Compositional Development of Information Systems: Methods and Facilities

Kalinichenko L.A., Briukhov D.O., Skvortsov N.A.
Institute for Problems of Informatics
Russian Academy of Sciences
Vavilova 30/6, Moscow, V-334, 117900
E-mail: {leonidk,brd,scvora}@synth.ipi.ac.ru

Abstract

The paper defines a new approach for information systems development intended for reuse of pre-existing components. The approach called 'compositional' consists in identification of relevant fragments of the component specifications and constructing their compositions satisfying specifications of requirements. Fundamentals of the method include the canonical model having formal interpretation and applied for specification of requirements and components; the theory of refinement, making possible to justify component reuse; specification calculus forming a basis for compositional constructing of the systems from components; ontological approach used for semantic contexts integration. Due to the used methods of component specification and retrieval, the compositional method is scalable with respect to the number of existing components, provides for information system construction using various kinds of components (software, information, process ones), provides for provable justification of the design.

The prototype considered combines conventional OAD methods support facilities with the facilities of the new compositional development method. This prototype is intended for implementation generation in the CORBA-like environment.

1 Introduction

The idea of constructing information systems (IS) from components is being developed for many years. Actually, megaprogramming (G.Wiederhold) is a goal – constructing systems from large blocks. According to this idea, it is more profitable to invest into development of reusable components than to apply top down construction of IS going from the specifications of requirements to working

systems. In spite of obvious attractiveness of the idea and intensive research and development in the area, the results reached are hardly comparable with the recent successes in the related technological developments.

The Internet is full of components. New technologies provide for technical ability of interoperable use of software components (OMG CORBA) as well as information components (WWW). CORBA and WWW technologies were integrated and can be used together in IS construction. Standard object interface description language (IDL) provides for creating repositories of server interface specifications being independent of implementation platforms and programming languages. Object request brokers, according to CORBA, technically provide for development of distributed, interoperable IS from objects. In the nearest future the development of WWW will allow to consider the Web as a database with XML [6] as the data model. WWW sites form information components that can be used for IS composition.

Alongside with software and information components, the process components are also widely developed. Workflows supported by numerous software products on the market constitute the well known technological example of the kind. Processes implemented as workflows are important components of IS. OMG and WfMC [23, 26] efforts allowed to unify various process specification languages making possible to consider workflows as components for their interoperable reuse in IS.

What has been reached in the area of constructing IS from components. Mainly the methods and tools for component-based software construction are being developed. Traditionally mostly software components are considered out of the three kinds of components mentioned. Examples of such technologies include Microsoft's ActiveX and SunSoft's JavaBeans. Existing tools for programs development, such as Microsoft's Visual Basic, Inprise's Delphi and Protosoft's Power Builder provide for construction of programs using the pre-existing graphical components, including components of ActiveX and JavaBeans. A distinguishing feature of these tools is their orientation on non-complete component specifications and on the detailed knowledge of components capabilities by programmers. These facilities are good to work with "local" libraries assuming that developers know the components well. This approach generally is unsafe because due to the uncompleteness of specifications it is not possible to check the adequacy of components to the specifications of requirements. Well known example is the crash of the Ariane 5 rocket that happened due to the incorrect reuse of a program module.

Alongside with such technologies, Object Analysis and Design methods (OAD) introduce graphical notations and methodologies for IC design. The basic notation that dominate now is the Unified Modeling Language (UML)[12]. In essence, these methods implement conventional top - down development. Tools supporting the OAD methods (such as, e.g., Rational Rose by Rational Software and Paradigm Plus by Platinum) to be used for reuse require deep knowledge of the components used by developers. Practically this works well for program

libraries (e.g., use of graphical librariies in visual development tools). Quite often the tools mentioned are used in combination - analysis and design is performed applying OAD methods and system implementation - by means of visual programming tools (e.g., Delphi, Power Builder).

Therefore, the known methods and tools do not allow to use potential capabilities of components stored in the Internet, they are not scalable with respect to the number of the existing components, they do not allow to construct systems reusing components of different kinds (software, informational, process ones), they are not safe.

In this paper a new method of IS development that is called "compositional" is considered. The method is being developed in frame of the SYNTHESIS project and is oriented on elimination of the deficiencies of existing approaches. The SYNTHESIS method [7] is intended for correct composition of existing components semantically interoperable in the context of a specific application. In contrast to the system, technical interoperability (provided by CORBA) in SYNTHESIS the interoperability is considered in a broader, semantic aspect. Semantic interoperability means combination of several abilities: an ability to decide whether the existing components are relevant to an application, whether their context fits the application context, as well as to decide that a composition of pre-existing components will be consistent in the context of the application.

