ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

B Novikov, University of St Petersburg, Russia and J W Schmidt, University of Hamburg, Germany. (Eds)

Advances in Databases and Information Systems, Moscow 1996

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS '96). Moscow, 10-13 September 1996

Modeling Facilities for the Component-based Software Development Method

Klaus Berg and Leonid Kalinichenko

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Modeling Facilities for the Component-based Software Development Method

Klaus Berg
Siemens AG
Munich, Germany
e-mail: Klaus.Berg@zfe.siemens.de

Leonid Kalinichenko
Institute for Problems of Informatics
Russian Academy of Sciences
e-mail: leonidk@ipian23.ipian.msk.su

Abstract

Component-based software development (CBSD) technology uses components as first-class objects and therefore requires a good understanding of the nature of components. Industrial approaches to CBSD based on interoperability standards (such as OMG CORBA) lack of component semantics in their descriptive models. In this paper we present an overview of the SYNTHESIS method emerging the CBSD approach by introduction of semantic information to enrich and complement the industrial modeling facilities. The paper contributes to the development of modeling facilities for CBSD focusing on the interoperable systems design. Proper balance of formal and semi-formal modeling facilities is demonstrated to cope with the CBSD requirements.

1 Introduction

Component-based software development (CBSD) has become one of the hottest topics in the area of software engineering. CBSD is a promising solution intended to break up large monolithic software systems into interoperable components and thus to move us from producing handcrafted lines of code to system construction based on object-oriented software parts or components and automated processes. The latter use semantic knowledge to guide the assembly of those components into the desired target system.

In this paper we consider CBSD issues in frame of the SYNTHESIS method¹ attempting CBSD with reuse of pre-existing heterogeneous components [13, 14, 15]. SYNTHESIS emphasizes megaprogramming metaphor capturing the idea of scaling-up from non-distributed object-oriented systems to large systems of heterogeneous, distributed software components. We consider interoperability to be the universal paradigm for compositional software development in the range of the systems mentioned. Basically, interoperability implies a composition of behaviors. Correct compositions of software components should be semantically interoperable in the context of a specific application.

The overall goal of the SYNTHESIS project is to provide a uniform collection of modeling facilities suitable for different phases of the forward engineering activities as well as for the reusable information resource specification in the reverse engineering phases. The method considered in the paper focuses on the semantic interoperation reasoning process [15] that should lead to the concretization of the specification of requirements by views over the pre-existing information resources.

The paper is structured as follows. After a short summary of the state-of-the-art and the related R&D directions we identify basic CBSD issues remaining to be open. Presenting the SYNTHESIS method, we discuss in more details its original features intended for the steps of the process of the semantic interoperation reasoning. Further we concentrate on the SYNTHESIS modeling facilities intentionally separated into semi-formal part used for plausible reasoning in course of CBSD and formal part used for strict justification of the design and specification solutions. A comprehensive example showing how different modeling facilities interact is presented.

¹The SYNTHESIS project developed at the Institute for Problems of Informatics of the Russian Academy of Sciences (IPI RAS) is partially supported by the INTAS grant 94 -1817, the Russian Foundation for Basic Research grant 94-07-20453 and by the cooperating research between the Siemens Corporate R & D and IPI RAS

2 State-of-the-Art

The software factory idea [19] as well as a research on software reuse has given important impact on CBSD. The importance of the reuse topic has led to a series of international reuse conferences, and to government funded research projects like REBOOOT [16]. But besides basic research some companies like IBM or Hewlett Packard have set up programs introducing CBSD [11]. Very important for CBSD are also industrial activities in the field of interoperability standards like OLE2 or CORBA [20, 22].

3 Main Directions of Research and Development

Analyzing the academic and industrial landscape of CBSD we have identified three main directions that can be described with the keywords *ComponentWare*, *Framework-based software construction*, and *Formal & knowledge-based methods for component software*.

ComponentWare is a term that stands mainly for a series of industry related activities in the field of interrelating components via middleware platforms like COM from Microsoft [20] or CORBA by OMG [22] that allow objects to communicate among themselves, discarding language, address space, machine and operating system boundaries. Additional document models like OLE2 from Microsoft or OpenDoc from CILabs provide the end-user with the desktop metaphor of a *compound document*, a unifying container for different types of data and the related applications that manage the data.

Framework-based software construction provides the system developer with domain specific application frameworks and reference architectures that are skeletons of applications that the developer fleshes out with new components. As opposite to ComponentWare which is reuse by composition the framework approach is reuse by evolution and difference. Examples for this direction are the European ITHACA research project [21], the US DSSA initiative [2] or HP's Hybrid Kits program [11].

Formal and knowledge-based techniques are mainly basic research activities to describe components and the component assembly process by means of formal or semi-formal methods, taking into account domain knowledge for system construction process automation. The SYNTHESIS method presented in this paper belongs to this approach. It is essential for this CBSD direction that component semantics are described, not only the syntax.

