Informally, the meet operation $T_1 \& T_2$ of types $T_1$ and $T_2$ produces a type $T$ as an "intersection" of specifications of the operand types. The join operation $T_1 \mid T_2$ produces a type $T$ as a "join" of specifications of
the operand types. To define type composition operations formally the following concepts are required.

A signature $\Sigma_T$ of a type specification $T = (V_T, O_T, I_T)$ includes a set of operation symbols $O_T$ indicating operation arguments and result types and a set of predicate symbols $I_T$ for invariants. Conjunction of all invariants in $I_T$ constitutes the type invariant $\text{Inv}_T$. $O_T$ is a union of type state attributes $\text{Att}_T$ and type methods $\text{Meth}_T$. Extent $V_T$ of the type $T$ (carrier of the type) is modeled by a set of admissible instances of the type. Each instance of the type is a tuple of pairs $\langle a, v \rangle$ such that $a$ is a state attribute of the type ($a \in \text{Att}_T$) and $v$ is a value of the attribute. Every instance must satisfy the invariant $\text{Inv}_T$.

A type reduct $R_T$ is a subspecification of an abstract data type $T$ specification. A signature $\Sigma'_T$ of $R_T$ is a subsignature of $\Sigma_T$ including the extent $V_T$, a set of operation symbols $O'_T \subseteq O_T$, a set of symbols of invariants $I'_T \subseteq I_T$.

Most common reduct $R_{MC}(T_1, T_2)$ for types $T_1$ and $T_2$ is a reduct $R_{T_1}$ of $T_1$ such that there exists a reduct $R_{T_2}$ of $T_2$ such that $R_{T_2}$ refines $R_{T_1}$, and there can be no other reduct $R'_{T_1}$ such that $R_{MC}(T_1, T_2)$ is a reduct of $R'_{T_1}$, $R'_{T_1}$ is not equal to $R_{MC}(T_1, T_2)$ and there exists $R_{T_2}$ of $T_2$ that refines $R'_{T_1}$.

Type $C$ is a refinement of type $R$ iff:

- there exists an injective mapping $\text{Ops} : O_R \rightarrow O_C$;
- there exists a total abstraction function $\text{Abs} : V_C \rightarrow V_R$;
- for all $v \in V_C$, $\text{Inv}_C(v)$ implies $\text{Inv}_R(\text{Abs}(v))$;
- for all $o \in O_R$, $o$ is refined by $\text{Ops}(o)$.

To establish an operation refinement $o_1$ refines $o_2$ it is required that operation pre-condition $\text{pre}(o_2)$ implies pre-condition $\text{pre}(o_1)$ and post-condition $\text{post}(o_1)$ implies post-condition $\text{post}(o_2)$. Concerning operation signatures refinement means that signatures are equal up to parameter type relationship: types of input parameters of $o_1$ are supertypes of types of respective parameters of $o_2$ and types of output parameters of $o_1$ are subtypes of types of respective parameters of $o_2$.

Type operation meet. Generally type $T$ – a result of an operation $T_1 \& T_2$ – includes attributes that are common for $T_1$ and $T_2$. Common elements of
the types are defined by most common reducts \( R_{MC}(T_1, T_2) \) and \( R_{MC}(T_2, T_1) \). More formally \( O_{T_1 \& T_2} \) is defined as

\[
O_{T_1 \& T_2} = O_{R_{MC}(T_1, T_2)} \cup O_{R_{MC}(T_2, T_1)}
\]

Type invariant of \( T \) is defined as a disjunction of operand types invariants \( Inv_{T_1} \cup Inv_{T_2} \).

Note that if \( T_2 \ (T_1) \) is a subtype of \( T_1 \ (T_2) \) then \( T_1 \ (T_2) \) is a result of a meet. Type \( T \) is placed in the type lattice as a supertype of types being operands of the operation.

Type operation join. Generally type \( T \) – a result of an operation \( T_1 \mid T_2 \) – includes a merge of specifications of \( T_1 \) and \( T_2 \). Common elements of specifications of \( T_1 \) and \( T_2 \) are included into the merge (resulting type) only once. Common elements of the types are defined by most common reducts \( R_{MC}(T_1, T_2) \) and \( R_{MC}(T_2, T_1) \). More formally \( O_{T_1 \mid T_2} \) is defined as

\[
O_{T_1 \mid T_2} = (O_{T_1} \cap O_{R_{MC}(T_1, T_2)}) \cup (O_{T_2} \cap O_{R_{MC}(T_2, T_1)})
\]

Type invariant of \( T \) is defined as a conjunction of operand types invariants \( Inv_{T_1} \land Inv_{T_2} \).

Note that if \( T_2 \ (T_1) \) is a subtype of \( T_1 \ (T_2) \) then \( T_2 \ (T_1) \) is a result of a join operation. A type \( T \) is placed in the type lattice as a subtype of the operand types.

Type operation product. An operation \( T_1 \ast T_2 \) produces as a result a type \( T \) that is a product of the operand types. A type \( T \) is placed in the type lattice as a subtype of other product types (but not of the operand types) in accordance with the interfaces defined for the product types. Values of a product type are pairs or generally \( n \)–tuples in which the first components is a value of \( T_1 \) type, the second component is a value of \( T_2 \) type, the \( n \)–th component is a value of \( T_n \) type.

An interface for a product type \( T = T_1 \ast T_2 \ast \ldots \ast T_n \) is constituted of the injecting functions \( f_i \ (1 \leq i \leq n) \) such that for an arbitrary value of a type \( T \), e.g., \( o \), a function designator \( f_i(o) \) or \( o.f_i \) returns a value of a type \( T_i \) representing an \( i \)–th component of \( o \).

A product type \( T_1 \ast T_2 \ast \ldots \ast T_n \) is placed in the type lattice as a subtype of a product type \( T'_1 \ast T'_2 \ast \ldots \ast T'_m \) if \( m \leq n \) and \( T_i \) is a subtype of \( T'_i \) for \( 1 \leq i \leq m \). It is placed in the type lattice as a supertype of \( T''_1 \ast T''_2 \ast \ldots \ast T''_k \) type if \( n \leq k \) and \( T_i \) is a supertype of \( T''_i \) for \( 1 \leq i \leq n \). If a product type cannot be placed into a type lattice as a subtype of another product type then it is placed there as a
subtype of the `Taval` type (if it is an immutable type) or as a subtype of the `Tsynth_object` type if it is a mutable type.

An equality of values of a product type (including cases when such values are treated as mutable) is determined by the pairwise check of equality of the respective components of two values of a product type (the number of components in the values should be the same).

Taking into account the above, the type expressions are constructed in accordance with the following rules:

```
< type expression > ::= < compositional term > |
< type expression > < compositional operation > < type term >
< compositional operation > ::= " | " | & | *
< type term > ::= < type variable identifier > | < frame − type > |
< type designator > | ( < type expression > ) |
< function designator >
```

Parenthesis are allowed in the type expressions. Enclosing of a subexpression in parenthesis means a creation of a new type. With respect to the eventual type that is computed by the type expression, operations of meet and join are commutative and associative. A product operation is neither commutative nor associative. A product operation has a higher priority comparing to the operations of type meet and join.

Using of parenthesis influences the result of a product in the following way. Assume that three type expressions are defined:

```
(T_1 ∗ T_2) ∗ T_3
T_1 ∗ (T_2 ∗ T_3)
T_1 ∗ T_2 ∗ T_3
```

A resulting type of the first expression includes two injecting functions: \( f_1 \) giving a result of a product type \( T_1 ∗ T_2 \) and \( f_2 \) giving a result of a \( T_3 \) type. A type of the second expression includes two other injecting functions: \( f_1 \) giving a result of \( T_1 \) type and \( f_2 \) giving a result of \( T_2 ∗ T_3 \) product type. A type of the third expression includes three injecting functions: \( f_1 \) giving a result of a \( T_1 \) type, \( f_2 \) giving a result of \( T_2 \) type and \( f_3 \) giving a result of \( T_3 \) type.

### 6.3 Function type

All operations over typed data in the SYNTHESIS language are represented by functions. A function type defines a set of functions, each function of the set being a mapping of the function’s domain (a carte-
sian product of the sets of the input parameters values) into a range of the function (a cartesian product of the sets of the output parameters values).

Functions can be passed as parameters, can be returned as values, can be used as values of abstract value attributes. This property of functions is valid also for the functions defined in the SYNTHESIS language by means of logic programs.

```
< function type > ::= in :< function kind > [, < metaclass name list >];
  [params : {< formal parameter list >}; ]
  [raises : {< list of exception state names >}; ]
  [selector : {name :< selector name >;
      type :< selector type >;
      priority :< priority >}; ]
  [< function specification >; ]
< function kind > ::= function | predicate
< formal parameter identifier > ::=< parameter kind symbol >< typed variable >
< parameter kind symbol > ::= − | + < empty >
```

The parameter kind symbol has the following meaning: '+' – input parameter; '-' – output parameter; '< empty > – input/output parameter.

A parameter kind symbol can be omitted in specifications of predicates where an interpretation of a parameter kind can be dynamic. Such functions are marked as predicate.

Functions returning results of expressions should include in their type definitions an output parameter given by a typed variable:

− returns/ < type designator >

The predicate function type assumes by default the definition of the output parameter -returns/boolean.

Typed variable definition is given in subsection 9.2.

```
< list of exception state names > ::=< exception state name >|
  < exception state name >, < list of exception state names >
```

A list of exception state names in a function type definition indicates what exception states will be fixed at the function call (the type exception is defined in the Appendix B). There exists a set of system built-in exception states that are considered to be included into a list by default. If the list in a function type definition is not given it is assumed that it exists
and includes the default states. The definition of the set of system built-in exception states is not a part of the SYNTHESIS language specification.

\[< \text{function specification} >::= \{ \text{predicative} : \{ < \text{formula} > \} \} \mid \{ < \text{rule} > \} \]

A predicative specification is given by mixed pre- and post-conditions defining the function. To express conditions related to changing of the information resource states an ability is needed to denote variables expressing a resource state before and after a function (transaction) execution. For that a reference to a value of a variable after execution of a function is denoted by the variable identifier marked by a subsequent apostroph. A variable identifier without apostroph refers to its value before execution of a function.

\[\text{in} : < \text{function kind} >, < \text{metatype name list} > \]
is used in case when an ADT attribute is a functional attribute and this attribute belongs to the attribute categories given by the association metaclasses (subsections 7.2, 7.4).

\[< \text{selector name} >::=< \text{string of symbols} >|< \text{parameter identifier} >\]

\[< \text{selector type} >::= \text{infix} | \text{prefix} | \text{suffix} |< \text{parameter identifier} >\]

\[< \text{priority} >::=< \text{integer} >|< \text{parameter identifier} >\]

Selectors define alternative names for functions. These names can be used for construction of operator expressions. Selectors provide for construction of operator expressions in natural form. Such expressions can be used, e.g., as constituents of formulae (section 9). The following types of operators are allowed:

- infix that is placed between its two arguments;
- prefix that is placed before its single argument;
- suffix that is placed after its single argument.

A priority given by an integer is assigned to each operator. To control operator’s priority in an expression, parenthesis are allowed.

A function description is a built-in type making possible to specify a function type (including generic function types). A function description that does not contain a definition of a program for the function defines a function type (a function signature and possibly its formal specification). Such definition creates a possibility of introducing of type variables typed by the signatures of such formally specified functions.
6.3 Function type

< function description >::= < frame name > | 
   {[< frame name >]; | < function type >; 
   [ < implementation >]}

If a function type contains in a function description formal parameters for a concretization of a function and these parameters are typed by a metatype or a parameterized type then the parameters mentioned can be substituted by respective types. Concretization expression for functions is the same as for any other parameterized type. In a concretization expression the places of formal parameters that are not subjected to concretization should be filled with anonymous variables and places of formal parameters that are subjected to concretization should be filled with actual parameters (type values or type expressions). It is important that during a concretization new identifier (an identifier of a type variable) can be given for the type identifier in a type specification.

Function declarators are collected in the function section of a module.

< function section >::= < empty > | 
   function :< function declarator list > 
< function declarator list > ::= < function declarator > | 
   < function declarator list >, < function declarator > 
< function declarator >::= < frame name > | 
   { [< function header >]; < implementation > } 
< function header >::= < identifier >; < function type >

Providing of a function declarator is equivalent to the function definition:

< identifier >:< function description >

Here for the < identifier > an identifier of a typed variable, a slot name, an identifier of an ADT attribute or an identifier of a type associated with a class can be used.

There are the following options for identifiers used as function names in function designators, predicates and functions in formulae (section 9). If a function is defined by description, an identifier of type variable or a slot name or an attribute name can be used as a function name. If a function is defined by declarator, a function identifier taken from the declarator header is used as a function name.

A function header can be omitted in case when a function type can be inferred from its implementation and the function identifier is not required. The latter is allowed, e.g., in case when a function declarator is used as an assertion (subsection 8.1).
A direct implementation of a function can be given by a logic program (subsection 9.12). Indirectly a function is defined by a reference to an implementation specification that is not fixed by the SYNTHESIS language.

< implementation > ::= < logic program > |
< reference to implementation specification >
< logic program > ::= { [< variable list >; ]< program > }
< variable list > ::= < variable declarator > |
< variable declarator > ::= < variable identifier > : < type designator >

Facts and rules in a logic program are given as follows:

< fact > ::= < atom >
< rule > ::= < head > : − < body > .
< head > ::= < atom >
< body > ::= < formula >

Concepts of an atom and of a formula are defined in (section 9). A program can consist of a single formula representing a body of a single rule. A head of such rule is defined by a function type signature. If a function implementation is given by a logic program then a function type can be defined implicitly if it can be inferred from a logic program. In this case a function type includes the only phrase in : predicate.

Function themselves are considered to be abstract values that can be manipulated as other abstract values. Specific operations on the interface of this type are operations of function compilation, function execution, determination that one function is a refinement of another one as well as specific implementation depending operations (e.g., derivation of a logic program tree as a result of a logic program compilation, getting of an optimizing program tree, etc.).

6.4 Abstract data type

In the SYNTHESIS language an abstract data type (ADT) is used as a constructor of an arbitrary user-defined type.

< abstract type > ::= [< type identifier >;
in : type], < metatype name list >];
[params : { < formal parameter list > }; ]
[class_params : { < class parameters > } ]; ]
6.4 Abstract data type

```plaintext
<supertype :< supertype list> ; [...] =
<rename : {< renaming list >}; ]>
<attribute specification list >}

<class parameters > ::= < identifier > ; < ADT identifier > |
< class parameters > ; < identifier > ; < ADT identifier >
<supertype list > ::= {< type identifier >; |
<class_params : {< class parameters >} ; |
{< type identifier >; |
<class_params : {< class parameters >} ; < supertype list > |
< type identifier > ; < supertype list >
```

Usually ADT defines a mutable type. To define an immutable type
the definition

```plaintext
in : type, nonobject[, < metatype name list >];
```

is required.

A collection of types forms an acyclic graph with the pre-defined type
Taval in the root on the basis of a subtyping relationship defined in ab-
stract types by the supertype attribute . In the type hierarchy a set of
supertypes for a subtype is allowed (providing for multiple subtyping).
The rules for inheritance of supertype properties by a subtype follows.

In case of name conflicts during inheritance an attribute renaming is
allowed. Renaming is specified by a renaming list rename. An element
of renaming list is defined as <property > ; < property > where a new at-
ttribute name is given after colon. Attribute names in the list are given
by attribute identifiers. In case of functional attributes a selector name is allowed.

```plaintext
<renaming list > ::= < renaming list element > |
<renaming list element > ; < renaming list >
<renaming list element > ::= < property > ; [ < property > ]
<property > ::= [ < supertype identifier > . ] < attribute identifier > |
<selector name >'
```

Selectors are defined in subsection 6.3. Attributes of a supertype can
be redefined. Redefinition of an attribute means a modification of its
type specification. Generally attributes of a type are defined as functions.
Redefinition of a function can be done in accordance with the subtyping
definition given in subsection 6.1. In the language there is no direct indi-
cation what should be an implementation of the inherited specifications.

Any abstract value exists in time. Preference is given to a uniform
model when to all attributes of an abstract value corresponding to an
ADT two temporal attributes are related: the attributes of existence and
belief (the same attributes that are related to each frame (subsection 4.4)). Alongside with such model a nonuniform temporal assignment can also be used when to each attribute its own temporal attribute is related (subsection 6.4.1).

By in an abstract data type can be declared belonging to different metatypes. type means an identifier of the built-in metatype. Parameterized (generic) ADTs are allowed. Type variable identifiers, SYNTHESIS language type values, or any other values and constants can substitute the parameters for concretization of the type. ADT concretization is provided analogously to other parameterized types.

The level of abstractness of ADT specification can be different. The most abstract is an ADT specification that defines the function signatures or the function signatures and predicative specifications of functions. The most concrete is an ADT specification that refers to the implementation of functions. In case of an abstract specification the implementation of a type can be defined dynamically. It means, in particular, that different coexisting values of the same type can have different implementations.

### 6.4.1 Attribute specification in abstract data types

Syntactically the specification of ADT attributes is defined as follows:

\[
<\text{attribute specification list}> ::= <\text{attribute specification}> | \\
<\text{attribute specification}> <\text{attribute specification list}>
\]

\[
<\text{attribute specification}> ::= \\
<\text{attribute identifier}> ::=<\text{attribute type}>; \\
\text{metaslot}[\text{card numb} ::= \text{unsigned integer}]; \\
\text{init} ::= \text{constant}; \\
\text{in} ::= <\text{attribute category name list}>; \\
<\text{assertion list}>; \\
\text{inverse} ::= <\text{type identifier}> . <\text{attribute identifier}>; \\
\text{history} : \{ \text{valid} ::= <\text{temporal slice type}>; \\
\text{belief} ::= <\text{temporal type slice type}> \}\end \\
<\text{attribute identifier}> ::= \{ <\text{assertion list}> \}; \\
<\text{attribute identifier}> ::= <\text{identifier}> \\
<\text{association metaclass identifier}> \\
<\text{attribute type}> ::= <\text{type}> | \{ <\text{attribute kind}> ::= <\text{type} > \} \\
<\text{attribute kind}> ::= \text{own} | \text{private} \\
<\text{type}> ::= <\text{type name}> | <\text{built-in type}> \\
<\text{parameter identifier}> | <\text{reduct}> \\
<\text{assertion list element}> ::= <\text{assertion name}>:
\]

\[\]

\[
<\text{type}> ::= <\text{type name}> | <\text{built-in type}> \\
<\text{parameter identifier}> | <\text{reduct}>
\]

\[
<\text{assertion list element}> ::= <\text{assertion name}>:
\]

\[\]
ADT attributes are treated as functions that can be defined explicitly or implicitly (with the stored attributes implicitly defined functions get and set are associated). On call of functions corresponding to attributes an ADT value (or an object) is assumed to be given for the first parameter. Analogously attributes of (partially) self-defined objects as well as frame slots are treated as functions. The domain of such functions consists of a single object (frame).

A built-in or an abstract data type, a parameter (to be substituted by a type or a class), a reference to a type of another attribute as well as an ordered indexed (array) or unordered (set) collection of abstract values can be given for an attribute type. Indication of a type in an attribute specification means that the values of the given type or of any of its subtypes can be used for components of ADT values. A recursive specification of an attribute is allowed when a name of an ADT being defined is used to indicate a type of an attribute of the ADT.

Objects included into an ADT value as mutable values of its attributes can be treated differently as completely autonomous objects possessing all object rights or as constituent parts of an ADT value. To define that, three different kinds of an attribute type are allowed: plain object, own object or private object.

If an attribute is specified as own object then the respective component object is an owner of a value of a type being defined. It means that on deletion of a value of this type the component object will be also deleted. As far as other ADT values can refer to the same component object their respective attributes will get undefined values none on deletion. Stored attributes having immutable types are treated as own components with respect to a deletion.

If an attribute is specified (after ADT binding to classes) as:

\[ \text{< attribute identifier >: \text{< class identifier >}} \]

then it means that the component is a completely independent object and a deletion of a value of this ADT does not lead to a deletion of the component object. The meaning of an attribute specified as private is the same as for the own attribute with the addition that other abstract values cannot refer to the private object as their component.

With each attribute definition a metaslot providing additional attribute characteristics can be associated.

\[ \text{3It is important to note that each specification in the language is a frame. Therefore it is allowed to associate metaframes, metaslots and metavalues with any respective elements of a specification. In particular, this feature is important for construction of the canonical model kernel extensions.}\]
Type system of the SYNTHESIS language

in :< metatag name list > is used in case when an attribute belongs to attribute categories defined by association metaclasses (subsection 7.4). A name of an association metaclass can be used as the category name.

Slot card_num defines the maximal admissible number of different values of a given type. init can be used to specify initial attribute value of a given type. history slot defines that for the attribute its own history should be maintained independently of the history of the whole ADT value. In expressions and formulae the constructions

< identifier > .belief
< identifier > .valid

define a temporal assignment of ADT values as a whole (including their attributes, assertions and other components).

By means of an attribute category name (built-in or arbitrarily defined) that is used in an assertion list (section 8) the roles defined by the given category can be determined for the given assertion list. In case when several roles should be assigned to an attribute, a list of attribute category names is used. Thus with an ADT value assertions having different roles can be associated, including, e.g.:

- a role of value-generating rule of given type;
- a role of a logical condition that should be satisfied by the values at any moments of their life cycle;
- a role of an action specifier.

By a slot in its metaslot any attribute can be associated to various categories. By means of the inverse slot it is possible to declare that a given attribute is an inverse function with respect to an attribute in another type (class). An example of such conjugate definition is given in subsection 7.4.

In the SYNTHESIS language it is not allowed to use class identifiers in the attribute specifications (this restriction is not applicable however to type specifications that are given directly in class specification (subsection 7.2)). The cause for this restriction is that type specifications in a type section are considered as having no explicit relationships with specifications of particular classes. As far as an ADT specification can be related at the same time to specifications of different classes, the ranges in specifications of type attributes corresponding to classes can be different for different class specifications referring to one and the same type. To cope with that special type parameterization (attribute class_params
of ADT specification) is introduced that provides for concretization of a type specification (and of all its transitive supertypes) to be reused in a specific class. Such concretization is based on mapping of types into classes in the original type specification. In a type specification a type of a class parameter should be given. It means that for a concretization only those actual class parameters can be used for which an instance type coincides with a formal parameter type or is its subtype.

6.5 Built-in types
The majority of the built-in types can be declared as the parameterized ADTs. Any element of a type specification can be declared as a parameter. Concretization of the built-in types is realized according to the general rules (subsection 6.2).

Descriptions of the built-in types are given in Appendix B.

6.6 Built-in metatypes
Built-in metatypes of the language are the following: metatype, nonobject, type, function, predicate, integer, real, string, bitstring, enum, range, set, union, sequence, product, array, time, interval, timeslice. This list can be extended.

7 Class representation facilities
7.1 Collections of objects modeling
Homogeneous sets of application domain entities are represented in the SYNTHESIS language as classes. On the other hand, services are represented by objects that may not belong to class extents.

A class together with the definitions of types related to the class defines an interface of objects of the respective resource (a mediator or an information resource registered at the mediator). With a class an extent is associated that is formed by a set of these objects. Generally by the SYNTHESIS language an abstract specification of a class can be given. An implementation of a class usually is completely separated of the specification. An implementation should be correct refinement of a specification. A broad range of possibilities is allowed: a single implementation is defined for all objects of a class; each object can have its own implementation; several different implementations support subsets of a class extent.
Explicitly distinguishing between type and class specifications the SYNTHESIS language emphasizes the role of a type as a specification of a structure and behavior of objects related to a class and the role of a class as a representative of a set of objects.

Main relationships associated with classes as sets of objects are the following:

- generalization/specialization relationship providing for the definition of an application domain structure as a hierarchy of interrelated classes. Usually this relationship is used to model differently the same real world entities by the interrelated classes. The generalization relationship can be based on one of the two different principles of establishing relationships between superclass and its subclass. According to the first, category principle, subclass objects are considered to be a specialized category of the supertype objects. The same real world objects are modelled by a subclass establishing for them an additional information with respect to the superclass objects. For the category principle an instance type of a subclass is a subtype of an instance type of superclass, total extent of subclass is a subset of the superclass extent. In another words, a set of real world entities corresponding to a subclass is a subset of a set of real world entities corresponding to a superclass.