Main features of this method distinguishing it from the other known OAD and component-based development methods are the following:

- applying complete specifications assuring correctness of the existing components use in the systems being developed;
- use of repository (possibly, distributed) of complete specifications of the existing components;
- search in the repository for components relevant to the specification of requirements;
- identification of component fragments that can serve as refinements of the respective fragments of the specifications of requirements;
- resolving of various discrepancies in specifications of component fragments vs application specification fragments;
- provisions for compositions of reusable fragments into a specification concretizing the specification of requirements;
- support of justifying the correctness of the specification obtained applying the formal methods;
- constructing implementations of the information systems in the CORBA-like environment as interoperable compositions of wrappers implemented above existing components in the Internet (intranets).

2 Fundamentals of the compositional IS development method

The main distinguishing feature of the method considered is a creation of compositions of component specification fragments refining specifications of requirements. Widely used component-based development methods (e.g., JavaBeans) construct aggregates of components differently – just linking ports of components with each other or considering their interactions on the contractual basis [21].

Refining specifications obtained during the compositional development, according to the refinement theory, can be used anywhere instead of the refined specifications of requirements without noticing such substitutions by the users. The refinement methods [4] allow to justify the fact of a refinement formally to guarantee the adequacy of the specifications obtained to that of the required.

2.1 Overview of the basic features of the canonical model

The Ontological model, Requirement planning/Analysis model, Design model, Implementation model, Component (Information Resource) Specification model and respective macro layers are constituents of the SYNTHESIS CISD framework [7]. The semantics behind any of these models is provided by one and the same descriptive canonical object model treated in a semi-formal style and having a formal interpretation. The canonical model should provide for the integrated support of various functions, including (i) semi-formal representation of specification of requirements and analysis models of information systems; (ii) description of ontological contexts of application domains and justification of their coherence; (iii) uniform (canonical) representation of specifications of heterogeneous components; (iv) support of semantic reconciliation and adaptation of components to form their reducts and compositions serving as concretizations of analysis models of information systems.

To cope with that, the semi-formal canonical model should have a formal interpretation. Mapping of semi-formal specification into a formal one allows to develop a formal model of an application domain and components reflecting their static and dynamic properties. Formal specifications and respective tools create a basis for provable reasoning on the specification consistency and provable concretization of specification of requirements by pre-existing components compositions.

The SYNTHESIS language [16] defines specification-oriented semi-formal canonical model. Here we present the very basic canonical language features to make the examples demonstrating ideas of the Compositional Specification Calculus readable. It is important to note that the Specification Calculus considered does not depend on any specific notation or modeling facilities. Canonical model provides support of wide range of data - from untyped data on one end

of the range to strictly typed data on another. Untyped data are represented as *frames* that are used as symbolic models of some entities or concepts. The language uses frames to describe any entity, including the entities of the language itself, such as types, classes, functions, assertions. A frame at any moment of its life cycle can be declared to belong to an admissible class (class is a collection of typed objects). At that moment the frame becomes an *object.*

Typed data should conform to *abstract data types* (ADT) prescribing behaviour of their instances by means of the type's operations. ADT describes the syntactic interface of the type (its signature) and operands whose signatures define names and types of their operations and their specifications define the operation algorithms.

Type specifications are syntactically represented by frames, their attributes - by slots of the frames. Syntactically frames are inserted 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.

Each frame may be declared to belong to one or several classes (metaclasses). Such frame membership is denoted by a slot in :< class or metaclass name list >.

Syntactically, each functional (non-state) attribute of a type is defined by description of a function:

< function description >::=< function identifier >; in : function;
[params : {< formal parameter list >};][< specification >]}
< formal parameter identifier >::=< parameter sort symbol >< typed variable >
< parameter sort symbol >::= − | + |< empty >

The meaning of a parameter sort symbol is:

'+' - input parameter; '-' - output parameter; < empty > - input & output parameter.

A function is defined by a predicative specification stating its mixed pre/post conditions. To describe a state transition, it is necessary to distinctly denote the variables that define the state before and after the function execution. Variables referring to the state after function application are primed.