4 The Influence of the Internet/WWW and Java on CBSD

When talking about CBSD today we cannot ignore the growing influence of the Internet technology, especially the WWW. First of all, the Web is an excellent resource of available components in the Componentware area, i.e., in the Microsoft OLE/COM environment (e.g. OCX components) as well as in the OpenDoc/CORBA area (e.g., OpenDocparts). Second, the WWW and the OMG CORBA are complementing each other and we can watch their merge now. Sun's object-oriented Internet development language Java [6] has the potential to speed up this process and - maybe - to revolutionize the Web. Before the coming of Java, Web technologies gave users a very crude, i.e. static, way to access the power of the Internet. Building component-based client/server, multi-user applications was almost impossible with pre-Java Web technology. Protocols such as HTTP focus on interaction with the user rather than on application interaction (http users can only really pull pages of text and graphics back and forth across the net) and so impose fundamental limitations on the nature of services accessible from a consumer application running on an Internet device or a home PC. Java increases the value of the Internet by bringing "live" applications into the picture and CORBA 2.0 ORB implementations like IONA's Orbix or Post Modern Computing's BlackWidow claim to bring this even one step further offering the ability to perform semantically rich client/server operations on the Web [7]. SunSoft is also working on a Java/CORBA connection, so that Java programs will be able to invoke remote methods in server based objects over the net.

The combination of ORBs and Java takes CORBA beyond the enterprise and into the global sphere. ORBs and Java together enable much more than simple Internet applications - they provide a truly portable platform for building and deploying large-scale, distributed client/server applications across both public and private networks.

Besides being an object-oriented, multi-threaded, and secure language, Java offers two interesting features to CBSD: First, Java is a cross-platform language because Java programs are "architectural neutral bytecodes". Second, Java allows small programs or applets (mini-applications) to be embedded within an HTML document. When the user clicks on the appropriate part of the HTML page, the applet is downloaded into the client workstation or PC environment, where it begins executing.

CORBA distributed object technology empowers the Java applet with standards-based connectivity to the world of information and computing services. Introducing CORBA to the Java environment means that applets are no longer restricted to simple interaction with the user, but are instead capable of taking part in complex interactions with backend services. With CORBA, Java applets transcend the limitations of simple Web browser technology - CORBA-compliant Java objects become the basis for the provision of Internet and interactive Multimedia Services on a world-scale. The Internet or enterprise-wide intranets build a kind of standardized sockets into which application components can be plugged in. According to IONA's vision [7] distributed applications could then be viewed as collections of "world-objects" - some may be downloadable to consumer devices, others may reside on backend corporate servers - all should be capable of sharing information with one another. Thus, a combination of the Java programming language with the CORBA standard for application integration offers an ideal solution for downloadable application components capable of accessing multiple, shared backend services located across the Internet. CORBA 2.0 provides the crucial missing link between the Java application (applet) running on a consumer device and the required backend service. Both CORBA and Java essentially seek to abstract the underlying hardware technologies and architectures. For component-based software development this factor brings about a reduction in learning curves and offers improvements in time-to-market as well as maintenance cost reduction.

As far as the SYNTHESIS environment is concerned, we plan to include the Internet and WWW facilities into the general architecture. OMG's CORBA 2.0 encapsulates the underlying information resources (components) and the architectures like IRO-DB [8] make it possible to represent heterogeneous databases with resource specifications in frame of the ODMG'93 standard. In this context we define three different kinds of the Internet sites to support the SYNTHESIS design method: the information resource provider sites, the designer sites and the application domain provider sites. Furthermore we distinguish between different design scenarios (centralized at the design site, cooperative that involves resource providers into the design, and an active resource scenario when information resources actively participate in the design process offering their own reuse possibilities).

5 Open Issues

Up to now we have no methods, guidelines or design heuristics on how to develop good frameworks. Possible steps in this direction are the metapattern/hot spots approach by W. Pree [23]. Framework adaptation, i.e. the customization to the user's needs, is also a field that needs more research. Active cookbooks [17] are an approach to support the user in this problem area. The technique of Design Patterns [10] could be helpful to document (part of) a framework.

On the other hand, the ComponentWare approach provides no application skeleton but individual components that can build the application when assembled and interconnected by a software bus (like CORBA's ORB). But the problem of components is that they do not have sufficient clean semantic specifications to rely on for their reuse. In this context the following issues are considered to be still open:

- Complete specifications (for machine and for human) of the available components and of the application requirements are necessary prerequisite for the method
- Homogeneous ("canonical") equivalent specifications for pre-existing components should be provided
- One and the same set of description facilities should be used for different layers of development (requirement specification, design and reverse engineering)
- Sound foundations are necessary to support provable requirement concretization and coherent component composition
- The design methodology should support design based on reuse and interoperable composition of components

- Componentware and framework approaches integration is desirable.

The SYNTHESIS method overviewed in the following sections addresses many of these open issues.

6 Related Work

In the context of CBSD the use of formal methods and domain knowledge is quite new but of growing importance. One significant activity in this direction is the U.S. Advanced Technology Program (ATP) *Component-Based Software*, sponsored by the National Institute of Standards and Technology NIST [5].

Scalable Automated Semantic-Based Software Composition is another project funded by NIST, the state of California and a consortium of companies. The project focus is on semantic -based composition and component synthesis based on specifications.