The second, role relationship principle relates a specialization class with more general class. An object that is a role of more general object communicates for it an additional information. For the role principle an instance type of a subclass is not a subtype of an instance type of a superclass.

- class - class instance relationship;

- class - metaclass classification relationship (subsection 7.3) that creates a classification hierarchy orthogonal to the generalization hierarchy of classes.

For the category principle used to form the generalization relationship, a subclass object in the class hierarchy can be used everywhere where an object of its superclass can be used (as far as the subclass instance type is a subtype of a superclass instance type). Physically subclass and superclass extents are used to contain disjoint sets of objects having different object identifiers. Such extents are called further own class extents. An example of an exceptional case is when objects are created belonging simultaneously to a class and a subclass (then an object with one and the
same object identifier becomes an instance of a class and of a subclass). However, with respect to the real world entities own extents of a superclass and of a subclass can be in different relationships: to be a mappings of one and the same set of real world entities; a set of real world entities to which a subclass extent corresponds can be included into a set of real world entities represented by a superclass extent or can have non-empty intersection with it. Actual relationship is defined by a discipline of insertion and removing of instances of a class and its subclass and is not restricted by the language. The language contains sufficient facilities to specify and support the required relationship. An example of one of the possible approaches for such discipline setting is presented in subsection 7.2.

For generalization according to the category principle a subclass extent is logically included into a superclass extent (such inclusion is valid for all its transitive subclasses). Analogously when an attribute of an object is defined on a class as its range it can take objects of any of its transitive subclasses as its values. A class extent that includes extents of all its transitive subclasses is called a total extent.

Facilities used for representation of classes and of autonomous (self-defined) objects provide for specification of mediation environment providing for refining mapping of various information resource specification into the mediator definition.

Objects can be completely typed (in this case all their attributes are defined by a class instance type specification), partially typed (part of their attributes are defined by a class instance type specification and other attributes can be arbitrarily defined for different instances of the class by the objects themselves) or can be completely autonomous (not associated to any class). Autonomous objects exist isolated or can be related to some worlds and are self-defined.

It is essential that an object can be an instance of several classes so that in each of them a structure and behavior of an object correspond to the class or are a specialization of a structure and behavior specified by a class.

An object belonging to a class in object model of the SYNTHESIS language can be treated as a mutable frame and a class - as a set of such frames. At the extreme object database can be treated as a classified set of frames with object identities. At the same time autonomous objects are allowed that may not belong to any classes. Generally, the behaviors of objects belonging to classes and autonomous objects are similar. Autonomous object at any moment of its life cycle can be defined as belonging to a class using suitable instance type definition.
Kinds of associations that can be used by frames in a world can be restricted. To do that it is required to declare in a class specification the types of binary associations (of the frame language) by which frames can be interrelated. Such class corresponds to a world in a frame base. By means of such associations arbitrary relationships of objects and frames can be established to define flexibly additional information about objects.

Class specifications combined with frame language facilities give flexible abilities for the representation of hybrid structured and semi-structured information about an application domain. Such kinds of information can be kept separately or can be related on the basis of object-frame associations.

With respect to dispatching of object method calls a classic method is assumed for the SYNTHESIS language.

### 7.2 Mediator or information resource specification section

The mediator or information resource descriptions are provided in the language by means of the resource component specifications given by singleton (interface) specifications and by class specifications. Thus resource specifications provide for a description of resource components represented by single objects as well as by collections of objects (classes). General rules for classes and singleton definitions are given in this subsection. Rules for definition of transaction and script classes that also are considered as resource components are given in separate subsections.

```
< resource component specification section >::=< empty >|  
  class_specification :< class specification list >; |  
  singleton_specification :< singleton specification list >;  
< singleton specification list >::=< singleton specification >|  
  < singleton specification >, < singleton specification list >  
< singleton specification >::=  
  < frame name >|< singleton declarator >  
< singleton declarator >::=  
  {< identifier >;  
    [interface : {< type identifier >;  
      [params :< class parameters of type >]}];  
    [< attribute specification list >]}  
< class specification list >::=< class specification >|  
  < class specification >, < class specification list >  
< class specification >::=  
  < frame name >|< class declarator >|  
```
A resource component specified as a singleton is defined by its identifier and an interface that can be defined by a type identifier (with class parameter binding if required) and (or) by an attribute specification list.

A resource component modeled as a collection of objects is defined by a class specification. By means of in a class can be declared belonging to different metaclasses (subsection 7.3). By default when in is not given it is assumed that a class belongs to the built-in metaclass class.

\[ \text{< built-in metaclass > ::= metaclass | class | association | transaction | script | world | activity} \]
Attribute types, types of formal parameters of functions as well as other elements of class specifications can be passed by parameters into the parameterized classes for their concretization.

A type variable is introduced by a parameterized class specification, by a specification (name) of a metaclass or by a set of classes in a given metaclass (direct or transitive). Values of type variables are classes being concretizations of a parameterized class or metaclass. Classes (metaclasses) that are instances of a metaclass are also considered to be valid values of a type variable introduced by metaclass.

A class specification combines information about two kinds of objects: about a class as an object itself and about objects - instances of the class. Objects that are instances of some classes are associated with classes to which they belong. Classes as objects themselves can have specific properties that are defined as attribute specifications of a type corresponding to a class as an object that is defined in a class section of the given class specification or defined in instance section or in an instance-instance section of a metaclass to which the class belongs as its direct or transitive instance.

Specification of a type of objects - instances of a class is given in an instance section of a class. Using a metaclass, type specifications for classes that are instances of the metaclass should be declared in an instance section of the metaclass; type specifications for objects that are instances of classes being instances of the metaclass should be given in the instance - instance section of the metaclass.

A type specifications associated with classes are formed according to the following rules. A type of a class instance object includes:

1. attributes of an object type defined in instance section of a class to which the object belongs,
2. attributes of an object type defined in instance - instance section of each metaclass of a class to which the object belongs,
3. besides that, each object (as a self-defined entity) can include its own attributes.

Type specification of a type of a class as an object itself or of an instance type of a class or of an instance - instance type of a metaclass can be provided in two different ways. The first way is realized by means of a reference to an ADT specification in a type section and provided with class parameters that are necessary for class binding to a type specification.

The second way consists in provision of a type specification directly inside of a class specification. In this case an ADT specification, its
7.2 Mediator or information resource specification section

Metatypes, supertype list (with class parameters for class binding of supertypes specifications if required) should be defined. The latter way is preferable if a type specification is used for a specification of the only one class. For the second way a simplified specification is allowed: a type identifier (as well as slots in, supertype) can be omitted. Such specification is called anonymous.

A set type corresponds to a class extent with an element type being an instance type of the class. The name of such set type and the name of a set corresponding to a class extent coincides with the class name.

A simple example of a class specification that includes a type specification follows:

```plaintext
{project; in : class;
 instance_section : 
 {Project; in : type;
 name : string;
 gets_grants_from : agencies; 
 }
}

{companies; in : class;
 instance_section :
 {Company; in : type;
 name : string;
 engaged_in : {set; type_of_element : project.inst; ];
 }
}
```

Classes can be interrelated by a generalization/specialization relationship based on the category or the role principle (subsection 7.1) that is defined by a slot superclass of a class specification. The role specialization is given by a slot role_of of a class specification. Class hierarchies in which the category and the role principle could be used interchangeably are not allowed.

In different classes different object instances can correspond to the same real world entities. Such correspondences can be explicitly defined. A general approach consists in introducing of an object identity predicate in the following way. For a pair of types of objects - instances of the mentioned classes, the predicate same based on the application domain semantics should express the object identity property by their correspondence to the same real world entities. The identity predicate should be redefined for the respective classes so that objects are considered to be identical if their object identifiers are equal or if they satisfy the predicate.
Other options are also possible. For instance, to establish a relationship of subclass and superclass objects (with respect to their correspondence to the same real world entities) a relationship subclass - superclass (that is defined by a slot superclass: &lt;superclass list&gt;) can obtain any semantic interpretation in the following way. With a slot superclass as with an attribute a metaslot can be associated that defines the attribute as a function that to each object of a subclass puts into correspondence a unique object of a superclass. An example of specifying of such relationship follows:

Let $c$ be a subclass and $s$ be a superclass. Then $c$ can be defined as:

```plaintext
{c;
in: class;
superclass: s;
metaslot
in: one_to_one;
r: invariant, {{all kc (in(kc, c) -> in(kc, s))}}
end
...
}
```

Here `one_to_one` is an association metaclass (subsection 7.4) that defines `association_type: {{1,1},{0,1}}` making $kc$ to be a unique key of classes $c$ and $s$. $kc$ key can have a product type or be typed by a reduct of an ADT. Its values can be created by means of collection comprehension formulae (section 9) applied to classes $c$ and $s$. Thus the assertion associated to a superclass slot states that to each object of $c$ the only object of $s$ corresponds; both objects having equal $kc$ values. Such definition allows to model situations in which objects of $s$ and $c$ having equal $kc$ values correspond to one and the same real world entity. This definition can induce additional actions on creation and deletion of objects of $s$ and $c$ classes that can be defined for instance as follows.

Creation of an object of $c$ class. On creation of an object it is assumed that all its attributes participating in the definition of $kc$ are defined. It is necessary to check whether exists an object in a superclass $s$ with an equal value of $kc$. If such object does not exist, an object of a superclass $s$ is created taking into account differences of instance types of classes $s$ and $c$. This action is applied to all transitive superclasses of $c$ having the same semantics of superclass - subclass relationship.

Deletion of an object of $s$. On deletion of such object the objects having the same $kc$ value should be deleted from the transitive subclasses having the same semantics of superclass - subclass relationship with $s$. 
A subclass-superclass association can acquire different semantics. For example, if in association metaclass to which an attribute `superclass` belongs, an association type is defined as `association_type: \( \{0,1\}, \{0,1\} \)` then the partial function semantics is specified that induces another actions on creation and deletion of objects.

\[
<\text{association section}> ::= <\text{association}> | \\
<\text{association}>, <\text{association section}>
\]

An association section can be included into a class specification in case when a class is in `world` (defining a world with the pre-defined set of admissible associations). In this case no instance attributes are declared. In the association section only those binary associations can be declared that can be used in a world of frames corresponding to this class. Associations are defined in accordance with the frame language.

A subclass/superclass relationship can be established between classes belonging to the `world` metaclass. Such relationship means that all association specifications declared for a superclass are inherited by a subclass extending its definitions in a subclass association section.

Classes as views are defined in the language by means of functions (section 9). In specifications of a view a function computing the class is defined in a metaslot associated with an attribute `in: class`. A view specification can declare creation of a virtual class that exists only during formula evaluation or of a materialized class.

Views preserving objects on which they are based and views that do not preserve those objects are distinguished. Object preserving views can be declared as subclasses (superclasses) of classes from which they were derived (base classes).

Object creation and deletion in a view (in cases when a respective relationship of a view class and base class (a subclass or a superclass) has been established) can result in creation (deletion) of respective object in a base class.

### 7.3 Metaclasses

A classification "dimension" that is orthogonal to a generalization / specialization dimension is formed as follows. As far as classes are objects themselves they can become instances of other more general or generic classes (metaclasses). Thus a multilevel classification system can be formed with the following levels:

- the zero level includes all objects and all frames that are not classes
and do not belong to any specific class;

• the first level includes all classes having zero level objects as their instances;

• the second level includes all metaclasses having classes of the first level as their instances;

• the third level includes all metaclasses having metaclasses of the second level as their instances, and so on.

The mixed level is also allowed having objects, classes and metaclasses of any levels as its instances. The set of built-in classes of this level includes:

• the class \texttt{frame} that is the universal class to which all frames and objects belong as its instances;

• the metaclass \texttt{class} containing all classes;

• the metaclass \texttt{metaclass} containing all metaclasses.

The stratification defined above should be taken into account when a generalization relationship is established. Only the classes belonging to one and the same classification level can participate in a generalization/specialization relationship. In a specification of a metaclass of any level \texttt{metaclass} should be defined. An inclusion of a class to metaclasses is defined by a list of metaclasses \texttt{in}. An absence of the list means that this class does not belong to any metaclass (this is allowed only for classes that belong to the metaclass \texttt{class} by default). Single indication of \texttt{metaclass} in the list means that a class is a metaclass that is not included into any metaclasses of higher level.

A specific feature of metaclasses (comparing to classes) is an ability to declare attributes specified in a metaclass as association metaclasses (subsection 7.4). Thus a metaclass can introduce attribute categories that can be used in classes being instances of this metaclass and in their instances. To declare an attribute specified in a metaclass as an association metaclass it is necessary to indicate in: \texttt{association} in its metaslot. In classes being instances of a given metaclass in metaslots of the respective attributes to indicate belonging of an attribute to an attribute category specified in the metaclass the construct in :< \texttt{attribute category} – \texttt{metaclass attribute identifier} > is to be used. In specifications of such attribute categories, the names of types and classes (referring in particular to metatypes and metaclasses including those that are subjected to further
concretization) are allowed for the names of domains (ranges) of associations. Besides that such metaclass attributes can be defined belonging to arbitrary association metaclasses. Thus generic form of a metaclass and general properties of attributes of its classes (and/or their instances) can be defined.

Generally metaclasses provide for introducing of generic concepts and of common attributes (or of their categories) for similar classes, for introducing of common consistency constraints and deductive rules for such classes and their attributes. Metaclasses provide for proper grouping of an application domain information and for proper differentiation of various application domains.

### 7.4 Association metaclasses

Object attributes are treated in their turn also as classes of association objects establishing a correspondence between a set of objects in an association domain and a set of objects (values) in an association range. Thus a specification of an attribute is considered as a specification of an association class. A unique identifier for a specific association (an instance of an association class) is a pair containing a unique identifier of an object including an attribute and a name of the attribute.

Obvious attributes of associations are their domain and range. Other properties of an association class and/or of its instances can be specified in metaslots of an attribute. Slots `card numb`, `in`, `inverse`, `init` define attributes of an association class, a slot `history` defines instance attributes. An attribute `superclass` is also allowed as an attribute of an association class denoting association classes that are considered to be superclasses with respect to a given class. In specification of an attribute as a class an instance section is not explicitly introduced.

Treating of an object attribute as an association class provides for introduction of association metaclasses setting a classification relationship on attribute classes and establishing properties of association classes and/or their instances. It is said that an object attribute (as an association class) belongs to a particular attribute category that is explicitly introduced by an association metaclass. Association metaclasses are structured similarly to metaclasses. An association metaclass hierarchy is constructed similarly to a metaclass hierarchy. However, association metaclasses have several predefined attributes. A specification of an association metaclass is defined as follows:

---

4 Association metatypes are defined similarly. The only syntactic difference is an indication of in : `association`, `metatype` instead of in : `association`, `metaclass`.

---
Here an anonymous form (subsection 7.2) of an instance type (instance-instance type) of an association class is presented. At the same time a complete form of a specification is allowed when a type name and its supertypes are indicated. This provides for using of common specifications of types for association classes and for their instance objects applicable for different association metaclasses. Besides that a complete form provides for an introduction of a hierarchy of such types that allows to formulate rules of substitution of given associations by the others.

Association metaclass names (for direct and inverse associations) are used for category names of class attributes. An association type is defined in an instance_section. (In the syntactic rules an anonymous form of a type specification is given. A complete form is similar to the given in subsection 7.2). In an association type specification two pre-defined attributes domain and range can be given. If these attributes are not given then an association type does not restrict a domain (range) of an association class belonging to the metaclass. If a domain of an association type is given then an object class attributes of which are declared belonging to this attribute category should be included into the association type domain. If a range of an association type is given then an object class (type) of an
7.4 Association metaclasses

An association type (as a type of binary relation) is set by an attribute association_type. If an association \( R \) is defined on a domain \( C_1 \) and a range \( C_2 \) then the bounds define the following. The first bound gives for any object \( c_1 \) of \( C_1 \) an admissible range (minimal and maximal value) of a number of different objects (values) \( c_2 \) of \( C_2 \) such that \( \langle c_1, c_2 \rangle \) belongs to \( R \). The second bound for any \( c_2 \) of \( C_2 \) gives a minimal and maximal value of a number of objects \( c_1 \) of \( C_1 \) such that \( \langle c_2, c_1 \rangle \) belongs to an association inverse to \( R \). \( \inf \) is a constant denoting an arbitrary positive integer.

By means of an instance section another attributes of an association type are defined as usual, that is “attributes of attributes” can be defined (e.g., for an attribute price attributes of this class can be introduced, such as price_status and currency). By an instance - instance section attributes of association class instances are introduced. If for a particular object class attribute a metaslot is declared and it is defined that the attribute belongs to a certain attribute category (categories) then for a specification of an association class attributes or for a specification of attributes of its instances a union of the respective specifications given in a metaslot and in an instance sections or instance - instance sections of the respective association metaclasses is formed.

Association metaclasses as well as ordinary classes can in their turn become objects of another metaclasses. It is essential that an object attribute as an association class can be related to several attribute categories. Association metaclasses provide for introducing of attribute categories. Category concretization is realized at the attributes that are declared as instances of such metaclasses (or attribute categories).

Attribute categories provide constraints for their admissible concretizations. Such constraints besides mentioned above are defined by means of assertions. As soon as an attribute is related to a category, consistency constraints defined in an association metaclass (subsections 8.3, 8.4) are applied to it. Thus association metaclasses provide for specification of the required attribute properties.

Possibilities of association concretization and of substitution of one association by another one as a parameter are determined by a type relationship defined for the associations. A substitution is allowed if types corresponding to associations are related by a subtyping relationship.

In the language there exists a possibility of indication that an attribute of a class is an inverse function of another attribute. An example of classes \( c1 \) and \( c2 \) having conjugate attributes \( a \) and \( b \) defined by a metaclass

attribute (defined as belonging to this attribute category) should correspond to (should coincide with or should be a subclass / subtype of) an association type range.
Class representation facilities

mandatory\_automatic similarly to a Codasyl data set with mandatory automatic class membership is defined below. In metaslots of specifications of a and b attributes it is shown that they are inverse functions with respect to each other.

An association metaclass specification:

\{
  \text{mandatory}\_\text{automatic};
  \text{in: \text{metaclass, association}};
  \text{inverse: \text{automatic}\_\text{mandatory}};
\}

\text{instance\_section:}
\{
  \text{association\_type: \{\{0, inf\}, \{1,1\}\};}
  \text{domain: class};
  \text{range: class}
\}

Specifications of classes c1 and c2:

\{
  \text{c1};
  \text{in: class};
\}

\text{instance\_section:}
\{
  \text{a: c2;}
  \text{metaslot}
  \text{in: \text{automatic}\_\text{mandatory}};
  \text{inverse: c2.b}
  \text{end }
\}

\{
  \text{c2};
  \text{in: class};
\}

\text{instance\_section:}
\{
  \text{b: \{set; type\_of\_element: c1; \};}
  \text{metaslot}
  \text{in: \text{mandatory}\_\text{automatic}};
  \text{inverse: c1.a}
  \text{end }
\}

Further an example of a metaclass attribute is defined to which belong all attributes being stored functions. get and set are methods of all instance classes, i.e. methods specified for all attributes.

\{
  \text{attribute}
  \text{in: \text{metaclass, association}};
\}

\text{instance\_section:}
\{
  \{get; in: function;
  \text{params: \{+t/\text{instance\_of}\text{of}(type), - v/this.range\};}
  \langle \text{read} \rangle \}
  \{set; in: function;
  \text{params: \{+iv/this.range, +t/\text{instance\_of}(type), - ov/this.range\};}
  \langle \text{write} \rangle \}
\}
7.5 Transaction classes and activities

Any action in the mediation environment can be realized in frame of a particular transaction. In simple cases ACID transaction model is assumed. A transaction class is similar to classes of other objects in a sense that a set of objects representing transactions can be related to it. A specification of a transaction class defines a generic class if it contains parameters params. Local variables of transactions are defined by attribute specifications.

An example of a transaction class specification:

```plaintext
{ employee_op;
in: transaction;
instance_section:
{ name: string;
  belongs: companies;
  works: { set; type_of_element: projects;};
ce: employees;
hire: { in: function;
  params: { +name, +belongs, +works, -emp};
  { predicative; { include(e,employees) &
      e.name' = name & e.works_on' = works &
      e.belongs_to' = belongs & emp=e } }
}
fire: { in: function;
  { predicative; { exclude(e,employees) } }
  }
}
```

Here a predicative specification of functions is given (subsection 6.3). transaction metaclass defines properties of ACID transaction classes. An instance section defines a transaction object type. (Here an anonymous
type specification is given. A complete form is analogous to the form
given in subsection 7.2.)

A creation of a transaction object in the language is analogous to a
creation of any other object. Every functional attribute of a transaction
class is treated as a transaction that is called as follows:

\[
\text{< function identifier > ( < object variable >, < coupling mode >,
< actual parameter list >),}
\]

where object variable defines an object of a transaction class. A cou-
pling mode is \text{deferred, decoupled, immediate}, etc.

Committing of transaction results (section 10) is provided by a state-
ment \text{commit( < object variable >).} Abort or rollback of transactions is re-
alized by specific statements (section 10). For representation of complex
long-running activities an atomic transaction model is not sufficient: ac-
tivity (workflow) models or flexible multitransection models are required
relaxing the limitations of atomic model. Different multitransection mod-
els can be represented by the language (such as nested, splitted, coopera-
tive, polytransaction models and others). Since such models have much
in common it is reasonable to represent them as a metaclass general-
ization hierarchy (based on the category principle) in which built-in or
pre-defined transaction metaclasses are located on each level defining spe-
cific multitransection models (submodels). Types corresponding to such
models (due to the category principle) are related by the subtyping re-
lationship. Thus any of the multitransection models can be specified by
means of a metaclass or a metaclass hierarchy. In this hierarchy the trans-
action model properties corresponding to superclasses are propagated to
the underneath transaction metaclasses.

An activity defines computations over information resources inter-
leaved with interactions with an application environment of an informa-
tion system that makes possible to model application activities. Activities
are distinguished of transactions by relaxing the ACID requirements.

Every activity consists of a number of actions being in their turn an
activity or a transaction. Activities have a hierarchical structure in which
other activities or transactions play subordinate role. A term "action"
will be used in cases when we do not care whether we have in mind an
activity or a transaction.

Activity behavior rules are dependent on an activity model that is de-
efined by a metaclass of the activity class. The rules below are fixed by the
built-in \text{nested_activity} metaclass. Similarly to transactions, activities can
be in an active state, in a committed state or in a state when an activity
has been aborted. Activities in a hierarchy should satisfy the following
requirement. A parental activity can be committed only after termination
of subordinate activities. To an activity hierarchy (contrary to a trans-
action hierarchy) serializability requirements as well as a mutual order of
commitment of hierarchy members are not applicable. Such requirements
can differ for different transaction models. In an activity hierarchy commit
of subordinate activities is realized independently of a parental activity.
On abortion of a parental activity all its subordinate active subactivities
and main transactions should be aborted, for all subordinate activities
that have been already committed compensating activities (having dif-
ferent organization for different activity (transaction) models) should be
activated.

In the SYNTHESIS language complex activities are specified by scripts
(subsection 7.6) providing for a definition of arbitrary dynamic construc-
tions consisting of activities, transactions and assertions enforced by events
that appear in an application environment or in course of running of ac-
tivity components.

7.6 Script classes

Scripts are used for description of long-running activities (transactions)
in terms of sequences of actions leading to changes of states of resources.
Mechanisms of scripts are quite different from a behavioral model of ob-
jects based on a message passing and method call. Like a Petri net, a
script models the dynamic system behavior in terms of states (or places)
and state transitions. Scripts are specified as classes, instances of which
correspond to specific script instances. Script class hierarchies that are
related by generalization / specialization relationship (respective types of
scripts should satisfy a subtyping relationship) are allowed.