Formulae appearing in the predicative specifications below are described as follows. A variant of the multisorted first order predicate logic language is used in SYNTHESIS [16]. Each predicate, function, variable and constant in the formulae is typed. In simplified form formulae are either *atoms* or appear as:

w_1 & w_2 (w_1 and w_2)
w_1 | w_2 (w_1 or w_2)
$\neg w_2$ (not w_2)
$w_1 -> w_2$ (if w_1 then w_2)
$\exists$ x/t (w) (for some x of type t, w)
$\forall$ x/t (w) (for all x of type t, w)

where w, $\mathsf{w_1}$, $\mathsf{w_2}$ are formulae. Existential and universal quantifiers are denoted by ex and all, respectively.

2.2 Compositional specification calculus

2.2.1 Subtyping relation and type reducts

A signature Σ_T of a type specification $T =< V_T, O_T, I_T >$ includes a set of operation symbols O_T indicating operations argument and result types and a set of predicate symbols I_T (for the type invariants) indicating predicate argument types. Conjunction of all invariants in I_T constitutes the type invariant. We model an extension V_T of each type T (a carrier of the type) by a set of proxies representing respective instances of the type.

Among the type T operations we shall distinguish state attributes $Att_T \subseteq O_T$. Each state attribute is modelled as a function $V_T \to V_S$ where V_S denotes a carrier of the attribute type.

We assume the existence for our types of an abstract interpretation universe $\mathcal{V}$ (that is assumed to consist of all proxies of instances of any type). A type is a specification (intensional denotation) of a subset of elements in this universe. In accordance with the denotational semantics concepts, $\top$ is the *least informative* type denoting the whole universe $\mathcal{V}$ and $\bot$ is the *overdefined,* inconsistent type denoting the empty set. The subtype relation is the *information ordering* relation $\preceq$. Extensional interpretation of $\preceq$ is by the set inclusion. A subtype is considered to be more informative than its supertype (we assume a subtype relation definition based on type specifications similar to the given in [20]). Let $\mathcal{T}$ be a set of types with a subtype ordering relation $\preceq$. An extensional type semantics is an order homomorphism [3]:

$h : (\mathcal{T}, \preceq) \to (\mathbb{P}\,\mathcal{V}, \subseteq)$

In particular:

$h(\top) = \mathcal{V}, h(\bot) = \{\}$

and for all T, S in $\mathcal{T}$

$S \preceq T \Rightarrow h(S) \subseteq h(T)$

These considerations give us a general hierarchy for placement of type specifications. The guiding intuition behind the hierarchy is that type specifications are partial descriptions of real-world entities ordered by the information content.

h(T) gives a set of proxies V_T. Each proxy has interpretation in a certain domain D that also is a poset: $D, \preceq$. The bottom element $\bot_D$ is assumed to be defined for each instance domain such that for any $d \in D$, $\bot_D \preceq d$.

The domains (sets of type instance state values) are defined using complex sort constructors like cartesian product $(\times)$, powerset $(\mathbb{P})$, set comprehension $(\{x \mid x \in s \wedge P\})$, relational sort constructors $(s \leftrightarrow t)$, functional sort constructors $(s \longrightarrow t)$, etc.

Definition 2.1 Type reduct *A signature reduct R_T of a type T is defined*

as a subsignature Σ'_T *of type signature* Σ_T *that includes a carrier* V_T, *a set of symbols of operations* $O'_T \subseteq O_T$, *a set of symbols of invariants* $I'_T \subseteq I_T$.

This definition from the signature level can be easily extended to the specification level so that a type reduct R_T can be considered a *subspecification* (with a signature Σ'_T) of specification of the type T. The specification of R_T should be formed so that R_T becomes a supertype of T. We assume that only the states admissible for a type remain to be admissible for a reduct of the type (no other reduct states are admissible). Therefore, the carrier of a reduct is assumed to be equal to the carrier of its type.

Operations and invariants of the reduct subspecifications taken from the original type specification should be systematically modified. The occurences of the discarded attributes of the original type into the operations and invariants of the reduct should be existentially quantified and properly ranged. For the type T and its reduct R_T the formula is formed as follows:

$\exists\, t/T(r = t/R_T$ & <predicate with discarded attributes; each of the latter should be qualified by $T>$)

Here the typed variable r/R_T is assumed to be universally quantified.