Composable Software Systems is a research project led by three members of the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA [25]. The project tries to develop a scientific and engineering sound foundation for designing, building, and analyzing composable systems, organized as collections of reusable components.

7 SYNTHESIS Modeling Facilities

A strategy for incorporation formal design method. SYNTHESIS modeling facilities should provide for semantic interoperation and reuse of the pre-existing resources to cope with the open CBSD issues identified above.

SYNTHESIS modeling facilities roughly can be subdivided into semi-formal and formal ones. The former (the SYNTHESIS language [13]) is intended as a mediator between the informal natural language specifications and the formal ones. We focus on model-based specifications for the latter [1].

For incorporation of formal specification method we exploit a transitional computer-assisted strategy [9]. These strategies have advantage of computer assistance available to move back and forth between semi-formal and formal specifications.

Semi-formal facilities of SYNTHESIS. Uniformity of the SYNTHESIS object model is based on the algebraic framework [18]. The fundamental concept of the SYNTHESIS object model [13] is *an abstract value*. Abstract values are instances of abstract data types (ADT) that resemble algebraic systems [18]. A SYNTHESIS object model is purely behavioral.

Type in the language is treated as the first-class value. Type variables have types as their values. Basic operations used in type expressions (mostly while implementing object calculus formulae) are operations of *type composition* (type meet and join) and of *type product*. Type specifications are abstract and completely separated of their implementations.

All operations over typed data in the SYNTHESIS language are represented by functions. Predicative specifications of functions are expressed by formulae of the SYNTHESIS object calculus.

Incorporating a sound foundation we focus on *model-based specifications* [4] chosen among other specification formalisms such as *logic-based*, *functional* and *algebraic*. The notion of execution of a model-based specification consists of the proof of the initial consistency of the model and the preservation of the invariants by the operations.

The model-theoretic methods [3, 24, 1] are based on pure mathematical abstraction of the specification of requirements and on the application of the provable stepwise refinement (including data and algorithmic refinement) in process of their development. During the refinement process, "off-the-shelf" components can be taken into account for their reuse. In SYNTHESIS we focus on the Abstract Machine Notation (AMN) [1] applying *transitional computer-assisted strategy*.

To succeed with the strategy, we are based on formal interpretation of the SYNTHESIS language features in AMN. We interpret each type of the SYNTHESIS specification by a separate abstract machine.

8 SYNTHESIS CBSD Method Overview

The method emphasizes integration, reuse, adaptation and reconstruction of the pre-existing components (the whole or the pieces of the existing components, legacy systems, databases, program packages, data files, multimedia data) for the new (or modified) system requirements. SYNTHESIS method is not considered as one rigid approach, but as top-down, bottom-up iterative processes of analysis, design and development. The interdependence of different phases of the SYNTHESIS method is shown on the Fig 1.

The conventional technique of the OO analysis and design is used for the requirement planning and domain analysis phases. This technique is augmented with the ontological specifications needed to resolve contextual differences with the pre-existing resources, with the specification of the result in the common declarative OO and logic based model that is two dimensionally uniform and with a possibility to justify the result using formal specification and proof facilities.

The information resource description technique is developed to complement the existing core interoperation technology (such as CORBA IDL) in order that the resources could be reused in the semantically interoperable environment. The specification of the resource should be complete for the semantic interoperation reasoning.

The design technique is based on the interoperable reuse of the pre-existing resources. For that the coherence of the contexts of the problem domains and of the resources is negotiated, the search of the relevant resource specifications is supported, the discrepancy reconciliation approaches, concretization view construction technique is provided. The results of the design can be provably checked.

The SYNTHESIS method is neutral to the possible methods providing the object-oriented requirement planning and analysis models as well as to the possible reverse engineering methods. The output of such methods should be transformed to the SYNTHESIS canonical model thus giving precise semantics to the diagrammatic notation.

9 Role of Formal Modeling Facilities

Justification of the decisions taken during the Forward phase. Formal checking of the result of the analysis phase is provided as follows. The type / class definitions of SYNTHESIS model are presented as abstract machines in the B AMN [1] notation to verify consistency of the resulting specification (in particular, to check that the methods defined preserve the invariants given by the assertions, ontological rules and other constraints of the model; to check consistency of type / subtype specifications: these specifications should have a model).

The design model is the refinement of the domain analysis model adapting it to the actual heterogeneous interoperable information resource environment. The formal counterpart of this concretization specification is given in AMN: the concretization of each application type is verified by its transformation into the B AMN and treating concretization as *a refinement* in B. Proof obligations corresponding to the refinement of an abstract machine are generated and proved.

Type specification mapping technique for the reverse engineering phase. For the heterogeneous world of information resources we provide a technique of mapping of the pre-existing resource type specifications into the canonical specifications uniformly defined in the SYNTHESIS language. The commutative type model mapping is developed through the following basic steps:

- construct the mapping of a source type specifications into type specifications of the canonical type model (including state and behavior mapping);
- provide an interpretation of source type model in the abstract machine notation;
- provide an interpretation in abstract machine notation of the types resulted in mapping of the source type model into the canonical types;
- justify the state-based and behavioral properties of the type mappings proving that a source type is a refinement of its mapping to the canonical type.