A script type is specified in an instance section of a script class. An
anonymous form of a type specification is given in syntactic rules below.
A complete form is analogous to the form given in subsection 7.2. Like
a Petri net, a script models the dynamic system behavior in terms of
states (or places) and state transitions. As usual, a net is represented by
a bipartite directed graph arcs of which connect nodes taken from two
sets: a set of states and a set of transitions. Arcs of the graph are defined
by lists of input and output states of transitions. Script dynamics are
defined by tokens that are produced in the initial states of a script on its
creation, in output states of transitions or are coming to external states
of a script from the outside. In one script instance a number of tokens of
different types can coexist. The states are places where sets of tokens can
be accumulated to wait for appropriate conditions to enter transitions.
Token type for a state is given by a phrase token. Token identifier can be
used in assertions (section 8) and conditions.

A state specification of a script can include a definition of assertions that should be satisfied in order that a token which was placed into the state could be activated. A specification of each transition can include a list of transition conditions that should be simultaneously satisfied in order that a transition could be fired and a description of actions that should be taken on such firing. Transition actions are prescribed by a function declaration given inside of the transition specification in the action slot. In particular, an action may lead to a call of a transaction (an activity) that can conform to various transaction (activity) models. A type of input tokens that can be moved into a transition from its input states is determined by input parameters of a function prescribing a transition action. Output parameters of a function determine output tokens that will be moved out of a transition after the function execution. Transition functions can create sets (of tokens) defined on types of output parameters of the functions. Functions having no parameters are allowed. In such cases tokens (input and/or output) have a type $T_{\text{none}}$. A type of ingoing to a state tokens should coincide with or should be a subtype of a token type defined for a state. A type of input tokens of a transition should coincide with or be a subtype of a type of the respective input parameter of a transition function.

On definition of a transition the rules for binding of its input and output may be given. Input (output) binding rules provide for establishing of a correspondence of tokens ingoing from (outgoing to) defined states to input (output) parameters of a transition function. Binding rules make possible to declare logical conditions that tokens ingoing from (outgoing to) defined states should satisfy and (or) to declare a number of tokens that should be consumed (produced) on an input (output) of a transition. This number is equal to one by default.

In each of the input states of a transition tokens can be accumulated and are considered in FIFO order. A collection of active tokens taken from all of input states (selected in FIFO order in quantity established for each input state) should be considered together to fire a transition. If a transition fires, all tokens of this collection are removed from input states. If several transitions are conflicting for consuming of the same tokens for transitions to be fired, nondeterministic choice of a transition is made.

The actions on transition firing consist in a call for a transition function with an actual parameters defined by tokens entering a transition according to input binding rules. As a result, tokens having types determined by output parameters of a function will be produced. If a function
has no output parameters an empty token will be formed.
To each state given by a list of output states to of a transition, tokens satisfying output binding rules and produced in the defined quantities will be directed. Syntactically one and the same state can be included several times into a list of input or output states of a transition.

In assertions declared for a state the variables referring to token attributes can be used alongside with the variables denoting arbitrary resources and script attributes. Such variables should be typed by a token type of a state. In assertions (in formulae) state names can be used to denote sets (of tokens). It is not obligatory to declare assertions for a state or conditions for a transition. If they are omitted, respective conditions are assumed to be true.

In a script external states can be specified to place tokens directed from other scripts or from objects. External input and output states of a script are declared differently. For declaration of an external input state functional attribute of a script should be defined. Names of these functions should coincide with names of states considered due to such definition to be external. Input parameter of these functions define a type of an external token (or a message) that could be directed to this state from another script or from an object. For that in a to list of a transition references to external states of another scripts (through gates) are allowed. Such names may be qualified by a script class name. Placing of a token (message) into an external input state is a result of a respective method call that will be associated with a script as an object. External output states are declared in a state section as gates. For external output states assertions should not be given. In transitions the names of external output states may be qualified by variables referencing proper instance of a script containing respective input states.

Using initial, it is possible to define initial marking of a script by declaring of constants (denoting tokens) that should be placed into predefined states on a script creation. The constant type should correspond to a token type of a given state.

A script is interpreted as a multitransection in which separate script components (assertions, conditions, actions) are executed as subtransactions. Such subtransactions can use different coupling modes (subsection 8.1) with a main transaction. A coupling mode for various script components syntactically is defined analogously to assertions (subsection 8.1): to a respective component of a script (an assertion, a condition or an action) a metaframe or a metaslot is appended that contains:

- a slot coupling taking an identifier of a coupling mode (immediate, deferred, decoupled, causal_decoupled) as a value;
• a slot priority giving a priority value of a respective transaction.

< script class specification > ::= < frame name > |
   { < class identifier > ;
   in : script[, < metaclass name list > ];
   [ params : {< formal parameter list >}; ]
   [ superclass : < superclass list >; ]
   instance_section : { < script type specifier > }}

< script type specifier > ::= { < type identifier >; |
   {< type identifier >; ]
   [ in : type, script[, < metatype name list > ]; ]
   [ supertype : < supertype list with class parameters >; ]
   [ rename : {< renaming list >}; ]
   [ < attribute specification list >; ]
   [ initial : < initial marking >; ]
   states : < state section >;
   [ gates : < state section >; ]
   transitions : < transition section >; }

< element of initial marking list > ::= |
   { < state identifier > [, < factor >][, < constant >] }|

< state section > ::= < state specification >|
   < state specification >, < state section >

< state specification > ::= < frame name > |
   { < state name > ;
   [ token : { < identifier > : < type > ; } ]
   [ assertions : { < assertion list > } ]}

< transition section > ::= < transition specification >|
   < transition specification >, < transition section >

< transition specification > ::= < frame name > |
   {< transition name > ;
   from : < state name list >; [bind_from : < input binding list >; ]
   to : < state name list >; [bind_to : < output binding list >; ]
   [ conditions : < condition list >; ]
   action : < action spec > }

< element of output binding list > ::= |
   { < state identifier > |
   [ , < factor >][, < condition >] }

< element of input binding list > ::= |
Hierarchical scripts in which a transition may be substituted by a whole (child) script can be defined. Parameters of a child script should include input and output states of a transition that should be substituted. In an action of a transition to be substituted instead of a function declaration such child script can be referred by a script call providing by actual parameters, such as actual input and output states of a transition (alongside with other parameters of a script). An identifier of a built-in type state is allowed to type formal parameters of a transition. An example defining hierarchical script types follows.

```plaintext
{trscript;
in type, script;
states:
  \{s1; \ldots\}; \{s2; \ldots\};
transitions:
  \{A;
    from: s2; to: s1;
    action: \{subscript1; params: s2, s1 \}
  \},
  \{B;
    from: s1; to: s2;
    action: ...
  \};
}
{subscript1;
in: type, script;
params: \{sp1/state, sp2/state\};
states: \{ss1; \ldots\}, \{ss2; \ldots\}, \{ss3; \ldots\};
transitions:
  \{P;
    from: ss2; to: ss1;
    action: ...
  \},
  \{Q;
    from: ss1; to: ss3, sp2;
```

```plaintext```
8 Facilities used for expressing assertions on application domain entities

8.1 Mediator and information resource assertions

Declarative and procedural assertions are used in the language for different purposes, for instance, as rules for generation of values, as consistency constraints of a mediator or collections of information resources, as triggers or demons. Resource assertions can be associated with separate attributes of a type (or a class), with a type (or with a class) or with a collection of resource components represented by a singleton or by a class. Assertions are specified in assertion entries of an assertion section of a module or are directly provided in specifications of types or in specifications of resource components.

All resource assertions are considered as instances of a built-in class assertion of assertional objects. Attributes of this class instances will be defined later. Each assertion (an instance of an assertion class) is represented in the language as a frame having predetermined structure. Slots
of this frame correspond to attributes of instances of a built-in class assertion. It is assumed that frames specifying assertions contain an implicit slot in : assertion. Due to that they are transformed into objects of a class assertion. Such treatment of assertions provides for establishing of associations between types (classes) and assertions as between objects. In such associations the assertions play a role of type attribute values. They are related to a type directly by introducing in a type specification of attributes associating a type with assertions as the attribute values (sections 6.4, 7). Indirect relationship of types with assertions consists in introducing of attributes the metaslots of which contain assertions playing roles defined by their specifications. Thus, specified directly or indirectly, assertions define type properties (and also properties of a resource component in a specification in which the type is used).

An example of a direct relationship of an assertion and a class:

```plaintext
{employees; 
in: class; 
instance_section:
{name: string; 
belongs_to: companies; 
works_on: {set; type_of_element: project}; 
on_emp_proj: {in: predicate, invariant; 
   {all e/employees (works_on(e) <= engaged_in(belongs_to(e)) )}}}
}
```

Classes companies and project are defined in subsection 7.2. An attribute on_emp_proj is an assertion that belongs to an attribute category invariant. Another example of a direct relationship of a class and an assertion is given in subsection 8.4.

An example of an indirect relationship of assertions and a class:

```plaintext
{car_model; in : class; 
instance_section : 
{model_id : id; 
metaslot 
a : invariant, 
{{all x/car_model, y/car_model( 
   model_id(x) = model_id(y) > > x = y)}}; 
b : invariant, {{obligatory}}
```

```
Here assertions express facts of uniqueness and of obligatory providing a value of an attribute model_id (obligatory is a built-in assertion (Appendix C)). Arbitrary attributes can be used to establish direct or indirect relationships of assertions and types. Association metaclasses provide for assigning of such attributes to particular generic categories defining properties of an association "type - assertion".

\[
\begin{align*}
\text{< resource component assertion section >} & ::= \\
& \quad \text{< empty >| assertion_rule :< assertion entry list >} \\
\text{< assertion entry list >} & ::= \text{< assertion entry >|} \\
& \quad \text{< assertion entry >, < assertion entry list >} \\
\text{< assertion entry >} & ::= \text{< frame name >|} \\
& \quad \{[< frame name >; ]} \\
& \quad \text{[resource :< resource component identifier list >; ]} \\
& \quad \text{[level :< level specifier >; ]} \\
& \quad \text{[enforcement :< event expression >; ]} \\
& \quad \text{[action :< function designator >; ]} \\
& \quad \text{assertions :< assertion list >; } \\
& \quad \} \text{ |< assertion >| } \{\text{< identifier >:< assertion >}\}
\end{align*}
\]

\[
\begin{align*}
\text{< level specifier >} & ::= \text{ global | locall} \\
\text{< event predicate >} & ::= \\
& \quad \{[< indicator > .]|[< event kind >]|. < event concretization >\} \\
\text{< indicator >} & ::= \text{ after | on | before | error} \\
\text{< event kind >} & ::= \text{ create | delete | update | commit | access | } \\
& \quad \text{transbeg | abort | time(< temporal type expression >)} \\
\text{< event concretization >} & ::= \text{< atom >}
\end{align*}
\]

A frame name in an assertion entry gives its unique identifier in a module. Assertion entries provide for structuring of a representation of resource component assertions. Assertions related to a resource component (represented by a singleton or by a class) or to a collection of resource components are included into one entry. In a common part of an assertion entry a common information for all or for a part of assertions can be included. The common part contains an information (on an assertion level) on an event expression being a condition of assertion enforcement, on resource components to which the assertions are related, and on actions (action ) that should be executed when assertion predicates are not satisfied.
Global or local level (section 5) determines whether an assertion is related to a mediator or to an information resource. By default an assertion level is considered to be global.

An event predicate is given by a label (generally - a qualified name) defining a kind of an event (creation of a new object, deletion of an object, update of an object, a transactional or temporal event, such as end of a day, of an hour, of a second or of another temporal units). An indicator determines a phase of application of an assertion with respect to an event (directly before, directly after, on error, etc.). An event kind list is extensible. An event can be concretized by an atom (subsection 9.2), indicating (when an event is detected) what data should be modified, what function (transaction) should be executed and for what parameters. An atom can denote a predicate defining a condition that if satisfied leads to a detection of an event. In formulae combinations of event predicates are allowed that are defined by rules of event expression specification (subsection 8.5).

Absence of a slot enforcement means that this assertion can be referred from a type (class) specification to use it in a role defined by a respective attribute category.

Syntactically each assertion is represented by a frame:

```plaintext
< assertion > ::= { < frame name > : 
    [resource : < resource identifier list > ; ]
    [level : < level specifier > ; ]
    [enforcement : < event expression > ; ]
    [action : < function designator > ; ]
    < assertion slots >
}
```

A name of an assertion frame can determine a built-in assertion (Appendix C), an assertion in an assertion entry or a name of a new concrete assertion. If an assertion is represented by a function then assertion slots and a function type are the same. An assertion frame includes slots specific for each type of assertions. They can be preceded by a list of identifiers of resource components to which an assertion is related, by a level specifier, by an event expression and (or) by an action. If such slots are defined in assertions included into an assertion entry the definition overloads a common specification of the assertion entry. For names of variables, predicates and functions in assertions (alongside with others) the names of types, classes or modules can be used.

It is allowed to form separate information resource specification modules containing only the assertion section. In such modules it is possible to declare a list of imported modules containing type, class and function
specifications that are referred by assertions. Another possibility consists in providing of a parameterized module containing only the assertion section with a subsequent concretization of the module during its import into various modules of resource specifications.

Any assertion in a module can have a unique name established by its identifier or a qualified identifier:

\[
<\text{assertion entry identifier}> . <\text{assertion identifier}>
\]

Everywhere a list of assertions is:

\[
<\text{assertion list}> ::= \{ <\text{named assertion list}> \}
\]
\[
<\text{named assertion list}> ::= \\
\quad <\text{named assertion}> | \\
\quad <\text{named assertion}>; <\text{named assertion list}>
\]
\[
<\text{named assertion}> ::= [<\text{identifier}> ] <\text{assertion}> | \\
\quad [<\text{module identifier}> . ] <\text{assertion identifier}>
\]

The following assertions of a general kind can be used:

\[
<\text{general assertion}> ::= <\text{function declarator}> |
\quad <\text{function designator in assertion}>
\]
\[
<\text{function designator in assertion}> ::= \\
\quad \{ <\text{function identifier}> ; \\
\quad \quad \quad \text{params} : <\text{actual parameter list}>
\}
\]
\[
<\text{function identifier}> ::= \\
\quad <\text{identifier}> | <\text{variable of function type}>
\]

Function declarators without parameters, function declarations by a logic program (or by a single formula) when a function type can be inferred from a formula or from a collection of rules can be used for assertions. In case of a function designator actual parameters are defined in a context of an assertion (a resource component or a collection of resource components). Variables of a function type in a function designator can get values dynamically. Event expressions (subsection 8.5) constraining an order of actions on a resource component interface are also allowed as assertions. A frame name as a general assertion denotes a frame specifying an assertion. A temporal assignment is associated with an assertion that defines a period of time when the assertion is valid.

A scheme of realization of assertions is the following. Generally assertions are executed in a form of a hierarchical transaction (activity) model. An assertion leads to creation of nested transactions which cause an enforcement of the assertion. When a need of enforcement is determined, a nested transaction \( C \) is created that evaluate predicates of the
assertion. If predicates are not satisfied then another nested transaction \( A \) that performs an action is created.

Different modes of coupling of nested transactions \( A \) and \( C \) with an upper layer transaction (activity) can be provided. Syntactically coupling mode is given as follows. To an entire assertion frame and (or) to an action slot a metaframe and a metaslot are attached containing:

- slot `coupling` taking as a value an identifier of coupling modes (immediate, deferred, decoupled, causal_decoupled);
- slot `priority` providing a priority value of the respective transaction.

Coupling modes have the following meaning:

- **immediate** – a transaction (action) of an upper layer is suspended until an assertion predicate (and if required, an action) will be evaluated;
- **deferred** – an assertion predicate is evaluated at the end of a transaction (action) of an upper layer (for a transaction immediately before commit);
- **decoupled** – an assertion predicate is evaluated as an independent transaction so that its evaluation is planned independently of a transaction (action) of an upper layer;
- **causal_decoupled** – an assertion predicate is evaluated as an independent transaction; its evaluation is planned after completion of an activity of an upper layer (after commit for a transaction).

A coupling mode list can be extended and modified.

Different coupling modes can be given for evaluation of an assertion predicate and for execution of an action. If for assertion or action a coupling mode is not given it is prescribed by default. If a coupling mode for action is not given and for assertion is given then for action it is considered to be the same as for an assertion predicate.

If a function that defines an action is specified in a particular transaction class then this class should not contradict to the coupling mode. If an action is not explicitly defined then it is assumed that a coupling mode for standard actions is the same as the mode given for the entire assertion. For built-in assertion attribute categories (e.g., `invariant`) a deferred mode is assumed.
8.2 Built-in assertions

Set of built-in assertions is included into the language. The assertions were defined in process of synthesis of canonical data model for a set of structured data models. Such assertions could have been expressed by function specifications. In spite of that the built-in assertions are introduced for readability and for efficiency of implementation.

A set of built-in assertions include:

\[
< \text{built-in assertion} > ::= <\text{assertion of uniqueness}> | \\
<\text{assertion of constancy}> | <\text{assertion of definiteness}> | \\
<\text{assertion of conditional uniqueness}> | <\text{assertion of conditional constancy}> | <\text{assertion of order}> | \\
<\text{function assertion}>
\]

Description of built-in assertions is given in Appendix C.

8.3 Metalevel assertions

Defining metatypes and metaclasses it is useful to introduce assertions influencing types and classes - instances of the metatypes and metaclasses. Assertions can play different roles. For instance, a role of deductive rules or of a consistency constraints. Built-in facilities by means of an event label make possible to set this difference and to define a moment of assertion enforcement. Thus, a label on access corresponds to deductive rules.

An arbitrary role of assertions and conditions of their enforcement can be defined by means of association metaclasses defining a kind of assertion relating types and assertions. A special predicate holds of the language taking an assertion as an argument provides a true value when an assertion is true in an information resource base. It is natural to introduce association metaclasses with which general assertions are related. These assertions use a predicate holds that refers to assertions used in classes. The latter can be represented by open formulae containing free variables that will be bound after a substitution of an assertion into a metaclass formula that uses a predicate holds.

Example. An association metaclass single_valued that establishes a constraint that to one and the same object in a domain class of an association not more than one instance of a range class of the association can correspond.
8.3 Metalevel assertions

\{single_valued; in : metaclass, association;
instance_section :
\{domain : class;
range : class;
assert : {
\{sv_check; in : predicate, invariant;
\{(holds(all x/this.domain.inst, v/this.range.inst, w/this.range.inst(
in(x, this.domain) & in(v, this.range) & in(w, this.range) &
in([x,v], this) & in([x,w], this)− > same(v, w))})
\};
\}}\}

An attribute assert defines an assertion list.

Example. An association metaclass bij that states that two classes extents have instances that are in one-to-one correspondence.

\{bij; in : metaclass, association;
instance_section :
\{domain : class;
range : class;
assert : {
\{two_way_check; in : predicate;
enforcement : on_commit:
\{\{holds(card(this.domain) = card(this.range) &
all x/this.domain.inst, y/this.domain.inst(
in(x, this.domain) & in(y, this.domain) & ¯same(x, y) − >
¯same(this(x), this(y)))\})
\};
\}}\}

To refer to assertions by means of holds the following variants of assertion naming exist:

- by identifier
  
  < class name > . < assertion identifier >

  if assertions are directly related to a class;

- by identifier
  
  < class name > . < attribute identifier > . < assertion identifier >
8 Facilities used for expressing assertions

if assertions are indirectly related to a class;

- by its own name if assertions are defined in a resource component assertion section.

Qualification of an assertion identifier by an identifier of a module, a class or an attribute is required in case of a nonunique identification.

8.4 Built-in assertional attribute categories

There is a number of built-in attribute categories possessing specific properties for association of assertions with types (classes):

- invariant: an assertion is an invariant;
- assertion: an assertion should be enforced in a period set by particular assertion;
- initial: an assertion should be enforced when an object becomes an instance of a class;
- final: an assertion should be enforced when an object cease to be an instance of a class.

Such categories can be used for direct and indirect association of assertions with types.

To relate assertions with transaction types special built-in association metaclasses are used: pretrans (posttrans) – an assertion is a runtime consistency constraint that should be satisfied before a transaction execution (pretrans) or at a transaction completion (posttrans) on a commit time.

To indicate that an assertion or a function defined in a type are given by a predicative specification, an association metaclass predicative is used: predicative: <function> – a predicative specification (by mixed pre- and post-conditions) of actions to be implemented by a transaction (function); the specification defines a precondition to run a transaction (function) (these conditions should be satisfied on a transaction (function) call) and conditions that should be satisfied in a state set by a transaction (function) on its completion.

To simplify definitions of assertions related to a transaction class the built-in predicates are used in the language:

- added(<variable>, <class name>) – becomes true in a terminating state if the variable denotes an object created within a given class;
8.5 Event expressions

Internal and external events are distinguished. Internal events appear in a mediation environment. External events appear in an application environment of a mediator. Primitive events are defined by built-in predicates a list of which can be extended. Besides that actions executed in a resource component or in an application environment are also considered as primitive events. Such events are denoted by predicates having names coinciding with names of respective actions (transactions, functions) and arguments coinciding with arguments of respective functions. Thus event predicates can contain variables. Immediate actions as well as actions having an arbitrary time duration correspond to events. Alongside with event predicates, event expressions can contain predicates restricting variable domains in event predicates. For example, an event "before payment of a large tax" is defined by means of a predicate "before execution of a method of a tax payment" and of a predicate "tax amount is greater than T".

Event expressions define rules of creation of compound events from simpler ones (from primitive events and their compositions). It is assumed that to each primitive event on its appearance a unique event identifier and event time correspond. Each event is considered to be an object belonging to an event class.

```
< event term >::=< event predicate >|< event predicate >&
    |< condition − formula >|< built − in term >|
    |< empty event term >|( < event expression >)
< empty event term >::= 1
< event expression >::=< event term >|
    |< event sequence >|< event choice >|< event iteration >|
    |< parallel event composition > |
    |< nondeterministic event choice > |
    |< event interleaving >
< event sequence >::=< event expression >. < event term >
< event choice >::=< event expression > + < event term >
< event iteration >::=< event term > *
```
9 Logic formulae representation

9.1 General rules

Logic formulae (or simply formulae) in the SYNTHESIS language are used to define rules in logic programs, to formulate assertions (e.g., consistency constraints of a mediator or an information resource), to express programs (queries) over a mediator, to form predicative specifications.

To specify formulae a variant of a typed (multisorted) first order predicate logic language is used. Every predicate, function, constant and variable in formulae is typed.

Predicates in formulae correspond to classes, worlds, collections and functions that are specified in resource component specification modules and have respective types.

For terms the variables, constants and function designators (in a specific case – expressions) are used. Each term has a well defined type. An expression denotes a function with arguments represented by variables in
the expression having types defined by an expression context. A result of this function and its type are implicitly defined by an expression.

An atom (atomic formula) appears as p(t₁,...,tₙ), where p is a predicate and tᵢ are terms. Atomic formulae define simple statements related to resource components.

In simple form a formula is an atom or appears as:

\[ w₁ \land w₂ \quad (w₁ \text{ and } w₂) \]
\[ w₁ \mid w₂ \quad (w₁ \text{ or } w₂) \]
\[ w₂ \quad (\text{not } w₂) \]
\[ w₁ \rightarrow w₂ \quad (\text{if } w₁ \text{ then } w₂) \]
\[ \exists x/T(w) \quad (\text{for some } x \text{ of type } T, \ w) \]
\[ \forall x/T(w) \quad (\text{for all } x \text{ of type } T, \ w) \]

where \( w, w₁, w₂ \) are formulae.

Variables in formulae can be bounded by quantifiers. A scope of a quantifier in a formula with quantifier is a formula \( w \). Occurrences of variables denoted by a quantifier in the scope of the quantifier are bounded and are free for their occurrences out of the scope. A notation \( x/T \) defines that a type of a variable \( x \) is \( t \).

Rules are represented as closed formulae of the form: \( a : -w \) where \( a \) is an atom and \( w \) is a formula. \( a \) is a head and \( w \) is a body of a rule. All free variables of \( a \) and all free variables of \( w \) are assumed to be bounded by a universal quantifier placed before the rule. If \( w \) is absent and attributes of \( a \) have constant values then such rule is a fact.

If a formula is used as a query to a mediator (such query is called a filter) and has \( x₁, ..., xₙ \) as free variables then the query results in a collection of substitutions of well-typed values of \( x₁, ..., xₙ \) such that on each of them a filter interpreted on an information resource base gets the true value. Formulae semantics are such that based on such substitutions a resulting collection is formed containing objects, ADT values or frames as its instances. In the sequel in cases when an exact nature of values and collections is not important, general terms "value" and "collections" are used (assuming that sets, worlds and classes are specializations of collections and that objects and frames are special cases of ADT values).