2.2.2 Type refinement

Using the operation ρ of taking a reduct $R_T = \rho(T, O_r)$ of type $T \in \mathcal{T}$ a set of type reducts $\{R_T \mid T \preceq R_T \wedge O_r \subseteq O_T\}$ can be produced. Thus, decomposing a type specification, we can get its different reducts on the basis of various type specification subsignatures.

Taking into account that types we consider are characterized by their state and behaviour, we introduce definition of type refinement as follows.

Definition 2.2 *Type* U *is a* refinement *of type* T *iff*

- *there exists a one-to-one correspondence* $Ops : O_T \Leftrightarrow O_U$;
- *there exists an abstraction function* $Abs : V_T \to V_U$ *that maps each admissible state of* T *into the respective state of* U;
- $\forall\, x \in V_T \;\; \exists\, y \in V_U(Abs(x, y) \Rightarrow I_T \wedge I_U)$
- *for every operation* $o \in O_T$ *the operation* $Ops(o) = o' \in O_U$ *is a refinement of* o. *To establish an operation refinement it is required that operation precondition* $pre(o)$ *should* imply *the precondition* $pre(o')$ *and operation postcondition* $post(o')$ *should* imply *postcondition* $post(o)$.

2.2.3 Common reducts

Based on the notions of reduct and type refinement, a measure of common information between types in $\mathcal{T}$ can be established.

Definition 2.3 *A* common reduct *for types T_1, T_2 is such reduct R_{T_1} of T_1 that there exists a reduct R_{T_2} of T_2 such that R_{T_2} is a refinement of R_{T_1}. Further we refer to R_{T_2} as to a* conjugate *of the common reduct.*

Though taking a common reduct for two types is not tractable, the following simple heuristics show what can be reasonable approaches for that. Operations of the type T_1 are suspected to belong to its common reduct with T_2 if operations with the equal signatures can be found in T_2 modulo variable renaming, parameters ordering and parameter type redefinition (contravariant for the argument types and covariant for the result types type differences for T_2 are acceptable). A specification of an operation of T_1 to be included into the resulting reduct is chosen among such pre-selected pairs of operations of operand types if the operations in a pair are in a refinement order (for the common reduct (resulting supertype) more abstract operation should be chosen). If the pre-selected operations are not in a proper refinement order then they are considered to be different operations and will not be included into the common reduct.

Definition 2.4 *A* most common reduct $R_{MC}(T_1, T_2)$ *for types T_1, T_2 is a reduct R_{T_1} of T_1 such that there exists a reduct R_{T_2} of T_2 that refines R_{T_1} and there can be no other reduct $R^i_{T_1}$ such that $R_{MC}(T_1, T_2)$ is a reduct of $R^i_{T_1}$, $R^i_{T_1}$ is not equal to $R_{MC}(T_1, T_2)$ and there exists a reduct $R^i_{T_2}$ of T_2 that refines $R^i_{T_1}$.*

Reducts provide for type specification decompositions thus creating a basis for their further compositions.

2.2.4 Type compositions

We introduce here type composition operations that can be used to infer new types from the existing ones.

Let $T_i (1 \leq i \leq n) \in \mathcal{T}$ denotes types.

Definition 2.5 Type meet *operation. An operation $T_1 \sqcap T_2$ produces a type T as an 'intersection' of specifications of the operand types. Generally the result T of the meet operation is formed as the merge of two most common reducts of types T_1 and $T_2 : R_{MC}(T_1, T_2)$ and $R_{MC}(T_2, T_1)$. The merge of two reducts includes union of sets of their operation specifications. If in the union we get a pair of operations that are in a refinement order then only one of them, the more abstract one is included into the merge. Invariants created in the resulting*

type are formed by disjuncting *invariants taken from the most common reducts specifications being merged.*

If T_2 (T_1) is a subtype of T_1 (T_2) then T_1 (T_2) is a result of the meet operation. Type T is placed in the type hierarchy as an immediate supertype of the meet arguments types and a direct subtype of all common direct supertypes of the meet argument types.

Meet operation produces a type T that contains common information contained in types T_1 and T_2.