SYNTHESIS METHOD FOR ANALYSIS AND DESIGN OF
SEMANTICALLY INTEROPERABLE SYSTEMS

Fig. 1. SYNTHESIS method structure

In the reverse engineering phase the correctness of the SYNTHESIS-based specification of the resource is guaranteed by such procedure justified by the refinement relation of abstract machines. The commutative type model mapping is a specific technique providing for uniform representation in the canonical model of different type models determined by programming languages and DBMSs.

10 An Example Demonstrating the Roles of Formal and Semi-Formal Modeling Facilities

10.1 Semi-Formal Specifications of Requirements

We imagine a centralized Agency managing funds dedicated for research and development projects. We assume that the following type specifications were produced as a result of the forward analysis and design based on some OA & D method and of the transformation of its resulting definitions into the SYNTHESIS specifications. Thus we get a definition of the type *Proposal* and class *proposal*. In the SYNTHESIS language type specifications are syntactically represented by frames, their attributes - by slots of the frames. Additional information related to attributes can be included into metaslots. Syntactically frames are included into figure brackets { and }, slots are represented as pairs <slot name> : <slot value> (a frame can be used as a slot value), slots in a frame are separated by semi-colons. Metaslots (that are represented by frames) are written immediately after the slots to which they are related.

```
{Proposal;
in: type, mutable;
name: string;
area: string;
budget: real;
metaslot
in: budget_sem, {
{area = 'computer_sci' →
budget_kind = 'annual' & currency = 'rouble'}}
end
status: {enum; enum_list:
{'accepted', 'terminated', 'review'}};
starting_date: time;
termination_date: time;
accept_proposal: {in: function;
  params: {+prop/Proposal, +m/real, +t/time};
  {{prop.status = 'review' &
after(prop.termination_date, prop.starting_date) &
prop.status = 'accepted' &
prop.starting_date = t & budget = m &
prop.termination_date =
add(t, minus(prop.termination_date,
prop.starting_date))}}}}
};
{proposal;
in: class;
instance_section: {Proposal} };
```

Simple concept definition rule is incorporated into the type *Proposal* by association with the *budget* attribute of a metaslot containing a definition of a rule: 'if proposal area is computer science then budget is annual and currency is roubles'. Here *budget_sem* is a SYNTHESIS metaclass introducing metattributes (*budget_kind*, *currency*) that are

used in the definition of rules characterizing budget contextual semantics required by the application. *budget_sem* in the language is treated as a class of attributes (*association* metaclass).

```
{budget_sem;
in : metaclass, association;
instance_section:
{ budget_kind : string;
currency : string
}};
```

10.2 Semi-Formal Specifications of the Pre-Existing Resources

We assume that the specifications of the pre-existing resources equivalently represent their semantics. We assume that at the Industrial Labs an information system is used that contains the following specifications of types and classes.

```
{Submission;
in: type, mutable;
name: string;
research_area: string;
req_money: real;
    metaslot
    in: budget_sem, {
        { (research_area='computer_sci' →
budget_kind = 'annual' & currency = 'rouble') &
(¬ (research_area='computer_sci') →
budget_kind = 'project_duration' &
currency = 'rouble')
        }}
    end
starting_date: time;
grant: real;
proj_starting_date: time;
termination_date: time;
duration: time;
accept_submission: {in_function;
    params: {+s/Submission,+m/real, +t/time};
    {{¬ in(s,project) &
include(s,project) & proj_starting_date'=t &
termination_date'=null &
grant' = m}}}};
};
{submission;
in: class;
instance_section: {Submission }}
};
{project;
in: class;
instance_section: {Submission}
};
```

The class *project* is a subset of the class *submission* that includes only those submissions that have been accepted.

The example shows how to design the *proposal* class reusing preexisting resources at the Industrial Labs.