Formula representation facilities in the SYNTHESIS language provide for more variety of formulae.

### 9.2 Formulae construction rules

Variables in formulae denote attributes of types and classes, slot names, program variables, function arguments. Variables in formulae can denote
also attribute categories of types (subsection 7.4). Such variables denote type attributes belonging to a given category.

Representation of variables in formulae:

\[
<\text{variable}> ::= <\text{compound identifier}> | <\text{index variable}>
\]

\[
<\text{index variable}> ::= <\text{identifier}> \[<\text{term list}>\]
\]

\[
<\text{compound identifier}> ::= <\text{identifier}> \mid <\text{identifier}> \[<\text{arithmetic expression}>\]
\]

\[
<\text{typed variable}> ::= <\text{variable}> /[<\text{type expression}>]
\]

A generalization/specialization hierarchy and a compositional hierarchy of classes are distinguished. A generalization/specialization hierarchy is defined by a "superclass-subclass" relationship in a class specification. A compositional hierarchy of classes (types) is formed by means of attributes specified in classes (types) so that an attribute of a class (type) is typed by a class or an ADT name. In case when an array is a type of a class attribute, for identification of array instances indexes are used. A typing of formulae by type variables provides for a specification of formulae parameterized by a type. A scope of a free variable is the entire formula, a scope of a bounded variable coincides with a scope of a quantifier.

A syntax of typed variables provides for necessary changes of denotations for establishing a correspondence of variables to class (type) attributes and for a proper denotation of resulting variables of subformulae to form a final result of a formula. It is required to provide a name of a type (class) or an attribute in a type designator of a typed variable if a type (class) cannot be uniquely inferred from the context.

An index variable is used for denotation of variables of a product type. An order of terms in an index variable should correspond to an order of respective attributes in a product type. A term type should coincide with a type of the respective attribute of a product type or should be its subtype.

It is essential that in one and the same scope denoting of variables having compatible types (one type is a subtype of another type) by the same names could be allowed.

\[
<\text{term}> ::= <\text{typed variable}> | <\text{entire value}> |\n\quad <\text{atom}> | \{<\text{arithmetic expression}>\} | <\text{multiterm}>
\]

\[
<\text{entire value}> ::= <\text{value}> [<\text{temporal assignment}>]
\]

\[
<\text{value}> ::= \text{none} | <\text{constant}>
\]
9.2 Formulae construction rules

Constants of complex types and multiterms are represented by frames or by a special notation (Appendix B).

\[
<\text{predicate name}> ::= <\text{resource name}> | <\text{function name}> |
<\text{collection name}> | <\text{context}>
\]

\[
<\text{function name}> ::= [ <\text{module identifier} >]. <\text{function identifier}> |
<\text{function type variable identifier}>
<\text{attribute name}>
\]

\[
<\text{collection name}> ::= <\text{compound identifier}>
\]

\[
<\text{atom}> ::= <\text{predicate name}> | <\text{predicate name}>(<\text{term list}>)
\]

\[
<\text{term list}> ::= <\text{term}> | <\text{term list}>, <\text{term}>
\]

Collections of values of abstract data types constitute an interpretation domain of formulae. A collection of values (collection of objects, collection of frames) corresponds to a predicate-collection (including a predicate-class and a predicate-context). Thus a predicate-class or a predicate-collection are always unary. A collection type and a collection instance type should be distinguished. A unary predicate corresponds to a collection, a variable (argument of the predicate) takes as its values instances of the collection (ADT values) having a type of the collection instances or its reduct.

Thus to an atom that appears as

\[
<\text{predicate name}>
\]

or

\[
<\text{predicate name}>(<\text{term}>)
\]

where a predicate name defines a collection, a collection of instance values or their projections corresponds. These values should have been created or derived (in case of views) up to the moment of the predicate evaluation. In case of a predicate-class all created (derived) objects in all direct and transitive subclasses of a given class are included. Such set is known as a total extent of a class. If it is required to use only own class extent that does not contain instances of its subclasses, it is necessary to mark predicate-classes in formulae by * placed before a name of a class. A singleton corresponds to a resource component predicate represented by a single object.

By a typed variable term it is possible to define a reduct of a collection instance type (e.g., of a class instance type).

Thus generally an unary predicate in formulae corresponds to a collection defined on an abstract data type.

In case when a predicate name is a function name then a function designator (as a predicate symbol) corresponds to it.
< function designator > ::=  
< function identifier > [(< actual parameter list >)]  
< actual parameter list > ::= < term list >

Everywhere when attribute names of ADT values are used, a call of the respective function associated with an ADT value is assumed. In this case a term used as the first parameter of a function designator should denote an ADT value on the interface of which a respective function is defined. All other parameters (terms of the function designator) positionally should correspond to formal parameters of a function declarator. A function designator is a logic representation of a result of a function computation that has a type of a result of a function. In predicates - functions as well as in terms included into index variables it is allowed to use terms corresponding to parameters - results of functions. To such term in an index variable a variable corresponds with a type and a denotation defined by a resulting variable of the function designator or with its subtype.

In case of attributes – stored functions (declared as $a: T,$) where $T$ is an attribute type, during application of a function designator $a(o)$ it is implicitly assumed that generic functions get or set are applied that extract or modify a value of an attribute $a$ belonging to an object or to an ADT value $o$.

Arbitrary compositions of functions can be used in formulae (in this case a function designator is used as an actual parameter). Compositions of functions having a function designator for the first parameter play a specific role. Such compositions provide for construction of path expressions denoting a path in a compositional class (ADT) hierarchy.

For instance, let specifications of classes family and person appear as:

```plaintext
{person; in: class;
 instance_section:
 {name: string;
  age: integer;}
}

{family; in: class;
 instance_section:
 {name: string;
  mother: person.inst;
  father: person.inst;
  children: {set; type_of_element: person.inst;}}
}
```

Let a variable $f$ denotes an object of the class family. Then the
composition of functions \( \text{name(mother(f))} \) defines a mother name in a family \( f \). Alongside with a bracketed notation more simple dot notation is allowed: \( f.\text{mother.name} \) having the same interpretation.

One implicit attribute \( \text{self} \) having a unique identifier of the object is included into each object. Extraction of this value from an object \( o \) can be obtained by a function designator \( \text{self}(a) \).

Predicates of the following kinds can be used in formulae:

\[
\begin{align*}
\text{< predicate> ::= & <atom> | <frame path expression> |} \\
\text{& <condition> | <compound condition>} \\
\end{align*}
\]

Frame path expression is defined in this section.

\[
\begin{align*}
\text{<condition> ::= & <function designator> |} \\
\text{& <arithmetic expression> | <sigma> <arithmetic expression>} \\
\text{& <condition list> ::= & <condition> | <condition list>, <condition>} \\
\text{& <compound condition> ::= & ** <condition list> **} \\
\text{& <sigma> ::= eq | lt | le | ge | gt | ne | <selector of function>} \\
\end{align*}
\]

Or alternatively:

\[
\begin{align*}
\text{<sigma> ::= & <|<=|=|>|<> | <selector of function>} \\
\end{align*}
\]

The comparison signs denote implicitly defined functions for different objects and values (such functions should be defined in the respective types). In conditions an application of arbitrary predicates defined in abstract data types is allowed. For instance, a predicate determining belonging of an instance to a collection (to a class) looks as:

\[
\text{in(<variable>, <collection>)}
\]

A variable denotes here a value of the collection instance type of a collection and \( <\text{collection}> \) is an identifier of the collection or a formula.

A compound condition is applicable to values of types having a hierarchical compositional structure with attributes of a collection type. Conditions related to attributes located on different levels of a compositional hierarchy of a type are given in a condition list. An object (ADT value) satisfies a compound condition if its attribute values satisfy to all conditions in the condition list and these values belong to one and the same path in a compositional hierarchy of the object (ADT value) that includes attributes of a collection type. Such path should not contain cycles.

As predicates in conditions temporal associations can be used. For parameters of temporal associations it is possible to use:
• identifiers of components of compositions of functional predicates denoting respective abstract values;

• identifiers of variables typed by valid or belief - the types defined on collections of values of existence and belief time slices for abstract values;

• constructions appeared as
  
  `< attribute identifier > .valid`

  or

  `< attribute identifier > .belief`

  for attributes containing in metaslots of their specifications definition of a slot history (subsection 7.2).

\[< \text{arithmetic expression} > ::= < \text{additive} > | \]

  \[< \text{additive operation} > < \text{additive} > | \]

  \[< \text{arithmetic expression} > < \text{additive operation} > < \text{additive} > \]

\[< \text{additive operation} > ::= + | − \]

\[< \text{additive} > ::= < \text{multiplier} > | < \text{additive} > \]

\[< \text{multiplicative operation} > ::= * | / \]

\[< \text{multiplier} > ::= < \text{variable} > | < \text{number} > | \]

\[(< \text{arithmetic expression} >) | < \text{function designator} > \]

Arithmetic operations are treated as implicitly specified functions defined on numeric values.

The frame path expression predicate is given by a regular expression in which ‘\*' means iteration of zero or more times of the preceding term, ‘+’ means iteration of 1 or more times of the preceding term, ‘||’ means arbitrary selection of one of the term, ‘?’ means a sequence of terms, ‘?’ means that the preceding term is optional.

\[< \text{frame path expression} > ::= \]

  \[[< \text{frame identifier} > ] < \text{regular term} > : [ < \text{valuation} > ] \]

\[< \text{regular term} > ::= < \text{internal term} > [(+ | [?])] | \]

\[< \text{regular term} > . < \text{internal term} > [(+ | [?])] | \]

\[< \text{regular term} > || < \text{internal term} > [(+ | [?])] | \]

\[< \text{regular term} >)||(< \text{internal term} > [(+ | [?])] | \]

\[< \text{internal term} > ::= < \text{slot name} > [(+ | [?])] | \]

\[\text{metaframe}, < \text{slot name} > ] | \]
A collection of frames is relevant to a predicate that is identified by a path given in the scope of a formula that should contain atoms denoting collections including frames. Declaration of the resulting variables for a frame or slot identifier (defined for instance by a collection comprehension formula) means that a collection of frames or a collection of slot values included into the relevant frames will be produced as the result.

On different levels it is possible to associate temporal conditions with the elements of the frame path expression (denoting abstract values in a path defined by the expression). Such conditions are specified using slots valid and belief qualified by an identifier of a slot, a frame or a value of the respective level. Thus it is possible to uniquely identify a level on which the temporal condition should be taken into account (a frame, a slot, a value or a metaframe).

Formulae in the SYNTHESIS language are represented syntactically as:

\[
<\text{formula}\> ::= \\
\quad <\text{predicate}> | \\
\quad <\text{quantified formula}> | \\
\quad ( <\text{formula}>) | \\
\quad <\text{formula}> \&<\text{formula}> | \\
\quad <\text{formula}> \neg<\text{formula}> | \\
\quad <\text{frame path expression}> | \\
\quad <\text{collection comprehension}> | \\
\quad \text{group_by}(<\text{grouping list}>,"<\text{atom}\text{-}\text{collection}>",<\text{formula}>) | \\
\quad \text{order_by}([<\text{order}>],<\text{formula}>) | \\
\quad \text{form_class}(<\text{formula}>,<\text{class identifier}>, <\text{superclass identifier}>, <\text{specification variable}>)
\]
Constructing well-formed formulae, it is necessary to take into account semantic rules of their interpretation that are defined further in this section. Formulae including predicate-collections and not including them are distinguished.

Formulae without predicate-collections satisfy the following properties:

- formulae cannot include subformulae with constructions `order_by`, `group_by`, `on`, `as_of`; interpretation of such formulae is defined by the first-order predicate logic (subsection 9.1);
- formulae are used for specification of mixed pre- and post-conditions of functions, for specification of invariants and assertions;
- a formula given as predicative specification of function may refer to the state of the system before function invocation (pre-state) as well as to the state after function invocation (post-state), so that formula specifies the system state change (so called ”effect” or ”side-effect”);
- if a formula is required to state that variable $v$ value belongs to collection $C$, built-in predicate `in(v, C)` should be used.

Formulae including predicate-collections satisfy the following properties:

- formulae do not include effects;
- formulae are used as right-hand parts of rules of logic programs.

General rules of interpretation of formulae are considered in subsection 9.3.

\[
\text{< quantified formula > ::= ex} \text{ < typed variable list > ( < formula > )} \mid \text{all} \text{ < typed variable list > ( < formula > )}
\]

\[
\text{< typed variable list > ::= < typed variable >} \mid \text{< typed variable list >, < typed variable >}
\]

\[
\text{< set function > ::= min} \mid \text{max} \mid \text{count} \mid \text{total} \mid \text{average} \mid \text{< user function >}
\]

\[
\text{< grouping list > ::= { < compound identifier list > } }
\]

\[
\text{< atom ∼ collection > ::= < collection identifier > [ [ < term > ] ]}
\]

\[
\text{< order > ::= < order specifier >} \mid \text{< order list >}
\]

\[
\text{< order specifier > ::= asc} \mid \text{desc}
\]

\[
\text{< order list > ::= < ordering element >} \mid \text{< order list >, < ordering element >}
\]

\[
\text{< ordering element > ::= < order specifier >} < \text{identifier >}
\]
9.3 General rules of formulae interpretation

To any formula including predicate-collections a result is related in a form of collection of ADT values. The result is called a result set of a formula. Constructs of formulae creating joins of collections of ADT values lead to creation of new ADT values. Multicollection formulae forming joins, collection comprehension formulae, group by formulae belong to such constructs. An application of functional terms producing new ADT values can also form value creating formulae. All other kinds of formulae are value preserving. It is essential that using of simple reducts in formulae does not lead to creation of new values: an existing value gets a new reduced interface.

Result set of a formula defines its semantics. In this section a result set of a formula \( F \) is denoted as \( s[F] \) and semantic function \( s \) is defined by structural induction on formulae starting with atomic formulae – predicate-collection.

9.3.1 Atomic formulae

The simplest case of predicate-collection \( C(x/T) \) where type \( T \) is an instance type of the collection \( C \). If so the result set \( s[C(x/T)] \) is a collection of ADT values contained in the collection \( C \). In case when \( T \) is a reduct of the instance type of \( C \), values of the collection \( C \) should be transformed according to reduct value transformation rules.

A function \( \Omega_R \) transforms an instance of the source ADT of the reduct \( R \) into a set of instances of \( R \) (the set can have more than one element when there is a flattening in the reduct). The function \( \rho_R \) transforms a set of instances of the source ADT into a set of instances of the reduct using the function \( \Omega_R \) in the following way:

\[
\rho_R(p) = \bigcup_{p(o)} \Omega_R(o)
\]

Here \( p \) is a class and its instances are included into a set of instances of the reduct \( R \) of the source ADT.

Consider a reduct \( R \) specified as \( T[e_1, \ldots, e_n] \) where \( e_i \) defines a reduct element with a name \( a_i \), a type \( T_i \) and a path \( t_i \).

Initially we define the function \( \Omega_R \) for the case when there is no flattening in the reduct \( R \) (section 6.2). In this case the function returns a set which consists of one element, an instance of the reduct \( R \):

\[
\Omega_R(o) = \{ \{ a_1 : v_1; \ldots ; a_n : v_n; \} \mid v_1 \overset{T_1}{=} o.t_1, \ldots , v_n \overset{T_n}{=} o.t_n \}
\]
Here equality $v_1 \equiv_T v_2$ of ADT values $v_1$ and $v_2$ (values should be at least of type $T$ or its subtype) where type $T$ has attributes $a_1, \ldots, a_n$ of types $Q_1, \ldots, Q_n$ respectively is defined as

$$v_1 \equiv_T v_2 \iff v_1.a_1 \equiv_{Q_1} v_2.a_1 \land \ldots \land v_1.a_n \equiv_{Q_n} v_2.a_n$$

In the case of a built-in scalar type $T$ the predicate $x \equiv y$ is defined as equality of values $x$ and $y$:

$$x \equiv y \iff x = y$$

In the case when the type of the equality predicate is a set type with an element type $T$ the equality predicate is true if two sets having an element type $T$ are identical.

Example. Consider an ADT $T$

$$\{ T; \text{ in : type}; \\ \quad a : \text{ integer}; \\ \quad b : \text{ integer}; \\ \quad n : \{ \text{ in : type; } x : \text{ integer; } y : \text{ integer; } \}; \\ \}$$

and reducts $T[a, b], T[c : a, n/[x]], T[a, c : n.x]$.

Note. In examples for a better perception the following notation is used: $\{ a_1 = v_1, \ldots, a_n = v_n \}$ instead of $\{ a_1 : v_1; \ldots; a_n : v_n; \}$.

Let $o = \langle a = 1, b = 2, n = \langle x = 3, y = 4 \rangle \rangle$, then:

$$\Omega_{T[a, b]}(o) = \langle a = 1, b = 2 \rangle$$
$$\Omega_{T[c : a, n/[x]]}(o) = \langle c = 1, n = \langle x = 3 \rangle \rangle$$
$$\Omega_{T[a, c : n.x]}(o) = \langle a = 1, c = 3 \rangle$$

Further the function $\Omega_R$ is defined in the case when the reduct $R$ contains flattening. Let the attributes $b_1, \ldots, b_k$ in the path of the reduct elements have types

$$\{ \text{ set; type_of_element : } B_1; \}, \ldots, \{ \text{ set; type_of_element : } B_k; \}$$

and flattening is applied. For an instance $o$ of the source ADT $\Omega_R(o)$ is defined in the following way.

For an instance $o$ a set of flattenings $F(o)$ is constructed: every ele-
9.3 General rules of formulae interpretation

Definition of the set $F(o)$ is obtained from $o$ where the values of the attributes $b_1, \ldots, b_k$ are replaced with some elements from sets $s_k$ where $s_1, \ldots, s_k$ are the actual values of these attributes, i.e.:

$$F(o) = \{ o(v_1, \ldots, v_k) \mid v_1 \in s_1, \ldots, v_k \in s_k \}$$

Here $o(v_1, \ldots, v_k)$ is an instance of the type $F(T)$ obtained from $o$ using the described replacement. The type $F(T)$ is obtained from $T$ setting the types of $b_1, \ldots, b_k$ equal to the types $B_1, \ldots, B_k$, respectively.

Notice that for the reduct $R' = F'[e_1, \ldots, e_n]$ where $F' = F(T)$ has no flattening the following definition is valid:

$$\Omega_{R'}(o) = \bigcup_{x \in F(o)} \Omega_R(x)$$

Thus, when a reduct has a flattening the transformation function $\Omega_R$ for an instance $o$ is defined as a union of sets returned by the function $\Omega_R$ for every $F(o)$, the flattening of $o$.

Example. Consider an ADT $T$ defined as

$$\{ T; \text{in} : \text{type}; \\
\quad a : \text{integer}; \\
\quad b : \text{integer}; \\
\quad n : \{ \text{set}; \text{type_of_element} : \{ \text{in} : \text{type}; \text{x : integer}; \text{y : integer}; \}; \}; \}$$

and reducts $T[a, n]$, $T[c : a, n/[x]]$, $T[c : a, n/\{ \text{set}; \text{type_of_element} : [x]\}]$, $T[a, c : n.x]$. Let $o = \langle a = 1, b = 2, n = \langle x = 3, y = 4 \rangle, \langle x = 13, y = 14 \rangle \rangle$. Then:

$$\Omega_{T[a,n]}(o) = \langle a = 1, n = \{ x = 3, y = 4 \}, x = 13, y = 14 \rangle \rangle$$
$$\Omega_{T[c,a,n/[x]]}(o) = \{ \langle c = 1, n = \langle x = 3 \rangle \rangle, \langle c = 1, n = \langle x = 13 \rangle \rangle \}$$
$$\Omega_{T[c,a,n/\{ \text{set}; \text{type_of_element} : [x]\}]}(o) = \langle c = 1, n = \{ x = 3 \}, x = 13 \rangle \rangle$$
$$\Omega_{T[a,c,n.x]}(o) = \{ \langle a = 1, x = 3 \rangle \rangle, \langle a = 1, x = 13 \rangle \}$$

9.3.2 Compound formulae

A collection of ADT values corresponds to an entire filter as a result. To formula $F & C$ represented as a formula $F$ with a condition $C$ related to it a result (a subcollection of the result set $s[F]$ satisfying the condition)
corresponds. An instance type of the resulting collection is a type of instance of the initial collection. Formally the result set \(s[F \& C]\) is defined as

\[
s[F \& C] = \{v \in V_T | v \in s[F] \land C(v)\}
\]

Here \(V_T\) denotes an extent of the type \(T\) – a set of all admissible instances of \(T\).

In formulae with several predicate-collections a result is formed from subcollections of ADT values relevant to conditions of constituent subformulae according to the following rules. First, formulae in which subformulae are related with each other by \&, |, \&^\prime\ connectives are considered.

On interpretation of formulae always when it is required to compare variable values or constants an equality predicate is used. The predicate is defined in a nearest common supertype of types of the given values (an equality predicate is always defined at least in a \(\text{Taval}\) type). In particular, for objects their unique identifiers are compared.

On comparison of ADT values with objects, objects are treated as immutable ADT values. To form a general result of several subformulae related by \&, |, \&^\prime\ connectives the following rules should be used.

A formula \(F_1 \&^j F_2\) denotes a join of collections corresponding to subformulae \(F_1\) and \(F_2\). A type of instances of the resulting collection is a result of a join (subsection 6.2.1) of types of initial collection instances. A temporal assignment of each instance of the join is equal to an intersection of temporal assignments of instances of initial collections. If at the temporal assignment of intersection instance an empty result is obtained then the result of value join is considered to be undefined. Formally the result set \(s[F_1 \&^j F_2]\) is defined as

\[
s[F_1 \&^j F_2] = \{v \in V_{T_1 \mid T_2} | \exists \lambda_1 \in s[F_1], \lambda_2 \in s[F_2] (v \overset{T_1}{\overbrace{\lambda_1}} \land v \overset{T_2}{\overbrace{\lambda_2}})\}
\]

So the result set \(s[F_1 \&^j F_2]\) is a subset of the extent of join type \(T_1 \mid T_2\) such that each element of subset is formed by joining a pair of some elements from result sets \(s[F_1]\) and \(s[F_2]\).

It is essential that if a result of one of the subformulae has an immutable instance type then \& always leads to a creation of a Cartesian product. In this case a result includes pairs of values equal with respect to a nearest common supertype.

A formula \(F_1 \&^i F_2\) denotes an intersection of collections corresponding to subformulae \(F_1\) and \(F_2\). A type of instances of the resulting col-
lection is a result of a meet (subsection 6.2.1) of types of initial collection instances. Formally the result set \( s[F_1 \& i F_2] \) is defined as

\[
s[F_1 \& i F_2] = \{ v \in V_{T_1 \& T_2} \mid \exists \lambda_1 \in s[F_1], \lambda_2 \in s[F_2] (v \models_{T_1 \& T_2} \lambda_1 \land v \models_{T_1 \& T_2} \lambda_2) \}
\]

So the result set \( s[F_1 \& i F_2] \) is a subset of the extent of meet type \( T_1 \& T_2 \) such that each element of subset is formed by intersecting a pair of some elements from result sets \( s[F_1] \) and \( s[F_2] \).

A formula \( F \& f(x_1, \ldots, x_n, y_1, \ldots, y_m) \) denotes an append of the output values of the function \( f \) to the elements of the result set of \( F \) (function append). Here \( f \) is a function; \( x_i \) are actual input parameters of \( f \) – terms, depending on attributes of instance type of result set of \( F \); \( y_1, \ldots, y_m \) are output parameters of \( f \) disjoint with the attributes of the type \( T \) of instances of the result set of \( F \). A type of instances of the resulting collection is a result of a join of type of initial collection instances with a type \( R \) of output parameters of \( f \). Specification of the type \( R \) looks as follows:

\[
\{ R; \text{in} : \text{type}; \text{supertype} : T; y_1 : W_1; \ldots; y_m : W_m; \}
\]

Here \( W_1; \ldots; W_m \) are types of output parameters \( y_1, \ldots, y_m \) respectively. Formally the result set \( s[F \& f(x_1, \ldots, x_n, y_1, \ldots, y_m)] \) is defined as

\[
s[F \& f(x_1, \ldots, x_n, y_1, \ldots, y_m)] = \{ v \in V_R \mid \exists w \in V_T (w \in s[F] \land v \models_T w \land
\]
\[
f(x_1(w), \ldots, x_n(w), v.y_1, \ldots, v.y_m)\}\}
\]

So the result set of a function append is a collection of instances of the type \( R \), every element \( v \) of which is formed by append of some element \( w \) of the result set of \( F \) with the result of applying of the function \( f \) to the element \( w \). This applying includes a substitution of the attribute names in terms \( x_i \) by their actual values taken from \( w \). The result of applying is a tuple of values of output parameters of \( f \).