Definition 2.6 Type join *operation. An operation $T_1 \sqcup T_2$ produces type T as a 'join' of specifications of the operand types. Generally T 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. The common elements are determined by another merge - the merge of conjugates of two most common reducts of types T_1 and T_2 : $R_{MC}(T_1, T_2)$ and $R_{MC}(T_2, T_1)$. The merge of two conjugates includes union of sets of their operation specifications. If in the union we get a pair of operations that are in a refinement order then only one of them, the more refined one (belonging to the conjugate of the most common reduct) is included into the merge. Invariants created in the resulting type are formed by* conjuncting *invariants taken from the original types.*

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 hierarchy as an immediate subtype of the join operand types and a direct supertype of all the common direct subtypes of the join argument types.

2.2.5 Type lattice and properties of compositions

Partially ordered by a subtyping relation set of types $\mathcal{T}$ with $\top$ and $\bot$ types is a *lattice:* for all $S, U \in \mathcal{T}$ there exists least upper bound (l.u.b.) and greatest lower bound (g.l.b.). Meet $T = S \sqcap U$ produces T as the g.l.b. for types S, U in the $(\mathcal{T}, \preceq)$ lattice (in case when S, U have no common reducts, meet results in $\top$). Join $T = S \sqcup U$ produces T as the l.u.b. for types S, U in the $(\mathcal{T}, \preceq)$ lattice (in case when S, U have no common reducts, join results in $\bot$).

$< \mathcal{T}; \sqcap, \sqcup >$ is an algebra with two binary operations. For such algebra the following properties are established:

1. Commutativity: $S \sqcap U = U \sqcap S$ and $S \sqcup U = U \sqcup S$
2. Associativity: $S \sqcap (U \sqcap S) = (S \sqcap U) \sqcap S$ and $S \sqcup (U \sqcup T) = (S \sqcup U) \sqcup T$
3. Idempotence: $S \sqcap S = S$ and $S \sqcup S = S$
4. Absorption: $S \sqcap (S \sqcup U) = S$ and $S \sqcup (S \sqcap U) = S$

Two another rules relate subtyping with operations meet and join:
$S \sqcap U = U \equiv S \preceq U$
$S \sqcup U = S \equiv S \preceq U$
as well as:
$S \preceq S \sqcap U$ and $U \preceq S \sqcap U$;
$S \sqcup U \preceq S$ and $S \sqcup U \preceq U$.

2.3 Ontological context reconciliation

Specifications of information components and specifications of requirements must be associated with ontological contexts containing concepts of the corresponding subject areas. Ontological concepts are described by means of the canonical model of the SYNTHESIS language. It should be stressed that the context of the shared ontology must be described by specifications in the canonical model. For this purpose, for the most popular models for representation of ontologies mappings to the canonical SYNTHESIS model are developed.

The ontological concepts have their verbal definitions and descriptor lists. Verbal definitions are similar to word definitions in an explanatory dictionary. Descriptor lists included in the specifications of ontology concepts are built on the basis of the meaningful words in the concept's verbal definition. Tools for lexical and morphological analysis are used in the process of building descriptor lists. Normalized words or word stems can be used as descriptors. Concept descriptors are necessary for establishing preliminary relation to other concepts lying outside the given ontological context.

Generalization/specialization (concept/subconcept relations) and positive relations (synonymies) may be defined between ontological concepts. These relations can be fuzzy; i. e., they can be characterized by degree (force) in the range between 0.0 and 1.0. If the value of relation is not defined explicitly it is implied to be equal 1.0.

To compare components and specifications of requirements schemes on the basis of the ontological information available, it is required to reduce local ontological contexts of the components and of the specifications of requirements to the same ontological context. For this purpose, local ontological contexts of the components and of the specifications of requirements are mapped into the shared ontology concepts [8]. Relations between concepts of different contexts are established by calculating the correlation coefficients between concepts on the basis of the verbal definitions.

3 SYNTHESIS compositional design method prototype architecture

Fig.1 shows a general structure of the SYNTHESIS compositional design method prototype.