10.3 Concretization View for the *Proposal* Class

An example of the specification of the concretization view for the *proposal* class follows:

```
{virt_proposal;
in: class;
metaslot

{comment; In specifications of a view class a function computing the class as a set is defined in a metaslot associated
with an attribute in: class. We do not care here whether we declare a virtual class that exists only during its evaluation
or a materialized class. The specification of the function is given below by a formula of the object calculus of the
SYNTHESIS language [13]. }

prop_view: {in: function;
params: -returns/virt_proposal as_class;
enforcement: on_access;

{{
submission(s) & area(s) == research_area(s) &
budget(s) == f_budget(s) &
status(s) == f_status(s) &
termination_date(s) == f_termination_date(s) &
virt_starting_date(s) == f_starting_date(s)
{comment; The assignment formulae containing == provide for appending new attributes to the resulting type and
evaluating them. The specifications of f_budget, f_status, f_termination_date, f_starting_date functions should be pro-
vided. Only f_budget is shown below to save space. }
}}}
end

{comment; We continue the definition of the virt_proposal class declaring its superclass and stating that there is one-
to-one correspondence between objects of the proposal and virt_proposal classes.}

superclass: proposal;
rename: virt_starting_date : starting_date;
metaslot

    in: one-to-one;
    r: invariant,
    {{ { all i/virt_proposal
        (ex p/proposal (name(p) = name(i)
        → same(i,p))))}}
end

{comment; Based on the analysis of the resource classes corresponding to the application class the concretization
axioms stating how to refine the application class states by the states of the relevant resource classes should be con-
structed. From the analysis of the submission/project specification the designer should deduce the following con-
cretization axioms showing how to refine proposal class state by the submission/project classes state. axioms is a
predicate constraining admissible states of the instances of the virt_proposal class. }

axioms: {in: predicate; {{ { all p/proposal ( ex s/submission
(name(p) = name(s) & area(p) = research_area(s) &
(in(s,project) & budget(p) = grant(s) & starting_date(p) =
proj_starting_date(s) & ((termination_date(s) < > null &
termination_date(p) = termination_date(s) & status(p) =
'terminated') | (termination_date(s) = null_time &
```

```

status(p) = 'accepted' & termination_date(p) =
add(duration(s), proj_starting_date(s))) |
( $\neg$  in(s,project) & budget(p) = req_money(s) &
starting_date(p) = starting_date(s) & status(p) = review &
termination_date(p) = add(duration(s), starting_date(s))))};

accept_proposal: { in: function;
    params: { +prop/virt_proposal,
    +m/real, +t/time});
    { { submission(s/submission) &
    name(s) = name(prop) &
    accept_submission(s,m,t) }} };

{ comment; The function accept_proposal overrides the corresponding function of the superclass. This function reuses
the resource function accept_submission. }
}

f_budget: { in: function;
    params: { +s/submission,
    -return/proposal.budget };
    { { in(s,project) &
    return = grant(p) |
     $\neg$  in(s,project) &
    return = req_money(s) }} }

```

10.4 Mapping of the SYNTHESIS Specifications to AMN

10.5 Abstract Machine Notation

Abstract Machine Notation [1] allows to get an exact mathematic definition (specification) of the modeled entity properties (in particular it may be a computer program). Then such definition may be analysed formally. So AMN program is used for proving of the correctness of initial entity definition. In particular it may be used for correctness proof and symbolic transforming of initial program (for example, composition, decomposition, refinement, etc.)

While specifying an abstract machine we should define the machine states and operations allowing to get such states. To specify state, two kinds of entities should be defined: variables defining the state components and invariants. Invariants are laws which must be satisfied by the static states of a system.

A specification of an operation describes properties and relationships that must be satisfied during a change of a state.

In common model-theoretical languages such as VDM or Z such properties are specified using logical assertions that relate values of the state variables before and after operation executing. For these puposes AMN uses so called *calculus of substitutions* allowing to express properties of operations in terms of predicate transformers which bind with some post-condition of the operation its weakest precondition. Generalized substitutions (which are operators of such calculus) may be considered as an abstract machine commands.

After an operation being specified one should prove it preserves invariants. To do this every invariant is considered as a post -condition to which the operation (i.e., a predicate transformer) is applied. This application results in forming of a new predicate which must be proved too.

So pre - and post conditions are integrated in notation which looks like a simple programming language. The commands of the language are generalized substitutions that generalize Dijkstra's guarded commands.

Every generalized substitution S defines a predicate transformer binding with some post-condition R its weakest pre- condition $[S]R$ that guarantees the invariance of R after an operation execution. If it is so, one says that S implements R . Kinds of the generalized substitutions are listed below.

- Multiple substitution

$[x, \dots, y := E, \dots, F]R \Leftrightarrow R'$,
where R' is R in which free occurrences of $x, \dots, y$ are simultaneously replaced by $E, \dots, F$ correspondingly

- Empty substitution

$[skip]R \Leftrightarrow R$

- Pre-conditioned substitution

$[R_1 \mid S]R \Leftrightarrow R_1 \& [S]R$

- Guarded substitution

$[R_1 \Rightarrow S]R \Leftrightarrow R_1 \Rightarrow [S]R$

- Bounded choice substitution

$[S \sqcap T]R \Leftrightarrow [S]R \& [T]R$

- Unbounded choice substitution

$[@z.S]R \Leftrightarrow \forall z. [S]R$,
where z is not a free variable in R

So every generalized substitution defines a rewriting rule transforming the next predicate to a pre-condition. Pre-conditions describe situations (states) which are admissible for the execution of the corresponding operation. Under such conditions the operation can be completed. Guarded substitution is implemented if predicate R_1 is satisfied.

Bounded choice corresponds to restricted form of non-determinism: any choice is admissible and the final decision depends on the operation developer during the concretization process. Any substitution must preserve the post-condition R . It is why the conjunction sign is used in the predicate.

10.6 Elements of the Abstract Machine Notation

We start with elementary notions [1] resembling programming notation for modules (packages) necessary to organize the specifications of large systems as independent entities having well defined interfaces. The simplified syntactic structure of the machine looks as follows:

machine

Identifier [(Identifier, ... , Identifier)]

sets

GivenSet; ... ; GivenSet

variables

Variables

invariant

Predicate

initialization

Substitution

operations

Operation;

...