A formula \( F_1 \mid F_2 \) denotes a union of collections corresponding to subformulae \( F_1 \) and \( F_2 \). A result contains a collection of ADT values that are contained at least in one of resulting collections corresponding to initial subformulae. A type of instances of the resulting collection is a result of a meet (section 6.2.1) of types of initial collection instances. If such meet is empty then the result is undefined. A temporal assignment of each instance of a result of a union is equal to a union of temporal
assignments of instances of initial collections. Formally the result set $s[F_1 \mid F_2]$ is defined as

$$s[F_1 \mid F_2] = \{v \mid v \in s[F_1] \lor v \in s[F_2]\}$$

In the following example it is assumed that instance types of classes student and employee are defined as subtypes of an instance type of the person class.

$$\text{major}(p) = 'history' \ & \ student(p/\text{Student}) \mid (\text{salary}(p) \geq 20000 \ & \ employee(p/\text{Employee}))$$

A formula $F_1 \&^c F_2$ denotes a difference of collections corresponding to initial subformulae. A type of instances of a resulting collection coincides with a type of instances of the first collection. The resulting collection includes instances of the first collection that do not contain equal counterparts in the second one. If such instances are included into the second collection it is necessary to determine an intersection of temporal assignments of initial instances $t = \text{intersect}(T_1, T_2)$. If it is empty then the first instance is included into the resulting collection. Otherwise two options are possible. If $\text{during}(T_1, t)$ is valid then the result is undefined. Otherwise the first instance is included into the resulting collection with a temporal attribute containing a time slice with an initial and terminating moments equal to $\text{start}(T_1), \text{start}(T_2)$ and $\text{fin}(T_2), \text{fin}(T_1)$.

Formally the result set $s[F_1 \&^c F_2]$ is defined as

$$s[F_1 \&^c F_2] = \{v \mid v \in s[F_1] \land v \notin s[F_2]\}$$

To formula with existential quantifier $\exists w/T[a_1, \ldots, a_n](F)$ an instance type $T[b_1, \ldots, b_m]$ of a resulting collection being a reduct of an instance type $T$ of a subformula $F$ contained in the quantifier scope corresponds. The instance type of the resulting collection is such a reduct that join $T[a_1, \ldots, a_n] \mid T[b_1, \ldots, b_m]$ is equal to the type $T$. The resulting collection is a projection of initial collection on the attributes of the reduct type $T[b_1, \ldots, b_m]$, i.e. for every element of resulting collection there exists such element of initial collection that result element is a “part” of initial element. Formally the result set $s[\exists w/T[a_1, \ldots, a_n](F)]$ is defined as

$$s[\exists w/T[a_1, \ldots, a_n](F)] =$$

$$\{v \in V_T[b_1, \ldots, b_m] \mid \exists \lambda \in s[F](v T[b_1, \ldots, b_m] \equiv \lambda)\}$$
Constructs on and as of have the same meaning as given by their definition in the frame language.

9.4 Interpretation of various formulae of the language

9.4.1 Collection ordering

A function order_by defines an ordering relation on a collection that is determined by a formula – an argument of the function. An identifier in an ordering element should refer to an attribute of an ADT instance of the resulting collection of the argument formula. If an order is defined by a specifier, then an ordering is provided by ascending (asc) or by descending (desc) values of collection instance attributes a sequence of which is determined by a formula.

With each element of an ordering list a specifier of an order is related that gives a direction of ordering for every ordering attribute. A sequence of ordering elements in a list defines an order of the collection by their application from major to minor elements.

9.4.2 Formulae with universal quantifiers

A formula with a bounded universal quantifier all appears as:

\[ \text{all } \langle \text{identifier list} \rangle \langle \langle \text{formula} \rangle \rightarrow \langle \text{formula} \rangle \rangle \]

Here as a formula it is possible to use a formula without predicates-collections (denoted as \( \tau \)) or a formula with predicate-collections (denoted as \( \varphi \)). The following variants of formulae with universal quantifiers are considered to be syntactically correct.

Let \( \varphi_1 \) and \( \varphi_2 \) are the formulae having a type \( T_2 \) that corresponds to \( \varphi_2 \) being a subtype of a type \( T_1 \) that corresponds to \( \varphi_1 \). Let \( T_1 \) includes attributes \( a_1, \ldots, a_n \); \( T_2 \) includes attributes \( a_1, \ldots, a_n, b_1, \ldots, b_m \) and \( T_3 \) includes attributes \( b_1, \ldots, b_m \) (type \( T_2 \) is join of types \( T_1 \) and \( T_3 \)). Formulae all \( w/T_1(\varphi_1 \rightarrow \varphi_2) \) and all \( w/T_1(\varphi_2 \rightarrow \varphi_1) \) are syntactically correct. Free variables of the formulae are remaining unbound free variables of \( \varphi_2 \).

Let \( \tau \) is a formula, \( \varphi \) is a formula, type \( T_2 \) mentioned earlier corresponds to \( \varphi \) and \( w.a_1, \ldots, w.a_n \) is a list of free variables of \( \tau \) where \( a_1, \ldots, a_n \) are attributes of the type \( T_1 \). Formulae all \( w/T_1(\varphi \rightarrow \varphi) \) and all \( w/T_1(\varphi \rightarrow \tau) \) are syntactically correct and equivalent. Free variables of the formulae are free variables of \( \varphi \) formula that remain unbound.
A result of a formula \( \text{all } w / T_1(\varphi_1 \rightarrow \varphi_2) \) is a collection defined on the type \( T_3 \). This collection includes such maximal subcollection of a projection of a result of \( \varphi_2 \) formula on a domain of free variables of \( \varphi_2 \) not bounded by a universal quantifier that every instance of this subcollection satisfies the following condition: a subset of values of the result of \( \varphi_2 \) including this instance as a part of each value is such that each instance of a result of \( \varphi_1 \) can be found as a respective part of an instance of this subset.

Formally the result set \( s[\text{all } w / T_1(\varphi_1 \rightarrow \varphi_2)] \) is defined as

\[
s[\text{all } w / T_1(\varphi_1 \rightarrow \varphi_2)] = 
\{ x \in V_{T_3} \mid \forall y \in s[\varphi_1] \exists z \in s[\varphi_2](z T_1 y \land x T_3 z) \}
\]

Example: find a set of suppliers supplying every red part:

\[
\text{all } p/\text{part.inst}(\text{part}(p) \& p.\text{colour} = \textquote{red'}) \rightarrow \\
\text{ex } s/\text{supply.inst}((\text{supplier}(sul) \& \text{supply}(s) \& \text{part}(p)) \& \\
\text{s.sp = sul} \& s.pt = p)
\]

Classes \textit{part}, \textit{supplier} and \textit{supply} are defined as follows:

\[
\{\textit{part}; \text{in: class}; \} \\
\{\textit{supplier}; \text{in: class}; \} \\
\{\textit{supply}; \text{in: class}; \}
\]

\[
\{\textit{pname: string}; \} \\
\{\textit{sname: string}; \} \\
\{\textit{sp: supplier.inst}; \}
\]

\[
\{\textit{colour: string}; \} \\
\{\textit{city: string}; \} \\
\{\textit{pt: part.inst}; \}
\]

A result of a formula \( \text{all } w / T_1(\varphi_2 \rightarrow \varphi_1) \) is a collection defined on the type \( T_3 \). This collection includes such maximal subcollection of a projection of a result of \( \varphi_2 \) formula on a domain of free variables of \( \varphi_2 \) not bounded by a universal quantifier that every instance of this subcollection is included into the resulting collection of \( \varphi_2 \) only together with the values of attributes corresponding to the variables bounded by a quantifier that form instances of the result of \( \varphi_1 \) and is not included into the resulting collection of \( \varphi_2 \) with any other values of bounded variables.

Formally the result set \( s[\text{all } w / T_1(\varphi_2 \rightarrow \varphi_1)] \) is defined as

\[
s[\text{all } w / T_1(\varphi_2 \rightarrow \varphi_1)] = 
\{ x \in V_{T_3} \mid \forall z \in s[\varphi_2](x T_3 z \rightarrow \exists y \in s[\varphi_1](z T_1 y)) \}
\]
A result of a formula \( \text{all } w/T_{1}(\varphi - \tau) \) is a collection defined on a product type specified on attributes - free variables of \( \varphi \). This collection includes such maximal subcollection of a projection of a result of \( \varphi \) formula on a domain of free variables of \( \varphi \) not bounded by a universal quantifier that every instance of this subcollection is included into the resulting collection of \( \varphi \) only together with the values of attributes corresponding to the variables bounded by a quantifier that form instances satisfying \( \tau \) and is not included into the resulting collection of \( \varphi \) with any other values of bounded variables.

Formally the result set \( s[\text{all } w/T_{1}(\varphi - \tau)] \) is defined as

\[
s[\text{all } w/T_{1}(\varphi - \tau)] = \{ x \in \mathcal{V}_{T_{3}} \mid \forall z \in s[\varphi](x T_{2} z \Rightarrow \tau(z)) \}
\]

Alongside with the universally quantified formulae of the kind introduced above formulae \( \text{all } w/T[a_{1}, \ldots, a_{n}](F) \) are allowed in which a domain of \( w \) is not explicitly bounded. Such formulae result in collections defined on a reduct \( T[b_{1}, \ldots, b_{m}] \) of \( F \) instance type \( T \) attributes of which are determined by a set of free variables of \( F \). The instance type of the resulting collection is such a reduct that join \( T[a_{1}, \ldots, a_{n}] \mid T[b_{1}, \ldots, b_{m}] \) is equal to the type \( T \). Instances of such collection are such projections of a result of \( F \) on a domain of variables not bounded by the quantifier that should satisfy \( F \) together with every admissible set of values of bounded variables and for every instance of resulting collection all the admissible sets of bounded variables are the same. Such formulae are allowed in cases when all admissible sets of values of bounded variables can be finitely determined.

Formally the result set \( s[\text{all } w/T[a_{1}, \ldots, a_{n}](F)] \) is defined as

\[
s[\text{all } w/T[a_{1}, \ldots, a_{n}](F)] = \{ v \in \mathcal{V}_{T[b_{1}, \ldots, b_{m}]} \mid \exists v_{0} \in s[F](v T[b_{1}, \ldots, b_{m}] v_{0}) \land \\
\forall v_{1}, v_{2} \in s[F](v T[b_{1}, \ldots, b_{m}] v_{1} \land v T[b_{1}, \ldots, b_{m}] v_{2} \Rightarrow v_{1} T v_{2}) \}
\]

### 9.4.3 Set functions

Set functions (\( \text{min}, \text{max}, \text{count}, \text{total}, \text{average} \)) are allowed in arithmetic expressions that are included into conditions of formulae.
An implicit argument of a function is a collection of values $s$ that is created by calculating of an expression (being an argument of a set function) over a resulting collection of a formula or of one group of a resulting collection of a formula that includes an expression containing a function (or the expression is computed in a scope of the formula). Elements of a value collection $s$ are values of an argument expression obtained over instances of the entire resulting collection or of a separate group.

Let a type of an argument expression value of a function is $t$.

Then the resulting type of a function is determined as follows:

- a type of a result of a function `count` is always integer and does not depend on $t$;
- a type of a result of functions `max` and `min` is $t$;
- a type of a result of functions `total` or `average` is integer, if $t$ is type integer, or is real, if $t$ is type real.

Values of the functions are determined as follows:

- a value of `count` is the number of elements in a collection $s$;
- a value of `min` or of `max` is a minimal (maximal) value in a collection $s$;
- a value `total` is a sum of values of elements of $s$;
- a value of `average` is an average value of elements of $s$.

Collection $s$ is created by calculation over groups of values in case when `group_by` is used in a formula. In other cases $s$ is created by a computation over the entire resulting collection of a formula.

In expressions arbitrary user-defined set functions are allowed. Identifiers of such functions should be denoted by $\$ in expressions.

### 9.4.4 Interpretation of `group_by` formulae

Elements of a grouping list in a formula `group_by` should be included into a set of attributes of a result of a formula – an argument of `group_by`. Consider a formula `group_by([\{a_1, \ldots, a_n\}, < C(c) >, F])`. Let $a_1, \ldots, a_n, b_1, \ldots, b_m$ are attributes of instances of type $T$ of a formula $F$ – an argument of `group_by`.

A result set of the `group_by` formula is defined on a type $Q$ including attributes $a_1, \ldots, a_n$ and an attribute `partition` of type
9.4 Interpretation of various formulae of the language

\{ set; type_of_element : R; \}

Here type \( R \) includes attributes \( b_1, \ldots, b_m \). So the operation \textit{group\_by} transforms a collection of type \( T \) into a collection of type \( Q \) in such a way that attributes \( b_1, \ldots, b_m \) are grouped into one attribute \textit{partition}.

A set of grouping element values is defined as:

\[
\{ v \in V_T[a_1, \ldots, a_n] \mid \exists w \in s[F](v \stackrel{T[a_1, \ldots, a_n]}{=} w) \}
\]

A number of instances in this set defines a number of different groups (partitions). A collection \( S \) defined on a type \( R \) and related to a particular value of \( v \) equal to \( v_0 \) is defined as:

\[
S(v_0) = \{ x \in V_R \mid \exists t \in s[F](x \stackrel{R}{=} t \wedge t \stackrel{T[a_1, \ldots, a_n]}{=} v_0) \}
\]

It is this collection that forms a partition.

In a formula \textit{group\_by} a collection \( S \) defined on a type \( R \) is denoted by an atom-collection. An identifier of a predicate in an atom is an identifier of a collection. As a term in an atom it is required to use a typed variable.

Formally the result set \( s[\text{group\_by}(\{ a_1, \ldots, a_n \}, F)] \) is defined as

\[
s[\text{group\_by}(\{ a_1, \ldots, a_n \}, F)] = \\
\{ v \in V_Q | \exists w \in s[F](v \stackrel{T[a_1, \ldots, a_n]}{=} w) \wedge \}
\]

\[
v.\text{partition} = \{ x \in V_R | \exists t \in s[F](t \stackrel{R}{=} x \wedge t \stackrel{T[a_1, \ldots, a_n]}{=} w) \}
\}

Example: every supplier identifies its supplied parts by unique names (a specification of a class \textit{supply} is given in subsection 9.4.2)

\[
\text{all } d \ (\text{in}(d, \text{group\_by}(\{ \text{sup}, \text{pn} \}), \\
\{ \text{sup}, \text{pn}, p | \text{ex } s/\text{supply}.\text{inst}(\text{in}(s, \text{supply}) \& \\
\text{sup} = s.\text{sp} \& p = s.\text{pt} \& \text{pn} = p.\text{name}) \})\to > \\
\text{count}(d.\text{group}) = 1)
\]

A construction of \textit{group\_by} with an identifier of an attribute-collection is applied if this collection is explicitly referred to in a context of a formula including \textit{group\_by}.

Example: put into correspondence to each mother her teenage children (to one mother children related to different families can belong)

\[
\text{group\_by}(\{ \text{mom} \}, < \text{ch}(p/\text{Q}), >,
\]

\[
\]

\[
\]
Logic formulae representation

\{ \text{mom, child} \mid \text{ex } f/family.\text{inst}(\text{in}(f,\text{family}) \& \text{mom} = f.\text{mother} \& \text{in}(\text{child}, f.\text{children}) \& \text{child.age} < 18) \} \)

Here $Q$ is defined as the following type:

\{ \text{mom} : \text{string}; \text{prt} : \{\text{set}; \text{type_of_element} : \text{child}\} \}

### 9.4.5 Class forming formulae

By a \text{form} \text{class} formula a new class is determined that is the only result of this formula. \text{form} \text{class} can be used as a part of another formulae.

\text{form} \text{class}(< \text{formula} >, < \text{atom} >, < \text{superclass identifier} > [, < \text{specification variable} >])

A resulting class of a formula gets a name given by an atom predicate identifier. A term in an atom (if it is given) is a variable denoting objects – instances of the class. Such explicit denotation can be used for formulae and rules construction. A superclass identifier defines a superclass in a generalization/specialization hierarchy. The class being formed becomes its subclass. A type of an object – an instance of the class – is defined by a type of the result of the formula. If a type of the formula is a collection of immutable ADT values then their transformation into objects is provided. A specification variable gives a frame that contains additional specifications of the resulting class of the formula (if required). Such specification can include, for instance, metaclass names, class attribute specifications, additional instance attribute specifications (e.g., specifications of predicates – consistency constraints).

### 9.5 Parameterized formulae

Parameterized formulae can be made concrete by a substitution of variables (formal parameters) by values of known types, by type (class) names, by formulae and functions (substituting function formal parameters).

Parameterization of formulae provides for a specification of generic formulae, for introducing of variables of function types, for an assignment of a result of the formula concretization to variables of a function type, for usage of functions represented by formulae. Due to that filters or formulae (as a particular case of a set of rules) can be passed as parameters, returned as values, can be used as constituents of abstract values (objects).
9.6 Formulae in rules

Syntactically parameterized formulae are specified analogously to functions:

\[
< \text{formula description} > ::= < \text{frame name} > |
\]
\[
\{ \text{in : function;}
\]
\[
[\text{params : } \{ < \text{formal parameter list} > \}; ]
\]
\[
< \text{formula} >
\]

Everything defined here for formulae remains valid for assertions.

9.6 Formulae in rules

Up to now, guides for a construction of formulae in filters, in rule bodies and in assertions were considered. For logic programs the main mechanism used is a bottom-up iterative program fixpoint evaluation. A guide for admissible program construction is given in Appendix E. A name of a predicate in a left hand side of a rule denotes a predicate-collection defined on an ADT that is implicitly given by a formula in a rule body. The collection is formed by evaluation of this formula.

A term of a predicate corresponds to collection instances that are determined by a rule body. Usually the term is a variable that gets for its values the instances of a resulting collection of the rule body.

A variable used for a term of a predicate in the left hand side of a rule can be typed by a reduct of a type of an instance of a resulting collection of a rule body. Types of unified terms of a head and of a rule body should be compatible: types of terms of a rule body should coincide with types of respective terms of a head or should be their subtypes.

9.7 Functions of creation and deletion of values in a head of a rule

A function of creation of a value in a head of a rule is expressed by atom:

\[
+ < \text{collection name} > (< \text{term} >)
\]

A collection name determines a collection for inclusion of a created value, a term defines a variable that gets a value of a collection instance type. A symbol + in a head indicates a creation of a value.

Example: specification of a function creating an object – a new car.

\[
\text{new\_car} : \{ \text{in : function;}
\]
\[
\text{params : } \{ + \text{sn/s serial\_no}, + \text{man/made\_by}; \}
\]
\[
\{ + \text{car}(c/[s/\text{serial\_no}, m/\text{made\_by}]) : -c.s = \text{sn} \& c.m = \text{man} \}\}
\]
A function of deletion of a value in a head of a rule is expressed by atom:

\[- \langle \text{collection name} \rangle (\langle \text{typed variable} \rangle)\]

A collection name defines a collection from which a value should be deleted. A variable should get values of a collection instance type from a rule body.

Example: specification of a function that leads to a deletion of a car when a destruction term expires.

```plaintext
delete_car : \{ \text{in} : \text{function};
\{ \neg \text{car}(c) : \neg \eq(\text{year}(\text{current}), \text{plus}(\text{year}(\text{destruction_date}(c)), 1)) \& \text{car}(c) \}\}
```

It is essential that usage of a class name as a predicate of a head without \(+\), \(-\) symbols leads to an update of class instances as mutable values.

### 9.8 Collection comprehension

A "collection comprehension" construction is intended for a constructive comprehension of a collection of values of a type for which an implicit specification of components is given by a target list of typed variables. Attributes of a resulting type are defined by names and types of variables of the target list. This construction can be used in the language wherever collections can be used.

\[
\langle \text{collection comprehension} \rangle \ ::= \{ \langle \text{typed variable list} \rangle \ | \ "\" \langle \text{formula} \rangle \}
\]

So, to a collection comprehension formula \( \{ v_1/T_1, \ldots, v_n/T_n \ | \ F \} \) an instance type \( T \) of a resulting collection consisting of attributes \( v_1, \ldots, v_n \) of types \( T_1, \ldots, T_n \) respectively corresponds. Subformula \( F \) should be a formula without predicate-collections or side-effects. Instances of a typed variable list should correspond to free variables of the formula \( F \). A number of values of the type \( T \) in a resulting collection of the collection comprehension is determined by a number of different valuations of variables of the target list relevant to the formula \( F \). An example of collection
comprehension formulae can be found in subsection 9.4.4 and in the next subsection.

9.9 Identification of an object of a class in formulae

In formulae used in functions and assertions defined in specifications of classes it is convenient to have an ability to refer in the formulae to an object of the specified class. A special identifier this is used for such purposes in the language. A formula Φ using this for a class C (Φ(this)) is equivalent to all x/C(Φ).

For example,

```{project; in : class;
instance_section :
{ consortium : { set; type_of_element : companies.inst; };
projcomp : { in : predicate, invariant;
   {{this.consortium = {x | in(this, x.engaged_in)&in(x, companies)}}}
}}}
```

9.10 Functions in views

Views in the SYNTHESIS language are defined by functions. Such functions are specified similarly to functions returning results in expressions. The only their return parameter should be a parameter returns. In a view a function can return as a result a collection of ADT values or a class. For the first case the return parameter should be defined as returns/ < type name > as [materialized] coll where a type name should refer to ADT and instead of coll it is necessary to use collection, set, bag, sequence. For the second case the return parameter should be defined as returns/ < class name > as [materialized] class. A class name should refer to a class given by a class specification or to a class derived by a formula form_class used in this function. In the function a computation of a collection of an ADT values or of a class with instances having a type corresponding to a type given for returns should be realized.

A predicate with the name of a view can be used in all cases when usage of predicate - collections and of predicate - views is allowed. Such predicate should include terms corresponding to input parameters of a function of a view (if such parameters exist). Values to the terms corresponding to input parameters should be assigned. A specification as [materialized]
indicates a necessity to materialize a view (it is required, for instance, when references to view instances from another collections are possible).

Example. Definition of a view over a class `person` with attributes `name`, `age`, `address`.

Case 1. Specification of a view function as an assertion in a metaslot of an attribute in `class`.

```
{adult; in : class;
  metaslot
    ad : { in : function;
      params : returns/adult as_class;
      enforcement : on_access;
      {{form_class(age(c) > 17 & person(c), adult, person)}}}
  end
superclass : person;
}
```

Case 2. Separate specification of a class and of a function of a view creation

```
{adult; in : class;
  metaslot
    a : {
      enforcement : on_access;
      {{ad}}
    }
  end
superclass : person;
}
```

Specification of a view creating function:

```
{ad; in : function;
  params : returns/adult as_class;
  {{assign_to_class(returs, age(c) > 17 & person(c))}}
}
```

Case 3. Dynamic formation of a view by means of `form_class`
9.11 Program control facilities

In the simplest case a program is represented as a list of rules separated by periods.

A function is one of the program control facilities. On a function call an execution of a current program is discontinued and a program of a function is executed. After that a current program is resumed.

Sometimes an explicit control facilities are required that provide for more clean and simple program description.

\[
\text{program} ::= \text{rule list} | \text{block} | \text{program sequence}
\]

\[
\text{block} ::= \text{begin} \text{program} \text{end}
\]

\[
\text{program sequence} ::= \text{program} ; \text{program sequence}
\]

Semicolon indicates a sequential execution of programs. Besides that, it is possible to use rule bodies that appear as:

\[
\text{conditional rule body} ::= \text{if condition then block}
\]

\[
\text{iterative rule body} ::= \text{while condition do block}
\]

Formulae and logical constants are allowed for conditions. Conditional and iterative rule bodies have the following procedural meaning. A conditional rule body leads to only one rule application while evaluating a least fixpoint of a program. An iterative rule body leads to multiple rule applications.