Operation

end

Abstract machine has a name and may have some formal parameters (to be either natural numbers or non-empty finite sets). An abstract machine has a number of variables that should obey a certain number of predicates forming together the invariant of the machine. The invariant allows to set-theoretically type each variable. The sets defined in

the set clause of a machine constitute the basis of its type system. An abstract machine also has an initialization that is a substitution. Finally, an abstract machine has a number of operations defined with the following syntax:

$$\begin{aligned} Variable &\leftarrow Identifier[(Variable)] \cong Substitution \\ Identifier[(Variable)] &\cong Substitution \end{aligned}$$

Once an abstract machine has been written, one has to check that a certain number of conditions are met. Such conditions together form the *proof obligations* of the machine that are shown below in a simplified form for the small machine skeleton that follows.

machine

Identifier

variables

z

invariant

R

initialization

T

operations

Operation_name $\cong$ *pre L then S end*;

end

Proof obligations templates:

$\exists z \cdot R$

$[T]R$

$R \wedge L \Rightarrow [S]R$

The first proof obligation is just the existence proof obligation for variables. The second one concerns the establishment of the invariant by the initialization. The last one concerns the preservation of the invariant by each operation.

10.7 Basic Modularization Features of AMN

includes and **promotes** clauses are introduced to make a machine that is based on the *composition* of other machines. **includes** gives names of included machines with *renaming* (dotted identification that required in the case of multiple inclusion) and with actual parameters. Natural number parameters are instantiated by expressions denoting natural numbers and the set parameters are instantiated by means of expressions denoting simple sets.

A machine including other machines can have its own glueing invariants that should be preserved by promoted and new operations of the new machine. New machine may have its own variables with the corresponding initialization.

The **promotes clause** contains the list of operation names of those operations of the included machines that are to become without any modifications genuine operations of the new machine.

The **uses** clause is different from **includes**. Operations of used machines cannot be mentioned in the using machine. Several machines can thus use the same machine. Parameters of the used machines are not instantiated. Sets, constants and variables of the used machines can be read in the using machine.

sees clause contrary to **uses** does not allow to mention elements of the seen machines in the invariants of seeing machine. Thus a seen machine can be refined independently of the machine that sees it.

10.8 Refinement of Abstract Machines

The ultimate goal of the B-technology is to have abstract machine implemented eventually as software modules by means of some programming notation. So, we have to transform abstract machines so that they could eventually be implemented by means of the programming notation. This will be done by a step by step restriction of the constructs that could be used further. This activity is called a *refinement*.

Algorithmic refinement consists in removing of non-determinism by being more and more precise about the way our operations are to be eventually made concrete. At the same time we should relax preconditions.

Data refinement consists in removing completely all variables whose types are too complicated to be implemented as such and in replacing them by simpler variables whose types correspond to those found in programming notations: that is, essentially, natural numbers taken in certain intervals (scalar types) and functions from scalar types to themselves (array types).

Defining data refinement we suppose that we have two substitutions S and T working within two different machines (within two distinct variable spaces represented say, by two variables x and y). We assume that these variables are members of the two respective sets s and t so that $x \in s$ and $y \in t$ are respective invariants of these machines. We suppose that these variables are related by a certain binary relation v from s to t such that $ran(v)$ is equal to t . The relation v is called the *abstraction relation*.

Now, the refinement of the abstract machines is defined as follows: a machine N is said to refine a machine M if a user can use N instead of M without noticing it.

Syntactic construct **refinement** is introduced that resembles a machine. However, a refinement can refine either a machine or another refinement. The invariant clause of refinement is just the abstraction relation defined above: it expresses the change of variables between the two constructs. The operations of the **refinement** only involve the variables of the **refinement**, not of the construct being refined. At another extreme pure algorithmic refinement can take place: in this case the variables of the **refinement** and of the construct being refined are the same. In a refinement new given sets may be introduced as well as a more precise value for a given deferred set.

After the refinement is specified it is necessary to prove that it indeed refines what it is claimed to refine. For this a number of proof obligations are generated according to the following templates related to the following abstract machine and its refinement:

machine

AM Identifier

variables

x

invariant

P

initialization

S

operations

$z \leftarrow OpName \cong pre\ Q\ then\ T\ end;$

end

machine

Identifier

refines

AM Identifier

variables

y

invariant

R

initialization

U

operations

$z \leftarrow OpName \cong pre\ L\ then\ V\ end;$

end

The proof obligations templates look as follows:

$\exists(x, y) \cdot (P \wedge R)$

$[U] \rightarrow [S] \rightarrow R$

$\forall(x, y) \cdot (P \wedge R \wedge Q \Rightarrow L \wedge [V'] \rightarrow [T] \rightarrow (R \wedge z = z'))$

Here V' stands for substitution V within which the variable z has been replaced by z' .

The first of the proof obligations is an existence proof obligation for the new variables of the refinement. The second deals with the correct refinement of initialization and the third of them deals with the correct refinement of operations.