Example. Reduce a tax 10 times.

\[
dec : =
\text{if} \text{true then}
\begin{align*}
\text{get\_tax(tax);} \\
\text{taxn} &= \text{tax} \times 0.1; \\
\text{put\_tax(taxn)} \\
\text{end}.
\end{align*}
\]
Example. Let a set of tuples defining salaries of employees of different departments is specified as:

\[ \text{dsal: } \{ \text{set; type of element: Employee; } \} \]

Here type \textit{Employee} is defined as follows:

\[
\{ \text{Employee; in : type; } \\
\quad \text{dep : string; } \\
\quad \text{name : string; } \\
\quad \text{sal : integer; } \\
\} 
\]

Increase salaries of employees of database department by 10 \% until a salary of Ivanov will not exceed 10000.

\[
\text{iter : } \quad \text{while} \\
\quad \text{ex emp/Employee}(\text{in(emp, dsal) } \& \text{ emp.dep = 'db' } \& \\
\quad \text{ emp.name = 'Ivanov' } \& \text{ emp.sal < 10000 }) \\
\quad \text{do raise('db').} 
\]

\text{raise} is a function providing for an increase of salaries of employees by 10 \%.

10 Information resource manipulation facilities

Information resource manipulation facilities are oriented on embedding into existing programming languages for programming in the information resource environment using such languages.

It is assumed that application program modules are independently compiled. A context of each module is defined independently on the other modules by a collection of imported information resource specification modules. All entities defined directly in such modules are accessible in a program. Here only some of the resource manipulation facilities characteristic for the SYNTHESIS language are presented. A complete set of resource manipulation facilities should be defined in course of the language implementation.

To form a dynamic context (contexts) of a program execution the
following frame manipulation functions are used:

\[
\begin{align*}
\text{create}\_\text{ctx} & (\text{context identifier}, \text{world identifier list}), \\
\text{extend}\_\text{ctx} & (\text{context identifier}, \text{world identifier list}), \\
\text{narrow}\_\text{ctx} & (\text{context identifier}, \text{world identifier list}).
\end{align*}
\]

It is essential that the information resource specification modules are represented as worlds of the frame base. Thus module identifiers are the same as world identifiers.

### 10.1 Variable binding and concretization

A special function is used to bind an identifier with a type. A notion of a <type> is defined in (6.4).

\[
\text{variable declaration} ::= \text{bindvar}(\text{identifier}, \text{type})
\]

Binding of variables with values (objects):

\[
\text{variable binding} ::= \text{assocval}(\text{identifier}, \text{identifier})
\]

The first variable is associated with a value determined by the second variable.

A dynamic concretization function:

\[
\begin{align*}
\text{concretization} & ::= \text{bind}(\text{typed variable identifier}, \\
& \quad \text{parameterized construction identifier}, \\
& \quad \text{actual parameter list}) \\
\text{parameterized construction identifier} & ::= \\
& \quad \text{module identifier} | \text{class identifier} | \\
& \quad \text{function identifier} | \text{type identifier}
\end{align*}
\]

An object can simultaneously belong to several classes. Such possibility is acquired by a function of inclusion of an object into a class (removing it from a class). The first parameter defines an object, the second parameter defines a class.

\[
\begin{align*}
\text{object inclusion into class} & ::= \text{include}(\text{identifier}, \text{identifier}, \\
& \quad [, \text{actual parameter list}]) \\
\text{object removing from class} & ::= \text{exclude}(\text{identifier}, \text{identifier})
\end{align*}
\]
10.2 Iterators

Several functions used as iterators of value collections are distinguished. All iterators have a standard construction (4.6). An iterator itself is the first operator to be executed in an iterative operator sequence. An identifier of the first iterator parameter serves as an identifier of the iterator. Two functions are associated with an iterator: a function of a get kind that is used for obtaining of a next value of a collection and for advancing of iterator and a predicate eoc that is used for determining that a collection is exhausted.

An iterator of a derived collection is oriented on a use of formulae leading to a derivation of new collections. An iterator is used for a browsing of values of the collection.

\[ \text{iterator of derived collection} ::= \text{find}(\text{variable}, \text{formula}) \]

An iterator of a collection indicates a specific collection in a formula and leads to a browsing of values of the collection that are relevant to a formula.

\[ \text{collection iterator} ::= \text{find-collection}(\text{variable}, \text{collection identifier}, \text{formula}) \]

10.3 Arithmetic and logic functions of formulae

Arithmetic functions give numeric values that are functions of collection comprehension formulae. Such functions are allowed in assertions and in expressions of a host language. \text{max}, \text{min}, \text{count}, \text{total}, \text{average} can be applied as arithmetic functions producing values of a type integer (a function \text{count}) and of a type integer or real depending on a variable type in a collection comprehension formula:

\[ \text{arithmetic function} ::= \text{function identifier} (\text{collection comprehension}) \]

\[ \text{function identifier} ::= \text{max} | \text{min} | \text{count} | \text{total} | \text{average} \]

In case of \text{max}, \text{min}, \text{total}, \text{average} a list of typed variables in a collection comprehension formula should contain a single variable of a type integer or real. A result of a function is maximal (\text{max}), minimal (\text{min}), total (\text{total}), average (\text{average}) value of this variable in the resulting collection of a formula or the number of instances in the resulting collection (\text{count}).
A logic function $\text{assert}(<\text{formula}>)$ gives a true value if a formula is closed and is true or if a nonempty collection of sets of free variable values corresponds to the formula. In other cases the function gives false value.

### 10.4 Transaction management

Facilities for a transaction management include primitives starting a transaction ($\text{transbeg}$), aborting a transaction ($\text{abort}$), committing a transaction ($\text{commit}$), preparing a transaction to commit ($\text{prepare}$), making a transaction rollback ($\text{rollback}$).

```
< transaction management primitive > ::=  
  transbeg | abort | commit | prepare | rollback 
< primitive call > ::=  
  < transaction management primitive > (< object variable >)
```

Functions for assertion management provide for enabling / disabling of an assertion when an event associated with the assertion takes place or for making an assertion to act ($\text{fire}$):

```
< assertion management function > ::=  
  < management action > (< assertion identifier >) 
< management action > ::= enable | disable | fire
```

Analogous management actions can be applied to transaction primitives associated with a particular transaction:

```
< primitive management function > ::=  
  < management action > (< primitive call >)
```

Transaction primitive management functions provide for enabling / disabling of a respective primitive to a particular transaction. Such primitives allow for establishing of required sequences of transactions and their actions in multitranaction structures.
A Frame language syntax

\[
\text{< frame > ::= } \{ \text{< frame identifier >} ; \\
\text{[in : < metaclass name list > ; ]} \\
\text{[params : { < formal parameter list > ; ]} \\
\text{[ < temporal assignment >] \\
\text{[ < metaframe >] \\
\text{[ < frame body >]}}
\]
\]

\[
\text{< temporal assignment > ::= (at < constant of a time slice >)} \\
\text{< constant of a moment > ::= < year > # < month > # < date >} \\
\text{< hour >:< minutes >:< seconds > [. < fractions of second >]} | \\
\text{< hour >:< minutes >:< seconds > [. < fractions of second >]} | \\
\text{current | inf | −inf} \\
\text{< constant of interval > ::= interval(< years > # < months >)} | \\
\text{interval(< days > [< hours >:< minutes >:< seconds > [. < fractions of second >])} | \\
\text{interval(< hours >:< minutes >:< seconds > [. < fractions of seconds >]) | interval(inf)} \\
\]

\[
\text{< constant of a time slice > ::=} \\
\text{< constant of initial moment > /} \\
\text{< constant of terminating moment > |} \\
\text{< constant of initial moment > / < empty > |} \\
\text{< empty > / < constant of terminating moment >} \\
\]

\[
\text{< frame body > ::= } [[ \text{< slot >} : [\text{< value list >}; [\text{< metaslot >}}]... \]
\]

\[
\text{< slot > ::= < slot identifier > | < association identifier >} \\
\text{< value list > ::= < entire value > [ < metavalue >] \\
[< entire value > [ < metavalue >]... \\
\]

\[
\text{< entire value > ::= < value > < temporal assignment >} \\
\text{< value > ::= < frame identifier > | < frame >| \\
\{< logic program >\} | \\
\text{< number > | < string > | < constant of a moment >} | \\
\text{< constant of interval > |} \\
\text{< constant of a time slice > | none | < constant >} \\
\]

\[
\text{< frame identifier > ::= < identifier > [(< actual parameter list >)] |} \\
\text{< variable > | string} \\
\]

\[
\text{< slot identifier > ::= < identifier > | < variable >} \\
\text{< variable > ::= < compound identifier > | [ < identifier > ] |} \\
\text{< typed variable >} \\
\]

\[
\text{< association identifier > ::= < identifier > |}
\]
is_a | instance_of | temporal association identifier

< metatime >::=
  metatime < frame identifier >; |
  metatime[< metatime identifier >]; |
  < metatime body >
end

< metaslot >::=
  metaslot < frame identifier >; |
  metaslot[< metaslot identifier >]; |
  < metaslot body >
end

< metavalue >::=
  metavalue < frame identifier >; |
  metavalue[< metavalue identifier >]; |
  < metavalue body >
end

< metatime body >::= [history :< temporal frame list >; ]
< temporal frame list >::= < temporal frame >|
  < temporal frame list >, < temporal frame >
< temporal frame >::= { valid :< constant of a time slice >; |
  belief :< constant of a time slice > }
< metaslot body >::= [history :< temporal frame list >; ]
  [range :< set of values >; ]
  [cardinality :< arithmetic expression >; ]
  [default :< value >; ]
< metavalue body >::= [history :< temporal frame list >; ]
< metatime identifier >::=: < identifier >
< metaslot identifier >::=: < identifier >
< metavalue identifier >::=: < identifier >
< set of values >::=: { < element > [, < element >]... ]
< element >::=: < value > | { < variable > " " < formula > } | < type >
< type >::=: < built − in type > | < abstract type >
< association >::=: { < association identifier >; < association body > } |
< association body >::=: in : association;
  [domain :< set of frames >; ]
  [range :< set of frames >; ]
  [inverse :< association identifier >; ]
  [association_type : ( < range >, < range >); ]
< set of frames >::=: { < element 1 > [, < element 1 >]... }
< element 1 >::=: < frame identifier > | { < formula > }
< range >::=: < lower bound >: < upper bound >
A Frame language syntax

< lower bound >::= < arithmetic expression > | inf
< upper bound >::= < arithmetic expression > | inf

Syntax of the frame manipulation facilities

< define_world >::= define_world(< world identifier > [, < frame list >])
< delete_world >::= delete_world(< world identifier >)
< define_frame >::= define_frame(< frame >
    [< temporal assignment >], < world >, < variable >)
< world >::= < world identifier >
< define_slot >::= define_slot(< frame selector >,
    < variable >,[< temporal assignment >])
< frame selector >::= < formula >| < variable >
< define_value >::= define_value(< frame selector >,
    < variable >, < value > [ < temporal assignment >], < form >)
< form >::= append | overwrite
< assign_value >::= assign(< variable > ( < parameter list > [, < world >]))
< delete_frame >::= delete_frame(< frame selector >)
< delete_slot >::= delete_slot(< frame selector >, < variable >)
< delete_value >::= delete_value(< frame selector >, < variable >,
    < value >)
< context >::= < world identifier >| < class identifier >|
    < context identifier >
< iterator >::= < value iterator >| < frame iterator >
< value iterator >::= find(< variable >, < formula >)
< frame iterator >::= findframe(< variable >, < formula >)
< iterator advancing function >::= getnext(< variable >)
< frame exhausting predicate >::= eoc(< variable >)
< slot_iterator >::= first_slot(< iterator variable >, < variable >)
< slot_analysis_function >::=
    slot_name(< variable >, < iterator variable >) |
    slot_value(< variable >, < iterator variable >) |
    slot_time(< variable >, < iterator variable >)
< slot_value_iterator >::=
    first_value(< iterator variable >, < variable >)
< value_analysis_functions >::=
    value(< variable >, < iterator variable >) |
    time_of_value(< variable >, < iterator variable >)
< formula >::= < subformula >|
    < formula > & < formula >|
    < formula > " " < formula >|
\[
< \text{formula} > \&^\ast < \text{formula} > | ( < \text{formula} > ) | \\
< \text{formula} > \mid \text{on} < \text{temporal selector} > | \\
[\text{as} \_ \text{of} < \text{temporal selector} >] \\
< \text{subformula} > ::= [ < \text{frame condition} > \& ] < \text{atom} > \\
< \text{frame condition} > ::= \\
< \text{frame path expression} > | ( < \text{frame condition} > ) | \\
< \text{frame condition} > \& < < \text{frame condition} > | \\
< \text{frame condition} > " \mid " < < \text{frame condition} > | \\
< \text{frame condition} > \&^\ast < < \text{frame condition} > \\
< \text{temporal selector} > ::= < \text{constant of a moment} > | < \text{constant of a time slice} > | \\
< \text{variable} > | < < \text{temporal function} > | \\
< \text{temporal association identifier} > ( < \text{parameter} > ) \\
< \text{temporal function identifier} > ::= \text{year} | \text{month} | \text{day} | \text{hour} | \\
\text{minute} | \text{second} | \text{fraction} | \text{start} | \text{fin} | \text{intersect} | \\
\text{union} | \text{extend} | \text{sp_convert} \\
< \text{frame name} > ::= < < \text{frame identifier} > | < \text{variable} >
B  Built-in types of the language

Syntactically definitions of built-in types are given by frames. Frame slots define type parameters given by fixed values in a concrete type or by parameters in a generic types. In a parameterized type specification a string `params : {< formal parameter list >}` follows a type name. A set of operations is associated with each built-in type excluding a type none. Generally built-in types are considered to be nonobject. At the same time built-in types `collection, set, bag, sequence, array` and temporal types `time, interval, timeslice` can be declared as object or nonobject. To define a non-object type the definition `in: type, nonobject [, < metatype name list >];` is required. Each object type is subtype of its nonobject twin and therefore inherits properties and behavior from respective immutable type. We do not consider here exception conditions for described type specifications. They can be introduced similarly to ODMG 3.0.

1. Integer type

Type specification

```plaintext
<integer type> ::= {integer:
    [precision : < precision > [: scale : < scale >; base : < base >]]}
```

```plaintext
<precision> ::= < unsigned integer >
< scale > ::= < unsigned integer >
< base > ::= < unsigned integer >
```

Operations

```plaintext
+, − : integer− − > integer   (prefix)
+, −, *, /, mod : integer, integer− − > integer   (infix)
=, >, >=, <=, <, <= : integer, integer− − > Boolean   (infix)
intreal : integer− − > real
intchr : unsignedshort− − > char
```

The following kinds of an integer type are allowed: `short, long, unsigned short, unsigned long`. The type `long` represents a range of values $-2^{31} .. 2^{31} - 1$. `unsigned long` represents a range of values $0 .. 2^{64} - 1$. `short` represents a range of values $-2^{15} .. 2^{15} - 1$. `unsigned short` represents a range of values $0 .. 2^{16} - 1$. We will use further keyword `integer` to denote a integer type if concrete kind of
integer type is not essential. The keywords unsigned integer will de-
note variant of unsigned short or unsigned long. At last in rest cases
we will use the keywords of the concrete integer types.

Operation intreal converts an argument of integer type into a
real type value, and operation intchr converts an argument of
unsigned short type into a char type value. Rest of type integer op-
erations have traditional meaning.

2. Real type

Type specification

< real type >::= {real; [precision :< precision >]}

Operations

+, − : real− > real
+_,−,*,/ : real, real− > real
=, >, >=, <, <=, <>: real, real− > Boolean
log, exp, sqrt, abs : real− > real
realint : real− > integer

The following kinds of a real type are allowed: float and double
with an ordinary and double precision corresponding to a standard

Operation realint converts an argument of real type into a integer
type value. Rest of type real operations have traditional meaning.

3. Boolean type

Type specification

< Boolean type >::= Boolean

Operations

and, or : Boolean, Boolean− > Boolean
not : Boolean− > Boolean

All operations of type Boolean have traditional meaning.
4. Character type

Type specification

\(< \text{character type} > ::= \text{char} \>

Operations

\(=, >, >=, <=, <, <>: \text{char, char} \rightarrow \text{Boolean} \quad \text{(infix)}\)

\(\text{ord} : \text{char} \rightarrow \text{unsigned short}\)

All operations of type \text{char} have traditional meaning.

5. Octet type

Type specification

\(< \text{octet type} > ::= \text{octet} \>

Operations

\(=, >, >=, <=, <, <>: \text{octet, octet} \rightarrow \text{Boolean} \quad \text{(infix)}\)

\(\text{ord} : \text{octet} \rightarrow \text{unsigned short}\)

Octet type is a special character type used for denoting of characters for which no transformation is required on their transfer through communication systems.

All operations of type \text{octet} have a traditional meaning.

6. Character string type

Type specification

\(< \text{character string type} > ::= \{ \text{string;}
\quad \text{length} ::= \text{< unsigned integer >}\}\}

Operations

\(+ : \text{string, string} \rightarrow \text{string}\)

\(\text{in} : \text{string, string} \rightarrow \text{Boolean}\)

\(\text{char_of_string} : \text{string, unsigned integer} \rightarrow \text{char}\)

\(\text{substring} : \text{string, unsigned integer, unsigned integer} \rightarrow \text{string}\)
like : string, string − > Boolean

cardinal : string − > unsigned integer

=, ! =, <, <=, >, >= : string, string − > Boolean

A parameter length gives a maximal length of a character string. Actual string length can differ of the maximal length but cannot exceed it. An absence of a slot length in a type specification denotes a type with an unlimited string length. A value of a character string type is a sequence of any values of character type.

Operation + produces the concatenation of character strings being first and second parameters. Operation in verifies if character string being second parameter is a substring of character string being first parameter. Operation cardinal returns a length of character string being first parameter. Semantics of rest operations are obvious.

7. Bit string type

Type specification

< bit string type >::= {bitstring:
    length : [ < unsigned integer > ]}

Operations

Bit string operations are the same as the operations over character strings excluding char_of_string operation.

A parameter length gives a maximal length of a bit string. Actual string length can differ of the maximal length but cannot exceed it. An absence of a slot length in a type specification denotes a type with an unlimited string length.

8. Enumeration type

Enumeration type provides for a representation of value sets by an explicit enumeration of their elements. On the set a strict order relation is imposed.

Type specification

< enumeration type >::= {enum:
    enum_list : [ < enumeration specification > ]}
< enumeration specification > ::= 
  < enumeration identifier list >|< range type >
< enumeration identifier list > ::= 
  < enumeration identifier > | 
  < enumeration identifier >; < enumeration identifier list >
< enumeration identifier > ::= < identifier >

Operations
There are none.

9. Range type

A range type provides for representation of a subset of consecutive values of a certain type with a strict order relation. A type is specified by indicating of an element type and a range bounds.

Type specification

< range type > ::= {range:
  l_bound : < lower bound >;
  u_bound : < upper bound >;
  type_of_element : < element type >}
< element type > ::= < a name of any type with strict order relation >
< lower bound > ::= < value of < element type > type >
< upper bound > ::= < value of < element type > type >

Operations

first, last : range → > range_element_type
pred, succ : range, range_element_type → > range_element_type

Operations first and last return respective first and last values of given range. Operations pred and succ return previous and following elements of given range relatively to an element specified by second parameter.
10. Collection types

A collection type defines a set of any collections of elements of a given type. There are two collection types in the SYNTHESIS. One of them is of object type, and another collection type is nonobject. The object type is a subtype of the respective nonobject type. Both these types are virtual.

Collection type specification

< collection type >::=
   { collection;  [in : nonobject; ] type_of_element :< type > }

Operations of nonobject collection type

cardinal : collection→ > unsigned integer
is_empty : collection→ > Boolean
is_ordered : collection→ > Boolean
is_duplicated : collection→ > Boolean
is_in : collection, collection_element_type→ > Boolean
is_equal : collection, collection→ > Boolean
copy : collection→ > collection
delete : collection→ > none

Operations of object collection type

Object collection type inherits all operations from its nonobject supertype described above. Additional operations are defined for this type as follows:

insert_elem : collection, collection_element_type→ > Boolean
remove_elem : collection, collection_element_type→ > Boolean

11. Set types

A set type defines a set of any subsets of elements of a given type. There are two set types - nonobject set type and object set type. The latter inherits from nonobject set type. Both set types inherit from respective collection types.
A shorthand notation for set of terms $\{t_1, t_2, \ldots, t_n\}$ appears as 
$\{t_1, t_2, \ldots, t_n\}$. Symbol \{} denotes an empty set.

Set type specification

\[
<\text{set type} > ::= \{\text{set; \ [in: nonobject; ] type of element : } <\text{type} >\}
\]

Operations of nonobject set type

\[
>, >=, <, =, : \ Publisher, Publisher { > } \text{ Boolean} \quad \text{(infix)}
\]

The operation symbols above denote set-theoretical operations $\supset$, $\subseteq$, $\subseteq$ and $\subset$.

Operations of object set type

\[
\text{union, intersect, differ : set, set} { > } \text{ set}
\]

Second parameter of these operations can be an object set type as well as an nonobject set type.

An object set type inherits also operations of an nonobject set type because it is its subtype.

12. Bag types

A bag type defines a set of any subsets of elements of a given type with duplicate elements. There are two bag types in SYNTHESIS language - nonobject bag type and object bag type. The latter inherits from nonobject bag type. Both bag types inherit also from respective collection types.

A shorthand notation for bag of terms $\{t_1, t_2, \ldots, t_n\}$ appears as 
$\{[t_1, t_2, \ldots, t_n]\}$. Symbol \{[]\} denotes here an empty bag.

Bag type specification

\[
<\text{bag type} > ::= \{\text{bag; \ [in: nonobject; ] type of element : } <\text{type} >\}
\]

Operations of nonobject bag type

\[
>, >=, <, =, : \ Publisher, Publisher { > } \text{ Boolean} \quad \text{(infix)}
\]

These operations denote corresponding set-theoretical operations $\supset$, $\subseteq$, $\subseteq$ and $\subset$. 
Operations of object bag type

union, intersect, differ : bag, bag → set

Second parameters of these operations can be an object bag type as well as a nonobject bag type.

An object bag type inherits also operations of nonobject bag type as its subtype.

13. Sequence types

Sequence type provides for representation of any ordered sequences of elements of a given type. There are two sequence types in SYNTHESIS language - nonobject sequence type and object sequence type. The latter inherits from nonobject bag type. Both sequence types inherit also operations of the respective collection type.

A shorthand notation for sequence of terms \( t_1, t_2, ..., t_n \) appears as \([t_1, t_2, ..., t_n]\). Symbol \([\ ]\) denotes an empty sequence.

Sequence type specification

\(<\text{sequence type} >=\)

\{sequence; [in : nonobject; ] type_of_element :< type >

[; length : < unsigned integer >]

Operations of nonobject sequence type

first : sequence → seq_element_type
last : sequence → seq_element_type
elem : sequence, unsigned integer → seq_elem_type
>, <: sequence, sequence → Boolean (infix)
concatenation : sequence, sequence → sequence
make_array : sequence → array

Presence of an optional slot length in specification of nonobject sequence type as well as object type defines a type of a restricted sequence. \(<\text{unsigned integer}>\) defines a maximal length of sequences of a given type. Actual sequence length can differ of the maximal length for the given type but cannot exceed it.

Operations of object sequence type
append: sequence, sequence − > sequence
reverse: sequence− > sequence
insert_after: sequence, seq_element_type, unsigned integer− > sequence
insert_before: sequence, seq_element_type, unsigned integer− > sequence
insert_first: sequence, seq_element_type− > sequence
insert_last: sequence, seq_element_type− > sequence
remove_at: sequence, seq_element_type− > sequence
remove_first: sequence− > sequence
remove_last: sequence− > sequence
replace_at: sequence, seq_element_type, unsigned integer− > sequence

Operation append changes a sequence that is first parameter by addition of second sequence elements to it tile. The operation concatenation in contrast to append operation creates a new instance of sequence.

An object sequence type inherits also operations of a nonobject sequence type as its subtype.