10.8.1 Abstract Machine Resulting from the Proposal Type Mapping

To save space we have omitted from the example mentioning of the machines constituting the environment into which the specified machine are embedded. We do not show also all the operations of the machines.

machine

mproposal

sets

STATUS = {accepted, terminated, review}

variables

proposal_name, proposal_area, proposal_budget, proposal_status,
proposal_starting_date, proposal_termination_date,
budget_smnt_currency, budget_smnt_budget_kind

invariant

proposal_name $\in$ proposal $\longrightarrow$ STRING

/* proposal is a set representing the proposal class */

/* Other invariants specifying variables types are omitted */

& $\forall$ xx.(xx $\in$ proposal &
proposal_area(xx) = "computer_sci" $\Rightarrow$
(budget_smnt_budget_kind(xx) = "annual" &
budget_smnt_currency(xx) = "rouble"))

operations

budget $\longleftarrow$ get_proposal_budget(pr) =

pre pr : proposal

then budget := proposal_budget(pr)

end ;

accept_proposal(prp, mm, tt) =

pre prp $\in$ proposal

& mm $\in$ NAT

& tt $\in$ TIME

& (after(proposal_termination_date(prp),

proposal_starting_date(prp)) = TRUE)

& proposal_status(prp) = review

then

let yy **be** yy = proposal_starting_date(prp) **in**

proposal_status(prp) := accepted

|| proposal_starting_date(prp) := tt

|| proposal_budget(prp) := mm

|| proposal_termination_date(prp) := add(tt,

(minus(proposal_termination_date(prp) , yy)))

end

end

end

10.8.2 Abstract Machine Resulting from the Submission Type Mapping

machine

 rsubmission

variables

 submission_name, submission_research_area, submission_req_money,
 submission_starting_date, submission_duration,
 project_grant, project_proj_starting_date,
 project_termination_date,
 budget_sem_currency, budget_sem_budget_kind

invariant

 submission_name $\in$ submission $\longrightarrow$ STRING

/* submission is a set representing the submission class */

/* Other invariants specifying variable types are omitted */

 & $\forall$ xx.(xx $\in$ submission $\Rightarrow$
 (submission_research_area(xx) = "computer_sci" $\Rightarrow$
 budget_sem_budget_kind(xx) = "annual" &
 budget_sem_currency(xx) = "rouble"))
 & $\forall$ xx.(xx $\in$ submission $\Rightarrow$
 ((not(submission_research_area(xx) = "computer_sci")) $\Rightarrow$
 budget_sem_budget_kind(xx) = "project_duration" &
 budget_sem_currency(xx) = "rouble"))

operations

 bb $\leftarrow$ prsubcl(pr)=

/* This is a boolean operation checking membership in the project class. */

pre

 pr $\in$ submission

then

 bb:=bool(pr $\in$ project)

end;

accept_submission(ss, mm, tt) =

pre

 ss $\in$ submission & not(ss $\in$ project)

 & mm $\in$ NAT

 & tt $\in$ TIME

then

 include_project(ss)

 || project_proj_starting_date(ss) := tt

 || project_termination_date(ss) := null_time

 || project_grant(ss) := mm

end

end

10.8.3 Abstract Machine Resulting from the Concretization View Mapping

implementation

 implpl

refines


```

mproposal
imports
rsubmission
invariant
proposal = submission
/* We require here that the set corresponding to proposal class should be equal to the extent of the submission class.
*/
/* The axioms below define the abstraction relation establishing data refinement of the Proposal type by the Submission
type. They are based on the axioms predicate defined for the concretization view. */
    & ∀ xx.(xx∈proposal ⇒ proposal_name(xx) = submission_name(xx))
    & ∀ xx.(xx∈proposal ⇒ proposal_area(xx) = submission_research_area(xx))
    & ∀ xx.(xx∈proposal ⇒ budget_smnt_currency(xx) = budget_sem_currency(xx))
    & ∀ xx.(xx∈proposal ⇒
budget_smnt_budget_kind(xx) = budget_sem_budget_kind(xx))
    & ∀ xx.(xx∈proposal & not(xx∈project) ⇒
proposal_budget(xx) = submission_req_money(xx))
    & ∀ xx.(xx∈proposal & xx∈project ⇒
proposal_budget(xx) = project_grant(xx))
    & ∀ xx.(xx∈proposal & not(xx∈project) ⇒
proposal_starting_date(xx) = submission_starting_date(xx))
    & ∀ xx.(xx∈proposal &
xx∈project ⇒
proposal_starting_date(xx) =
project_proj_starting_date(xx))
    & ∀ xx.(xx∈proposal & not(xx∈project) ⇒
proposal_status(xx) = review)
    & ∀ xx.(xx∈proposal & xx∈project &
not(project_termination_date(xx)= null_time) ⇒
proposal_status(xx) = terminated )
    & ∀ xx.(xx∈proposal & xx∈project &
project_termination_date(xx) = null_time ⇒
proposal_status(xx) = accepted)
    & ∀ xx.(xx∈proposal &
not(xx∈project) ⇒
proposal_termination_date(xx) =
add(submission_duration(xx),
submission_starting_date(xx)) )
    & ∀ xx.(xx∈proposal & xx∈project &
not(project_termination_date(xx)= null_time) ⇒
proposal_termination_date(xx) =
project_termination_date(xx))
    & ∀ xx.(xx∈proposal & xx∈project &
project_termination_date(xx) = null_time ⇒
proposal_termination_date(xx) =
add(submission_duration(xx),
project_proj_starting_date(xx)))
operations
budget ←— get_proposal_budget(pr) =
    var bb in

```

```

    bb ← prsubcl(pr);
    if bb = TRUE
    then
    budget ← get_project_grant(pr)
    else
    budget ← get_submission_req_money(pr)
    end
    end ;
accept_proposal(prp, mm, tt) =
    begin
    accept_submission(prp, mm, tt)
    end
end

```

The machines above were proved by I.A.Chaban using the B-Toolkit environment [1]. The proof justifies consistency of semi-formal specifications and correctness of the design with reuse of the pre-existing components.