14. Array types

Array types provides for representation of any ordered set of elements of a given type. There are two array types in SYNTHESIS language - nonobject array type and object array type. Last one inherits from nonobject array type. Both array types inherit also operations of the respective collection type.

A shorthand notation for array of terms \( t_1, t_2, ..., t_n \) appears as \(< t_1, t_2, ..., t_n >\). Symbol \(< >\) denotes an empty array.

Array type specification

\(< array type >::= \{ array; [ in : nonobject; ]
   index :< index type list >;
   type_of_element :< type >\}
\(< index type >::=< type identifier >|< range type >|
< enumeration type >
\(< index type list >::= < index type >|
< index type >, < index type list >}
Operations of nonobject array type

\text{lbound, ubound : array, unsigned integer} \rightarrow \text{index type}
\text{elem : array, array\_of\_index} \rightarrow \text{type\_of\_elem}
\text{array\_to\_seq : array} \rightarrow \text{sequence}

Here \text{array\_of\_index} specifies an index value for each dimension of
the considered array. The second parameter of \text{lbound} and \text{ubound}
operations specifies a dimension number under consideration.

Operations of object array type

\text{update : array, type\_of\_element, array\_of\_index} \rightarrow \text{array}
\text{remove\_at : array, array\_of\_index} \rightarrow \text{array}
\text{resize : array, array\_of\_length} \rightarrow \text{array}

An \text{array\_of\_index} specifies here an index value for each dimension
of the considered array. In addition an \text{array\_of\_length} specifies a
length for each dimension of considered array.

15. Union type

A union type provides for representation of set of values that is a
labeled union of a given collection of types. Each value of a union
type can be represented as a pair \text{< type label, value >}.

Type specification

\text{< union type > ::= \{ union; type\_of\_label :< label type >;
< union element list > \}}
\text{< label type > ::=< type of integer >|< character type >|}
\text{< Boolean type >|< enumeration type >| < type identifier >}
\text{< type of integer > ::=< type short >|< type long >|}
\text{< type unsigned integer >}
\text{< union element list > ::=< union element >|}
\text{< union element >; < union element list >}
\text{< union element > ::=}
\text{< label >:< attribute specification >|}
B Built-in types of the language

default :< attribute specification >
< label >::=< occurrence_of_label_type >
< attribute specification >::=< type identifier >| < type declarator >

A label is an occurrence of a type type_of_label. If < label type > is given by a name of a type, then this type should be one of the following built-in types: a type of integer, a character type, a Boolean type or an enumeration type.

Operations

type_of_value : union → string
value : union → type_of_value_defined_by_the_tag

16. Product type

A product type provides for definition of a set of multi-component data values so that each component has its own type.

Type specification

< product type >::= {product;
< component specification list >}
< component specification list >::= < component specification >|
component specification >; < component specification list >
< component specification >::=
< component_identifier >:< component_type >
< component_type >::=< type identifier >

Operations

. : product, string → component_type (infix)
set_component : product, string, component_type → product
get_component : product, string → component_type
clear_component : product, string → product
copy : product → product
delete : product → none
=,<>: product, product → Boolean (infix)
Second parameter of the first four operations is a value of type string that is a product component identifier.

17. Frame type
A value of a frame type is an arbitrary frame.

Type specification

```
< frame type >::= frame
```

Frame operations are defined in section 4.

18. Exception type
An exception type provides for definition of data of a product type that can be returned to a program to denote an exceptional conditions raised during a function execution.

```
< exception type >::= {exception;
    exception_id : {string; }
    [< component specification list >]}
```

An `exception_id` slot gives a name of an exception state. If an exception state has been detected during a function execution then its name can become accessible for a user. By component specifications a structure of a supplementary information can be provided that defines a state of a product type that can become accessible for a user. If this slot is not included into a type specification then the only information given for a user is a state name.

Operations

```
get_exception_id : exception\rightarrow string
get_component : exception,string\rightarrow component_type
```

19. Temporal types
There are three temporal types: type time, type interval and type timeslice. A type time is structural type a value of which can be a date value, a time value or timestamp value. A value of type interval is a duration of time between two time moment values. Finally a
value of type timeslice is a closed interval of time that includes time moment of the interval beginning and time moment of its end.

As it had been mentioned above there are nonobject and object variants for each temporal SYNTHESIS type. Then object types inherit properties and operations from respective nonobject temporal types.

Temporal type specifications

\[
\begin{align*}
\text{< temporal moment type >} &::=  \\
&\{ \text{time}; \ [\text{in : nonobject}; ] \ < \text{temporal qualifier} > \}
\end{align*}
\]

\[
\begin{align*}
\text{< temporal interval type >} &::=  \\
&\{ \text{interval}; \ [\text{in : nonobject}; ] \ < \text{temporal qualifier} > \}
\end{align*}
\]

\[
\begin{align*}
\text{< temporal slice type >} &::=  \\
&\{ \text{timeslice}; \ [\text{in : nonobject}; ] \ < \text{temporal qualifier} > \}
\end{align*}
\]

Temporal qualifier has the following format:

\[
\begin{align*}
\text{< temporal qualifier >} &::= \text{from} : < \text{start field} >; \ \text{to} : < \text{end field} > \\
\text{< start field >} &::= \{ < \text{datetime field} > \\
&\ [\ < \text{interval leading field precision} >] \}
\end{align*}
\]

\[
\begin{align*}
\text{< end field >} &::= \{ < \text{datetime field} > \\
\text{< datetime field >} &::= \text{yy} \ | \ \text{mo} \ | \ \text{dd} \ | \ \text{hh} \ | \ \text{mi} \ | \ \text{ss} \\
&\{ \text{ff} : < \text{fractional seconds precision} > \}
\end{align*}
\]

\[
\begin{align*}
\text{< fractional seconds precision >} &::= < \text{unsigned integer} > \\
\text{< interval leading field precision >} &::= \\
&\text{precision} :< \text{unsigned integer} >
\end{align*}
\]

Temporal type operations

Temporal types of a discrete time moment (time), a discrete time interval (interval), and timeslice (slice) are parameterized (generic) types. Temporal types are treated as ADTs represented by interface operations. A collection of interface operations for temporal types is decomposed into groups with functionally homogeneous operations included into each of the groups:

- arithmetic operations;
- transformation operations;
- time predicates.
< arithmetic operation identifier > ::= add | minus | ti_minus | tt_minus | mult |
        div | ii_div | mod | intersect | union |
        next | previous | add_days | minus_day | minus_date
< transformation operation identifier > ::= year | month | day | day_of_week |
        hour | minute | second | fraction |
        length | start | fin | current | extend | i_convert |
        si_convert | n_interval | sp_convert |
        tz_hour | tz_minute | month_of_year
< time predicate identifier > ::= equals | after | ge | before | le | ne |
        after_ts | before_ts | during | starts | finishes |
        leap_year | between | eq | gt | lt | is_zero |
        starts | finishes | before_starts | before_ends |
        meets | overlaps

Arithmetic operation semantics will be considered first.

Arithmetic operations denote: add, add_days for a temporal addition, minus, minus_days, minus_date and ti_minus for a temporal subtraction, mult for a temporal multiplication, div and ii_div for a temporal division, mod for a residue by modulo. Finally operations intersect and union denote intersection and union operations for time intervals.

If the first argument in operations of time addition or subtraction is a value of the time type and the second argument is a value of the interval type then the following rules are valid:

- if x is the first argument with a precision a to b then x is substituted by extend(x, yy to b);
- if the second argument of an interval type contains the field mo then the first argument cannot contain the field dd;
- a value of these operations is a value of the time type that cannot contain fields not included into the first argument.
If both arguments in operations of time addition or subtraction have an interval type then these operations should conform to the following rules:

- if one of the arguments contain fields yy or mo then the second argument should also contain fields yy or mo;
- if one of the arguments does not contain fields yy or mo then the second argument cannot contain fields yy or mo;
- a value of these operations is a value of the interval type that cannot contain fields that are not included into the arguments of an operation.

If both arguments in an operation minus has the time type, then operation minus is constrained as follows: an operation should produce a value of the interval type that cannot contain fields mo and dd simultaneously.

Operations mult and div having an interval and an unsigned integer types for the arguments are constrained as follows: the operations should produce a value of an interval type that does not contain fields mo and dd simultaneously.

It worth noting that an arithmetic operation produces an undefined value if a value of at least of one of its arguments is not defined.

An operation length determines a length of a time slice given for the operation argument. The operation length produces a value of the interval type that does not contain mo and dd fields simultaneously.

Transformation operation semantics will be given further. Operations year, month, day, hour, minute, second and fraction produce unsigned integer for their value if their argument is a value of the time type. If their argument has the interval type then these operations produce an integer value.

Operations tz_hour and tz_minute provide current time value for local time zone: an unsigned integer value for hour and an unsigned integer value for minutes.

An argument of an operation day_of_week has the time type and character string is a type of its value.

A current operation provides an access to a computer timer. A value of current is a current discrete time moment with a precision yy to ff(3).
If a temporal qualifier is used as an argument of \texttt{current} then its value is a discrete time moment with a collection of fields defined by the temporal qualifier - the operation argument.

If an argument of an \texttt{extend} operation has a type of a discrete time moment then its value is the same discrete time moment with a precision \texttt{yy to ff(3)} . If arguments of an \texttt{extend} operation are a discrete time moment and a temporal qualifier then its value is the same discrete time moment with a precision implied by the temporal qualifier - the second argument of the operation. An \texttt{extend} operation should satisfy the following constraints:

- if the second argument implies fields to the left of those contained within fields implied by the first argument of an operation then these additional fields are initialized with values from \texttt{current};
- if the first argument of an operation contains fields not included within the precision specified by a temporal qualifier then these fields are truncated;
- if an argument - temporal qualifier contains fields to the right of those contained within values defined by the first argument of the operation then these additional fields are initialized with the value 0 for fields \texttt{hh, mi, ss} and \texttt{ff (n)} and the value 1 for the fields \texttt{yy, mo, dd}.

An argument of a \texttt{i_convert} operation has an interval type. A value of a \texttt{i_convert} operation is an integer expressed in units of an end field of a temporal qualifier argument.

A \texttt{sp_convert} operation provides for a transformation of its first argument of an \texttt{integer} type into a value of the \texttt{time} type in accordance with a precision defined by a temporal qualifier given by the second argument of the operation. Start and end fields in a temporal qualifier contain the same values.

A \texttt{si_convert} operation is analogous to a \texttt{sp_convert} for a transformation of integer values into values of an interval type.

An operation \texttt{n_interval} produces a value of an operation argument (a value of an interval type) with the minus sign.

Both arguments of \texttt{union} and \texttt{intersect} operations are time slices. These operations produce as their value a time slice that is a union or an intersection of the argument time slices.
Operations start and fin produce a value of a time type; their argument type is a timeslice. An operation start determines a start of a time slice and an operation fin determines an end of a time slice.

The transformation operations defined above produce undefined values if one of their arguments is undefined.

Definition of operations for temporal types follow.

Operations of nonobject time type

\[
\begin{align*}
\text{year, month, day, hour, minute, second,} \\
\text{fraction : } \text{time} & \rightarrow \text{unsigned integer} \\
\text{tz\_hour : } & \rightarrow \text{unsigned integer} \\
\text{tz\_minute : } & \rightarrow \text{unsigned integer} \\
\text{day\_of\_week : } & \text{time} \rightarrow \text{string} \\
\text{month\_of\_year : } & \text{time} \rightarrow \text{string} \\
\text{next, previous : } & \text{time} \rightarrow \text{string} \\
\text{equals, after, ge, before, le, ne : } & \text{time, time} \rightarrow \text{Boolean} \\
\text{after\_ts, before\_ts, during : } & \text{time, timeslice} \rightarrow \text{Boolean} \\
\text{starts, finishes : } & \text{time, timeslice} \rightarrow \text{Boolean} \\
\text{current : } & \text{[string]} \rightarrow \text{time} \\
\text{day\_of\_year : } & \text{time} \rightarrow \text{unsigned integer} \\
\text{leap\_year : } & \text{time} \rightarrow \text{Boolean} \\
\text{between : } & \text{time, time, time} \rightarrow \text{Boolean}
\end{align*}
\]

Operations of object time type

\[
\begin{align*}
\text{add\_days, minus\_days : } & \text{time, unsigned integer} \rightarrow \text{time} \\
\text{minus\_date : } & \text{time, time} \rightarrow \text{unsigned integer} \\
\text{next, previous : } & \text{time} \rightarrow \text{time} \\
\text{add : } & \text{time, interval} \rightarrow \text{time} \\
\text{ti\_minus : } & \text{time, interval} \rightarrow \text{time} \\
\text{tt\_minus : } & \text{time, time} \rightarrow \text{interval} \\
\text{extend : } & \text{time, [string]} \rightarrow \text{time} \\
\text{sp\_convert : } & \text{integer, string} \rightarrow \text{time}
\end{align*}
\]
Operations of nonobject interval type

year, month, day, hour, minute, second,
    fraction : interval→> unsigned integer
i_convert : interval, string→> unsigned integer
si_convert : unsigned integer, string→> interval
n_interval : interval→> interval
eq, gt, ge, lt, le : interval, interval→> Boolean
is_zero : interval→> Boolean

Operations of object interval type

add, minus : interval, interval→> interval
mult, div : interval, unsigned integer→> float
mod, ii_div : interval, interval→> float

Operations of nonobject timeslice type

length : timeslice→> interval
start, fin : timeslice→> time
before, equals, after, during : timeslice, timeslice→> Boolean
starts, finishes : timeslice, timeslice→> Boolean
before_starts, before_ends : timeslice, timeslice→> Boolean
meets, overlaps : timeslice, timeslice→> Boolean

Operations of object timeslice type

intersect, union : timeslice, timeslice→> timeslice

20. Representation of constants and multiterms

Constants of types having complex structures are represented as frames of the kind:

< constant >::=< component >| {constant_spec: < component list >}
< component list >::=< component >|
    < component >; < component list >
< component >::=< nonconstant value > [/ < type designator >] |
< attribute name > : < nonconstant value >
[: metaslot
   < type designator >
end]
< type designator >::=< type name > | < attribute name > |
   < reduct >|< simple built in type >
< reduct >::=< type name > "[" < attribute name list > "]"
< attribute name list >::=< attribute name > |
   < attribute name >, < attribute name list >
< attribute name >::=< attribute identifier > |
   < attribute name > . < attribute identifier > |
   < attribute name > "[" < arithmetic expression > "]"|
< attribute identifier >::=< identifier >
< multiterm >::={multiterm_spec; < multiterm component list >}
< multiterm component list >::=< multiterm component > |
   < multiterm component >; < multiterm component list >
< multiterm component >::=< component > |
   < typed variable >|< multiterm > |
   < attribute name >:< typed variable > |
   < attribute name >:< multiterm >

Constants of enumeration type, set type, array type or constants of
user defined ADTs obtain the representation defined. Component
values in their turn can be constants of types with a complex struc-
ture. For instance, arrays can be components of multidimensional
arrays (see definition < attribute name > above).
Type designator for a component is required in case when the com-
ponent can acquire alternative types. Usually a type of a constant
can be inferred from a context of its usage. If this is not possible
then an explicit type designator should be used.

21. None type
A none type is a single instance type with empty set of attributes
and empty set of operations.
Type specification

< none type > ::= none
C   Built-in assertions

C.1   Simple assertions

• An assertion of uniqueness.

\[
<\text{assertion of uniqueness}> ::= \{\text{unique;}
\]
\[
[\{<\text{attribute list}>\}]\}
\]

\[
<\text{attribute list}> ::= <\text{attribute identifier}> |
\]
\[
<\text{attribute identifier}>, <\text{attribute list}>
\]

An assertion of uniqueness of values of a given attributes defines a list of attributes a set of values of which uniquely identifies an object among other instances of a class. In this assertion as well as in other built-in assertions an attribute list can be omitted if an assertion corresponds to a single attribute or to a single attribute category.

• An assertion of constancy.

\[
<\text{assertion of constancy}> ::= \{\text{constant;}
\]
\[
[\{<\text{attribute list}>\}]\}
\]

An assertion of constancy provides for declaration of values of given attributes to be unmodifiable during the life cycle of an object containing the attributes. This assertion should be valid for all objects of a class to which it is related.

• An assertion of definiteness.

\[
<\text{assertion of definiteness}> ::= \{\text{obligatory;}
\]
\[
[\{<\text{attribute list}>\}]\}
\]

An assertion of definiteness is used to declare that values of attributes listed should be obligatory defined (not equal to none) in all objects of a class to which this assertion is related.

• An assertion of conditional uniqueness.

\[
<\text{assertion of conditional uniqueness}> ::= \{\text{unique nonnull;}
\]
\[
[\{<\text{attribute list}>], [\{<\text{attribute list}>]\}... ]\}
\]

An assertion of conditional uniqueness of the given collections of attributes defines for a class to which it is related a set (sets) of attributes each of which is listed in figure brackets and possesses the following property: if values of all attributes in a set are not equal
to none then this set of values uniquely identifies an object of a class among its other objects. If several sets of attributes simultaneously have nonnull values then they should identify one and the same object.

- An assertion of conditional constancy.

\[
<\text{assertion of conditional constancy}> ::= \{\text{constant nonnull;}
\]
\[
[[\{\langle\text{attribute list}\rangle\}, \{\langle\text{attribute list}\rangle\}]]
\]

An assertion of conditional constancy declares that the values of given attributes of an object should be unmodifiable while an object is an instance of a class to which the assertion is related. The assertion becomes effective after all the values of given attributes listed in figure brackets get nonnull values.

- An assertion - function.

\[
<\text{assertion}\text{−}\text{function}> ::= \{\text{func;}
\]
\[
\{\langle\text{attribute list}\rangle\}, \{\langle\text{attribute list}\rangle\}
\]

An assertion - function expresses for a class C to which it is related the fact that there exists in C a functional dependency of an attribute collection B (given by the second attribute list) on an attribute collection A (given by the first attribute list). This means that at any moment to each tuple value of a projection of C on a collection of attributes A, in which values of all components have nonnull values, the one and only one tuple value (with nonnull component values) of a projection of C on a collection of attributes B corresponds that is included together with the first projection tuple into some object of C.

- An assertion - partial function.

\[
<\text{assertion}\text{−}\text{partial function}> ::= \{\text{pfunc;}
\]
\[
\{\langle\text{attribute list}\rangle\}, \{\langle\text{attribute list}\rangle\}
\]

An assertion - partial function expresses for a class C to which it is related the fact that there exists in C a partial functional dependency of an attribute collection B (given by the second attribute list) on an attribute collection A (given by the first attribute list). This means that at any moment to each tuple value of a projection of C on a collection of attributes A, in which values of all components have nonnull values, not more than one tuple value (with nonnull component values) of a projection of C on a collection of attributes B
corresponds that is included together with the first projection tuple into some object of C.

- An assertion of order.

\[
\langle \text{assertion of order} \rangle ::= \{ \text{ordered}; \ \langle \text{order} \rangle, \\
\quad \text{[restricted_by:}\ \langle \text{attribute list} \rangle; \ ] \ \{ \langle \text{ordering list} \rangle \} \}
\]

\[
\langle \text{ordering list} \rangle ::= \langle \text{ordering element} \rangle |
\quad \langle \text{ordering element} \rangle, \langle \text{ordering list} \rangle
\]

\[
\langle \text{ordering element} \rangle ::= \{ \langle \text{direction} \rangle; \ \{ \langle \text{attribute list} \rangle \} \}
\]

\[
\langle \text{direction} \rangle ::= \text{ascending} \ |
\quad \text{descending}
\]

\[
\langle \text{order} \rangle ::= \text{chronologically} \ |
\quad \text{chronologically backward}
\]

An assertion of order provides for considering of a class as a linearly or partially ordered set of objects. If a construction restricted_by is not given then an assertion of order states that a class is a linearly ordered set. If a construction restricted_by is given then an assertion of order defines a class as a partially ordered set of objects. In such set a linear order is established on a subset of a class that includes objects having equal values of attributes given in a list of attributes of a restricted_by construction. It is essential that for a class not more than one linear order assertion and an arbitrary number of partial order assertions can be specified.

A construction \( \langle \text{direction} \rangle \{ \langle \text{attribute list} \rangle \} \) defines a set of attributes listed in an order from major to minor ordering elements. Objects of class are ordered by values of the attributes from this set. A direction prescribes that an ordering is by ascending or by descending attribute values. A construction chronologically and chronologically backward prescribes an order that is determined by moments of object creation. If only a construction \( \langle \text{order} \rangle \) or an attribute list is given then an order is established only chronologically or only by values of given attributes. If both constructs are given simultaneously then those comparable objects of a class that have equal values of ordering attributes are ordered chronologically with respect to each other.

C.2 Functional assertions

A set of built-in assertions includes various functional assertions defining functional (referential) dependencies on classes (or on collections defined on product types).
C.2 Functional assertions

For definitions of functional assertions the following notation is used: 
\(C[A]\) denotes a projection of a class on an attribute set \(A\), \(c[A]\) denotes a projection of an instance of a class \(C\) on an attribute set \(A\).

A class \(C_i\) is functionally dependent on a class \(C_j\) on a nonredundant sets of attributes \(A_i, A_j\) if \(A_i\) defines a set of attributes of \(C_i\) satisfying an assertion of uniqueness for \(C_i\). \(A_j\) defines attributes of \(C_j\) such that to each \(c_j[A_j]\) from \(C_j[A_j]\) with nonnull values of all \(A_j\) the one and only one object \(c_i\) belonging to \(C_i\) corresponds. This object is such that \(c_j[A_j] = c_i[A_i]\). Numbers of attributes in attribute lists \(A_j\) and \(A_i\) should be the same, attributes in the respective positions of the lists shoud have compatible types.

A functional dependency of a class \(C_i\) on a class \(C_j\) on sets of attributes \(A_i, A_j\) is \emph{total}, if for each object \(c_j\) of \(C_j\) there always exists the one and only one object \(c_i\) of \(C_i\) such that \(c_j[A_j] = c_i[A_i]\).

A functional dependency of a class \(C_i\) on a class \(C_j\) on sets of attributes \(A_i, A_j\) is \emph{weak partial} (or simply \emph{partial}) if for each object \(c_j\) of \(C_j\) the one and only one object \(c_i\) of \(C_i\) can be put into correspondence such that \(c_j[A_j] = c_i[A_i]\) in case when all attributes \(A_j\) in \(c_j\) have nonnull values.

A class dependency expressed by an assertion of a \emph{strong partial} functional dependency is analogous to a weak partial functional dependency. The difference of a strong partial dependency consists in its dynamics: after in an object \(c_j\) of \(C_j\)nonnull values of all attributes of \(A_j\) were set (it means that a relationship of \(c_j\) with the only object \(c_i\) of \(C_i\) has been established), this relationship cannot be removed until objects \(c_i\) or \(c_j\) will not be deleted.

A partial functional dependency with an \emph{initial linkage} is the same as a weak partial functional dependency with the difference that on creation of an object \(c_j\) all its attributes \(A_j\) should have nonnull values.

A total or a partial functional dependencies with a \emph{feedback} differs on the basic dependency (total or partial) by an existence of a feedback. A feedback acts so that after deletion of an object \(c_j\) from \(C_j\) such that \(c_j[A_j] = c_i[A_i]\) if no object \(c'_j\) of \(C_j\) remains such that \(c'_j[A_j] = c_i[A_i]\) then an object \(c_i\) will be also deleted.