11 Conclusion

The CBSD method complementing industrial OAD methods and interoperable environments is overviewed. The design technique proposed is based on the interoperable reuse of pre-existing components. For that the coherence of contexts of the problem domains and of the resources is negotiated, the search of the relevant resource specification is supported, the discrepancy reconciliation approaches, concretization view construction technique are provided.

The CBSD method is based on heuristic provisions for search and composition of relevant components into views serving as refinements for the specification of requirements. Formal specification languages are far from being ideal tool for exploring and discovering the problem structure during the refinement. On the other hand, object models widely used in the OAD CASEs are not sufficiently semantically rich to cope with the CBSD open issues identified.

To provide adequate semantic facilities suitable for heuristic methods augmented with formal proof and refinement facilities the SYNTHESIS modeling tools are separated into semi-formal and formal ones. The uniform purely behavioral object model was developed for the former. Model-theoretic facilities were used for the latter.

The paper contributes to understanding of semi-formal and formal modeling facilities interaction in forward and backward phases of the semantically interoperable information systems design.

References

- [1] Abrial J.-R. B-Technology. Technical overview. BP International Ltd., 1992, 73 p.
- [2] Armitage, J.W.: Process Guide for the Domain Specific Software Architecture Process Life Cycle, CMU-SEI-93-SR-021, Report, Software Eng. Inst., Dec. 1993
- [3] Bjorner D., Jones C.B., Formal specification and software development. Prentice-Hall Series in Computer Science. Prentice-Hall, 1982
- [4] Borgida A., Mertikas M., Schmidt J.W., Wetzel I. Specification and refinement of databases and transactions. DAIDA ESPRIT 892, Hamburg Univ., 1990
- [5] Fisher, D.A.: Supplemental Information for Program Competition 94-06, May 1994. Part of ATP Component-Based Software.
- [6] Information about Sun's Java language available at WWW: <http://java.sun.com/>.

- [7] Information about IONA's Orbix Java connection available at WWW: <http://www.iona.com/>.
- [8] Huck G., Fankhauser P., Busse R., Klas W. IRO-DB: an object-oriented approach towards federated and interoperable DBMS. Proceedings of the International Workshop on Advances in Databases and Information Systems, Moscow, May 1994
- [9] Fraser M.D., Kumar K., Vaishnavi V.K. Strategies for incorporating formal specifications in software development. CACM, v. 37, N 10, October 1994
- [10] Gamma, E., Johnsohn, R., Vlissides J.: Design Patterns. Addison -Wesley, 1994.
- [11] Griss, M., Wentzel, K.D.: Hybrid domain-specific kits for a flexible software factory. In Proceedings of the ACM Symposium on Applied Computing, Reusability Track, Phoenix, AZ, March 1994.
- [12] Hutt A. (editor). Object Analysis and Design. Description of methods. Object management group. John Wiley and Sons. 1994. p. 202
- [13] Kalinichenko L.A. SYNTHESIS: a language for description, design and programming of interoperable information resource environment. Institute for Problems of Informatics of the Russian Academy of Sciences, September 1995
- [14] Kalinichenko L.A. A Complementary Architecture Integrating Industrial and Semantic Interoperation Environments. Proc. of the East EurOOPe'93 International Conference, Bratislava, November 1993
- [15] Kalinichenko L.A. Emerging semantic-based interoperable information system technology. Computers as our better partners. Proceedings of the International IISF/ACM Symposium, Tokyo, World Scientific, 1994
- [16] Karlsson, E.A. (Ed.): Software Reuse - A holistic approach. Wiley & Sons, 1995.
- [17] Krasner, G.E. and Pope S.T.: A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. JOOP, 1(3), 1988.
- [18] A.I. Maltsev. *Algebraic systems*. Moscow, Nauka, (in Russian), 1970.
- [19] Matsumoto, Y.: A Software Factory: An Overall Approach to Software Production. IEEE Computer Society Press, 1987.
- [20] Microsoft & Digital Equipement: Common Object Model Specification, DRAFT Version 0.1, Juli 1994.
- [21] Nierstrasz, O., Gibbs, S., Tschritzis, D.: Component-Oriented Software Development. CACM, Vol. 35, No. 9, September 1992.
- [22] OMG: Object Management Group: *The Common Object Request Broker - Architecture and Specification*, OMG Document No. 91.12.1, December 1991.
- [23] Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.
- [24] Spivey J.M. The Z Notation. A reference manual. Prentice-Hall, 1989
- [25] Shaw, M., Wing, J., Garlan, D.: Web page (1995): <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/compose/-www/index.html>