Syntactically functional assertions for classes are defined as follows:

\[
<\text{functional assertions}> ::= \\
<\text{assertion of total functional dependency}>| \\
<\text{assertion of partial functional dependency}>
\]

\[
<\text{assertion of total functional dependency}> ::= \\
\{\text{total}; \\
\text{dep_type } ' <\text{total f.d. symbol}>; \}
\]
Built-in assertions

\[
\text{domain} : \{ \langle \text{extension kind} > : \langle \text{identifier of class/collection} > ; \\
\quad \text{attr} : \{ \langle \text{attribute list} > \} \}; \\
\text{range} : \{ \langle \text{extension kind} > : \langle \text{identifier of class/collection} > ; \\
\quad \text{attr} : \{ \langle \text{attribute list} > \} \}; \\
\}
\]

\[
\text{assertion of partial functional dependency} ::= \\
\{ \text{partial} ; \\
\quad \text{dep\_type} : \langle \text{partial f.d. symbol} > ; \\
\quad \text{domain} : \{ \langle \text{extension kind} > : \langle \text{identifier of class/collection} > ; \\
\quad \quad \text{attr} : \{ \langle \text{attribute list} > \} \}; \\
\quad \text{range} : \{ \langle \text{extension kind} > : \langle \text{identifier of class/collection} > ; \\
\quad \quad \text{attr} : \{ \langle \text{attribute list} > \} \}; \\
\}
\]

Constructions domain and range are excluded when a functional assertion is related to an attribute of an object of the kind: a: C, where C is a class name. A type of dependency is assumed by default for a total f.d. to be ‘\(- >\)’, for a partial dependency ‘\(-' >=\)’.

\[
\text{total f.d. symbol} ::= \langle \text{basic t.f.d. symbol} > | \\
\quad \langle \text{modified t.f.d. symbol} > \\
\text{partial f.d. symbol} ::= \langle \text{basic p.f.d. symbol} > | \\
\quad \langle \text{modified p.f.d. symbol} > \\
\text{basic t.f.d. symbol} ::= \text{- } > \\
\text{basic p.f.d. symbol} ::= \text{=} > \\
\text{modified t.f.d. symbol} ::= \text{- } < \text{t.f.d. modifier } \text{- } > \\
\text{modified p.f.d. symbol} ::= \text{=} = < \text{p.f.d. modifier } \text{=} > \\
\text{p.f.d. modifier} ::= \langle \text{p.f.d. kind} > [ \langle \text{p.f.d. kind} > ]... \\
\text{t.f.d. modifier} ::= \langle \text{p.f.d. kind} > \\
\text{p.f.d. kind} ::= s | i | i | \text{empty} > \\
\text{t.f.d. kind} ::= i | \text{empty} >
\]

Different kinds of functional dependencies are denoted as s for a strong partial f.d., l for a partial f.d. with an initial linkage, i for a f.d. with a feedback.
Examples of mediator specifications in the SYNTHESIS language

D.1 Personnel management application domain

For a mediator specification it is convenient to use the SYNTHESIS language in a declarative style. This style is characterized by using of type, class and metaclass specifications, using of rules, assertions and temporal facilities, using of predicative specifications of functions by means of mixed pre- and post-conditions, associating of assertions with types and their attributes.

A simple example of a mediator in a personnel management application domain follows.

```synthesis
{personnel
example; in: module, mediator;
class_specification:
    {entity
class; in: metaclass; }
    {activity
class; in: metaclass; }
instance_section:
    {input: entity
class; metaslot in: association end
output: entity
class; metaslot in: association end
pred: assertion;
metaslot
    in: association;
def
pre : {in: predicate, invariant;
    {overlaps(when
holds(this.pred.pre)), this.valid}};
def
post : {in: predicate, invariant;
    {meets(this.valid, when
holds(this.pred.post))}};
end
}
{comment; In the SYNTHESIS language function specifications based on mixed pre- and post-conditions are used. In this example pred.pre and pred.post denotes the respective parts of specifications. The function when.holds is assumed to be built-in. The function produces a time slice during which an assertion (an argument of the function) is true.
}
{project; in: entity
class; }
{company; in: entity
class;
instance_section:
    {engaged
in: {set; type
of
element: project
inst;}}
}
```
{person; in: entity_class;
  instance_section:
  {name: string;
   metaslot
     a: invariant, {{obligatory;}}
   end}
  }
{employee; in: entity_class;
  superclass: person;
  instance_section:
  {belongs_to: company;
   metaslot
     b: invariant, {{obligatory;}}
   end}
  }
{proj_worker;
  in: entity_class;
  superclass: employee;
  instance_section:
  {works_on: {set; type_of_element: project.inst};
   metaslot
     c: invariant, {{obligatory;}}
   end
  work_inv: invariant, {{this.works_on <= this.belongs_to.engaged_in}}}
  }
{hire_emp; in: activity_class;
  instance_section:
  {p: person;
   metaslot
     in: input
   end
  belongs: company;
  metaslot
     in: input
   end
  e: employee;
  metaslot
     in: output
   end
  assert: pred, {{e' = p & e'.belongs_to = belongs}}
  },
D.2 Research companies application domain

Important features of the SYNTHESIS language used in this example include:

\[
\{ \text{hiring employee for project; } \\
\text{in: activity_class; } \\
\text{instance_section: } \\
\text{works: \{set; type_of_element: project.inst\}; } \\
\text{metaslot } \\
\text{in: input; } \\
\text{d: \{\{obligatory\}\} } \\
\text{end } \\
\text{e: proj worker; } \\
\text{metaslot } \\
\text{in: output } \\
\text{assert: pred, \{\{e'.works_on = works\}\} } \\
\}\},
\{ \text{assign employee to project; } \\
\text{in: activity_class; } \\
\text{instance_section: } \\
\text{e: employee; } \\
\text{metaslot } \\
\text{in: input; } \\
\text{f: \{\{obligatory\}\} } \\
\text{end } \\
\text{works: \{set; type_of_element: project.inst\}; } \\
\text{metaslot } \\
\text{in: input; } \\
\text{g: \{\{obligatory\}\} } \\
\text{end } \\
\text{pw: proj worker; } \\
\text{metaslot } \\
\text{in: output } \\
\text{assert: pred, \{\{intersect(e.works_on, works) = \{} & } \\
\text{pw' = e & pw'.works_on = works\}\} } \\
\}\}
Examples of specifications in the SYNTHESIS language

- using of type (class) hierarchies convenient for representation of an information system structure;
- using of transaction classes and entity classes to describe a system being designed;
- preserving of predicative specifications style for a definition of assertions and transactions;
- introducing of attribute categories providing for indication of a role of assertions related to attributes of types (classes).

```d
{research_companies;
in: module, mediator;
type: {agencies: {enum; enumlist: {'ESPRIT', 'NSF', ...}}};
class_specification:
{companies;
in: class;
instance_section:
{name: string;
metaslot
   a: invariant, {{unique;}};
   b: invariant, {{constant;}}
end;
engaged_in: {set; type_of_element: projects.inst;}
}},
{employees;
in: class;
instance_section:
{name: string;
metaslot
   c: invariant, {{constant;}}
end;
belongs_to: companies;
works_on: {set; type_of_element: projects.inst;};
on_emp_proj: {in: predicate, invariant;
   {all e (works_on(e) <= belongs_to(e).engaged_in & employees(e))}}
}},
{projects;
in: class;
instance_section:
```
E Rules and recommendations for admissible program construction

E.1 General recommendations

Let \( p \) and \( q \) are predicates. It is said that a predicate \( p \) derives \( q \) (this fact is denoted as \( p \rightarrow q \)) if the predicate occurs in a body of a rule in which
A symbol $\rightarrow +$ denotes a transitive closure of $\rightarrow$.

A predicate $p$ is **recursive** if $p \rightarrow +p$. Two predicates $p$ and $q$ are **mutually recursive** if $p \rightarrow +q$ and $q \rightarrow +p$.

A recursive rule is **linear** if in its body there exists the only predicate mutually recursive with a predicate in a head of the rule. A set of rules is linear if every recursive rule in it is linear.

An example of a linear program (relatives of the same generation). It is assumed below that on an interface of a type of instances of a set parent the operations name and $\text{name}$ are defined giving a person name and its parent name respectively.

\[
\begin{align*}
\text{sg}(s[x, y]) : & \quad \text{ex } p, p_1, x_1, y_1 (\text{parent}(p) & x = p.\text{name} & x_1 = p_1.\text{name} & \\
& \text{sg}(s_1[x_1, y_1]) & \text{parent}(p_1) & y_1 = p_1.\text{name} & y = p_1.\text{pname}).
\end{align*}
\]

\[
\begin{align*}
\text{sg}(s[x, x]) : & \quad \text{ex } p(\text{parent}(p) & x = p.\text{name}).
\end{align*}
\]

An example of nonlinear program (a blue blooded Frenchman)

\[
\begin{align*}
\text{bbf}(x) : & \quad \text{ex } p_1, p_2, f, m(\text{parent}(p_1) & x = p_1.\text{name} & \\
& f = p_1.\text{pname} & \text{parent}(p_2) & x = p_2.\text{name} & \\
& m = p_2.\text{pname} & \text{bbf}(f) & \text{bbf}(m)).
\end{align*}
\]

\[
\begin{align*}
\text{bbf}(x) : & \quad \text{ex } p(\text{bb_person}(p) & x = p.\text{name}).
\end{align*}
\]

A general recommendation for logic programming is to try to reduce programs to a linear form (this is not always possible).

A query $q$ to a database $D$ is **safe** if a response to $q$ is finite.

Among sources of unsafety are the following: computed predicates that are interpreted on infinite sets (for instance, $20 > x$); occurrences of variables in heads of rules that do not occur in rule bodies (for instance, $\text{likes}(l[x, y]):- \text{nice}(x)$).

A rule is **domain-restricted** if every variable in a head occurs in a rule body. A set of rules is domain-restricted if each rule in the set is domain-restricted.

A rule is **strictly safe** if

1. it is domain restricted;
2. every variable in a computed predicate term occurs also at least in one predicate of extensional database.

A set of rules is strictly safe if every rule in this set is strictly safe.
Any query defined on a set of strictly safe rules is safe. It is necessary to define safe logic programs.

A variable in a rule body is protected if it occurs in a noncomputed predicate in a rule body or if it occurs in a position $i$ of computed predicate $p$ and there exists a subset $I$ of variables $p$ that are protected and there exists a functional relationship $I \to \{i\}$. This definition is a recursive one.

**Examples.**

It is assumed that sets with instances defined on product types correspond to predicates $p$ and $q$.

$p(pe[x,y]) : \neg \exists y1(x > y1 \& q(qe[y1,y]))$

A variable $x$ here is not protected.

$p(pe[x,y]) : \neg \exists y1, y2(p(pe1[y1,y2]) \& y = x + 1 \& x = y1 + y2)$

Here $\{y1, y2\} \rightarrow \{x\}, x$ is protected, $\{x\} \rightarrow \{y\}, y$ is protected.

A rule is bottom-up computable if

1. it is domain restricted;

2. every variable in the rule is protected.

A set of rules is bottom-up computable if every rule in the set is bottom-up computable.

In an extensional database facts contain only ground terms (without variables).

A general rule of proper order of computation consists in the following: such order of computation should be found in which an execution of computable predicates is postponed until all their input arguments will obtain values.

### E.2 Admissible logic programs in the SYNTHESIS language

In the beginning admissible programs with negation are considered. A closed world assumption is a basic consideration: if a fact is not logically deducible of the context then its negation is true. Negative facts can be deduced but they cannot be used to deduce another facts. Therefore a program with negation should be stratifiable. Computation of a rule with one or several subformulae in a rule body is realized by evaluation

---

E.2 *Admissible logic programs in the SYNTHESIS language*  

In the beginning admissible programs with negation are considered. A closed world assumption is a basic consideration: if a fact is not logically deducible of the context then its negation is true. Negative facts can be deduced but they cannot be used to deduce another facts. Therefore a program with negation should be stratifiable. Computation of a rule with one or several subformulae in a rule body is realized by evaluation

---

1. This subsection is written together with V.I.Zadorozhny
of predicates corresponding to such negative subformulae. A closed world
assumption is applied locally to such subformulae. It is required that
before evaluation of a predicate in a head of a rule it should be possible to
evaluate negative subformulae in a rule body as well as all predicates that
are required for evaluation of such subformulae. If a program satisfies this
condition it is stratifiable.

Any stratifiable program $p$ can be subdivided into pairwise disjoint
sets of rules $p = p_1 \cup \ldots \cup p_k \cup \ldots \cup p_n$ that are known as strata organized
so that $p_1$ contains rules without negative literals, or rules that contain
negative literals corresponding to database predicates and every stratum
$p_i$ contains only the rules with negative literals corresponding to strata of
lower layers. Computation is realized on a stratum - by - stratum basis.
A closed world assumption is interpreted locally in each stratum.

An example of a program $Q$ ( $d$ is the only database predicate). It is
assumed that sets with instances defined on product types correspond to
predicates $p$, $q$, $s$, $d$ and $r$.

$$
\begin{align*}
& r_1 : p(pe[x, y]) : \neg q(qe[x, y]) & s(se[x, y]). \\
& r_2 : q(qe[x, y]) : \exists x z (q(qe_1[x, z]) & q(qe_2[z, y])). \\
& r_3 : q(qe[x, y]) : \neg d(de[x, y]) & \neg r(re[x, y]). \\
& r_4 : r(re[x, y]) : \neg d(de[y, x]). \\
& r_5 : s(se[x, y]) : \neg \exists x z t (x <> y & (q(qe_1[x, z]) & q(qe_2[x, t]))).
\end{align*}
$$

The program has strata $Q : Q_1 = \{r_4\}, Q_2 = \{r_2, r_3, r_5\}, Q_3 = \{r_1\}$.

Generally a program can have several different equivalent stratifications.

In the SYNTHESIS language a program can be stratifiable not only
with respect to a negation but also with respect to a construction of
grouping in a head of a rule, to an implication in the universal quantifier
formulae, to a construction $form_{class}$. These constructions similarly to
a negation can lead to nonmonotonic fixpoint iteration of the SYNTHES-
SIS program. Arbitrary formulae with the universal quantifier, $group_{by}$
constructions, parameterized formulae, program control facilities belong
to such constructions. Due to the fact that a concretization of param-
eterized formulae can lead to nonadmissible programs, it is required to
complete their concretizations before the beginning of a computation pro-
cess. Program control facilities should not violate a program execution
order that is given by its stratification. A definition of a notion of the
SYNTHESIS program stratification follows.

It is essential that the program admisibility depends on an order of
execution of functions updating an information resource base (such as
functions of frame base manipulation, object creating (deleting) functions) that can be used in a process of a SYNTHESIS program evaluation. In the SYNTHESIS language a required order of execution of update functions can be established by program control facilities.

Example:

\[
p(y) : \neg \\
    \text{all } x(p(x) \rightarrow b1(b) \& x = b.f1, y = b.f2)) \mid \\
    b2(bb) \& y = b.f3.
\]

Assume that a set \( b1 \) contains three ADT values \( v1, v2 \) and \( v3 \) with interfaces containing operations \( f1 \) and \( f2 \), and a set \( b2 \) contains one value \( v4 \) with an interface operation \( f3 \). Assume that

\[
    v1.f1 = 1, \ v1.f2 = 2,
    v2.f1 = 2, \ v2.f2 = 2,
    v3.f1 = 1, \ v3.f2 = 3,
    v4.f3 = 1.
\]

On the first step of a fixpoint iteration a value of a set \( p \) will be obtained. A result of an application of the only interface operation of this value (that corresponds to a position of the first argument of a predicate \( p \)) is 1. This value will be denoted as \( p(1) \). On the second step two new facts will be obtained: \( p(2) \) and \( p(3) \). However, on the third step new facts obtained on the second step will be disproved.

A result of a formula \( \text{form\_class} \) is a new class of objects satisfying the constraints imposed by an argument formula. Obviously, till the moment of this construction evaluation an argument formula should be completely evaluated. In the opposite case on the subsequent steps of a fixpoint iteration the constraints that should be satisfied by the formed class will become stronger. Therefore a disproval of an information that was previously deduced becomes possible.

To define a condition of a stratifiability of the SYNTHESIS programs a notion of a dependency graph (DG) of a SYNTHESIS program \( P \) is introduced. A schema of a SYNTHESIS language formula is a form derived from an initial formula by a substitition of predicate occurences by their names and by elimination of variables remaining bound by quantifiers. In constructions \( \text{form\_class} \) only schemas of the argument formulae will be preserved. For instance, to a formula "all \( x ( \neg p(x) \rightarrow q(x, y)) \)" a schema "\( \text{all} ( \neg p \rightarrow q) \)" corresponds. The nodes of DG(\( P \)) are names of intensional predicates and schemas of nonatomic formulae that occur in
E Rules and recommendations for program construction

A directed arc in DG from a node $p$ to a node $q$ (denoted by $< p, q >$) exists iff a formula with a schema $q$ is a rule body with a head predicate $p$ or is a direct subformulae of the formula with a schema $p$. An arc $< p, q >$ can be marked by:

- symbols $\text{ˆ}$ and $\text{@}$ iff there exists a rule with a head predicate $p$ and one of the following cases takes place respectively: 1) a formula with a schema $q$ and with a symbol $\text{ˆ}$ is a rule body, 2) a head predicate contains a grouping construction;
- a symbol $\text{form}$ iff there exists a rule with a head predicate $p$ and a formula with a schema $q$ is an argument formula of a construction $\text{form\_class}$ that in its turn is a rule body;
- symbols $\text{ˆ}$ and $\rightarrow$ iff there exists a formula with a schema $p$ and one of the following two cases takes place respectively: 1) a formula with a schema $q$ occurs with a symbol $\text{ˆ}$ directly in a formula with a schema $p$, 2) a formula with a schema $q$ occurs directly in a formula with a schema $p$ as an antecedent of an implication;
- a symbol $\text{form}$ iff there exists a formula with a schema $p$ and a formula with a schema $q$ is an argument formula of a construction $\text{form\_class}$ that in its turn is a direct subformula of a formula with a schema $p$.

Example. Let a program $P$ appears as follows:

1. $r_1 : p(< y >) : \neg \forall x((p1(x) \& \text{ˆ}p3(x))\rightarrow$
   $\text{form\_class}(p2(p[x, y]), cl, spel)).$
2. $r_2 : p1(x) : \neg b1(x) \& b2(x)$.
3. $r_2 : p2(p[x, y]) : \neg b1(x) \& b3(b[x, y]).$
4. $r_2 : p3(x) : \neg b1(x) \& b4(x).$

Here $b1, b2, b3, b4$ are some database predicates. It is assumed that sets with instances defined on product types correspond to $p2$ and $b3$ predicates. A dependency graph $\text{DG}(P)$ is shown in the figure.

A stratification of the SYNTHESIS program $P$ consists in a decomposition of a set of nodes of $\text{DG}(P)$ on subsets (strata) $P_1, \ldots, P_n$, so that the following conditions should be satisfied:

- if $p$ is an instance of $P_i$, $q$ is an instance of $P_j$ and $< p, q >$ is an arc of $\text{DG}(P)$ then $i \geq j$;
E.2 Admissible logic programs in the SYNTHESIS language

- if \( p \) is an instance of \( P_i \), \( q \) is an instance of \( P_j \) and \( < p, q > \) is an arc of \( DG(P) \) marked by one of the following symbols ⊹, @, − or form then \( i > j \).

One of the possible variants of stratification of a program example above follows:

\[
\begin{align*}
P_1 &= \{p3\}, \\
P_2 &= \{p_1, p_1\&\neg p_3, p_2\}, \\
P_3 &= \{all((p_1\&\neg p_3)− > form класс(p_2))\}, \\
P_4 &= \{p\}.
\end{align*}
\]

An order of execution of predicates and formulae correspond to an order of strata that is defined by their indices.

A SYNTHESIS program is stratifiable if its DG does not contain loops that include marked arcs.

A SYNTHESIS program is admissible if it is stratifiable, safe and bottom-up computable.
F SYNTHESIS Formulae Language Subset

F.1 General rules

Syfs (SYnthesis Formulae language subset) is a subset of the language of logic formulae of SYNTHESIS. Syfs is presented in this Appendix.

The basic construct in Syfs is a logic program. The logic program is defined as a set of rules. A rule consists of a head and a body. A head of the rule is a class predicate. An instance type of the class is defined by an ADT (object or nonobject). The logic program can contain few rules with class predicates in the heads of these rules corresponding to one and the same class. In this case the logic program specifies each such result class as a union of result classes of the respective rules. A body of the rule is a formula which consists of a conjunction of class predicates, function predicates and constraint. Logic programs in Syfs can be used for the same purposes as in SYNTHESIS (subsection 9.1).

A logic program is interpreted in a particular context defined by a SYNTHESIS schema consisting of a set of modules. Main constituents of modules used in Syfs include: function, class and type specifications accessed by their identifiers (or qualified identifiers). Reducts and anonymous specifications of ADT are used in Syfs. Frames and temporal facilities of SYNTHESIS are beyond the Syfs capabilities.

F.2 Logic program

Logic program is represented as a rule list (subsection 9.11). The logic program defines a number of class predicates by one or more rules containing each such class predicate in the head of the rule. In the case when few rules define the same class predicate the logic program defines the class predicate with a result set equal to a union of result sets of these rules. In Syfs a rule has quite restricted syntax (most of specific formulae of the SYNTHESIS language (subsection 9.2) are not included into Syfs):

\[ r : -p_1 \land \ldots \land p_n \land f_1 \land \ldots \land f_m \land C \]

Here \( r \) is a head of the rule, \( p_1, \ldots, p_n \) are class predicates, \( f_1, \ldots, f_m \) are function predicates and \( C \) is a constraint. The head of the rule is a class predicate. It has the following form:

\[ q(x/R) \]
Here \( q \) is a class identifier, \( x \) is a variable identifier (an underscore symbol can be used instead of \( x \) to denote an anonymous variable) and \( R \) is an ADT (or more often its reduct (section 6.2)).

The body of the rule is a formula

\[
p_1 \land \ldots \land p_n \land f_1 \land \ldots \land f_m \land C
\]

that consists of class predicates (including those defined in the logic program), function predicates and constraint. It is required that at least one class predicate should occur in the body. The formula of the body defines a result set. The result class of the rule is the result of a transformation of the result of the body into the result set. Each class predicate contains a typed variable (subsection 9.2). The variable can be typed by ADT or by its reduct.

The following reduct element construct is allowed:

\[
a/T : t
\]

Here \( a \) is identifier defining a name of a reduct attribute, \( T \) defines a type of the reduct attribute, \( t \) is a path of the reduct attribute. The path is a dot-delimited list of attribute identifiers having the following form:

\[
a_1. a_2. \ldots. a_k
\]

Here \( a_1 \) is an attribute of the source ADT, \( a_2 \) is an attribute found in the type of an attribute \( a_1 \), etc. A path in a reduct attribute can contain attributes with a type \( T \) that is included into set type definition \( \{ \text{set; type_of_element: } T; \} \). In such cases for simplicity no explicit typing of path elements is required but flattening always takes place (implicit typing always exists).

Constraint is a collection of conditions (subsection 9.2) linked by ampersands. In conditions the following predicates can be used: \(<, <=, =, >, >=\) (comparing to SYNTHESIS, \(<>\) is eliminated). The same symbols are used for the set predicates (Appendix B). In arithmetic expressions (subsection 9.2) the following variable types can be used: \( \text{integer, real, Boolean, string and time} \). Identifiers of type attributes encountered in the class predicates of the body can be used in the constraint (qualified by an ADT name if required).

Functional terms can be used in formulae and in constraints as function designators (denoting methods or autonomously of ADT defined functions) described in subsection 9.2.
F.3 The result set of the Syfs formula

In this section the result set of the Syfs formula is defined inductively by the structure of the formula:

- for a formula with single class predicate $p$;
- for a formula $B \& p$ where $p$ is a class predicate, $B$ is a formula;
- for a formula $B \& f$ where $f$ is a function predicate, $B$ is a formula;
- for a formula $B \& C$ where $C$ is a constraint, $B$ is a formula.

The result set of a formula $B \& p$ is defined as a join of the result set of $B$ and the result set of $p$. The result set of a single class predicate and of the operation join are described in subsection 9.3. The result set of a formula $B \& f$ is defined as an operation append of the function predicate $f$ to the result set of $B$. The operation append is defined in subsection 9.3.

The result set of a formula $B \& C$ is defined as follows. For every element of the result set of $B$ a conjunction of conditions in the constraint $C$ is evaluated. The result set of the formula $B \& C$ includes the elements that satisfy the condition.
References


Leonid Andreevich Kalinichenko
Sergey Alexandrovich Stupnikov
Dmitry Olegovich Martynov

SYNTHESIS: a Language for
Canonical Information Modeling and
Mediator Definition for
Problem Solving in Heterogeneous
Information Resource Environments

Typesetting by S.A. Stupnikov

Approved for printing 24.04.07.

IPI RAS Publishing House
119333, Vavilov st., 44 bldg. 2, Moscow.