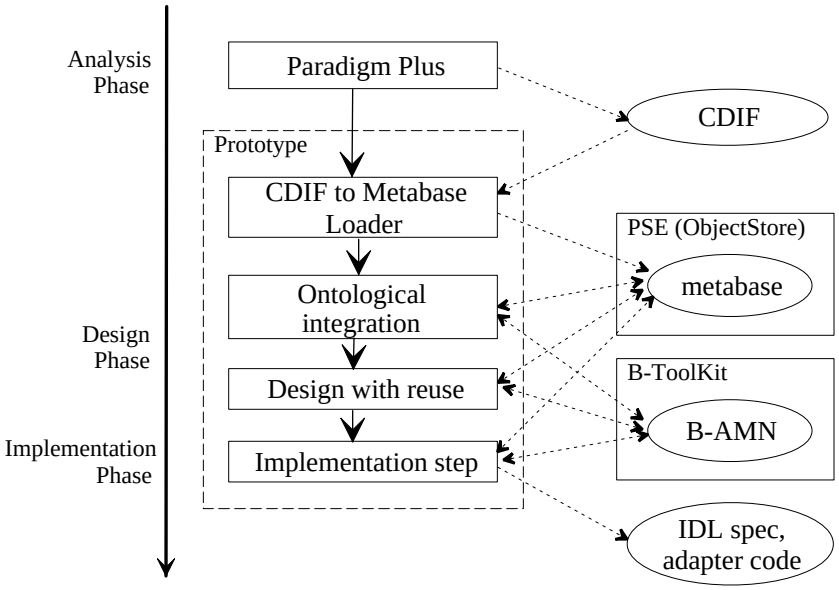

Figure 1: SYNTHESIS Compositional Design Method Prototype Structure

We choosed the ParadigmPlus [24] to support the Requirement Planning and Analysis phases (the SYNTHESIS tool is independent on this choice). The UML notation augmented with SYNTHESIS specific features, such as ontological specifications and predicative specifications of operations is used. Actually any standard representation facilities can be suitable (as the UML subset with some extensions) but the approach is different from conventional one, it is the canonical model that has primary importance providing proper interpretation and meaning for graphical representation.

Using the CDIF representation [9] taken out of the ParadigmPlus, the specifications of the analysis phase are loaded into the metaobject repository supported by the PSE ObjectStore.

For formal modeling the B Abstract Machine Notation is used [1] that together with B-Toolkit [2] provides for type specification consistency check, adequacy of component specifications [15], establishment of a refinement condition, generating and proving respective proof obligations.

SYNTHESIS prototype is implemented using Java 2 under Windows NT 4.0. The tool is oriented on work in a CORBA-like interoperable environment: the tool generates an IDL specifications and CORBA-based adapters code.

4 Design phase

Design is the component-based process of concretization of a specification obtained on an analysis phase by an interoperable composition of pre-existing information components. It includes integration of application domain and of information resource ontological contexts establishing the ontological relevance of constituents of requirements and components specifications, identification of component types (classes) and their fragments suitable for the concretization of an analysis model type (class) capturing its structural, extensional and behavioural properties, composition of such fragments into specifications concretizing the requirements, justification of a property of concretization of requirements by such compositions.

4.1 Ontological integration

4.1.1 Mapping component contexts into the shared ontology

Integration of contexts of ontological descriptions of components and specifications of requirements is based on the mapping into a shared ontology. Integration at the level of verbal descriptions is based on the analysis of concept descriptors performed with the aim of mapping concepts of one ontological context into the other. For this purpose, the degree of relation between concepts of two ontological contexts is calculated using the vector-space approach [25].

The analysis of a descriptor begins with the estimate of its significance in the definition of particular concepts. Every descriptor is assigned a weight that takes into account the frequency of its occurrence in the definition of a particular concept and the number of concepts in the context whose definitions include this descriptor. The more the frequency and the less the number of concepts defined with the help of this descriptor the more its significance and, thus, its weight. Let X and Y be concepts of different ontological contexts (local context and shared ontology). Let V_X and V_Y be the vectors consisting of descriptors that define the corresponding concepts in X and Y. For V_X and V_Y vectors C_X and C_Y are generated that contain lists of weights W_{Xk} and W_{Yk} for every descriptor k that participates in the definition of X and Y, respectively. The weights are calculated by the following empirical formulas [25]:

$$W_{Xk} = \frac{(1 + \frac{f_{Xk}}{f_{max}}) \cdot \log \frac{N}{n_k}}{\sqrt{\sum_{i \in V_X} ((1 + \frac{f_{Xi}}{f_{max}}) \cdot \log \frac{N}{n_i})^2}} \quad (1)$$

$$W_{Yk} = \frac{f_{Yk} \log \frac{N}{n_k}}{\sqrt{\sum_{i \in V_Y} (f_{Yi} \log \frac{N}{n_i})^2}} \quad (2)$$

where f_{Xk} and f_{Yk} are the frequencies of the occurrence of the descriptor k in V_X and V_Y respectively, f_{max} is the maximal frequency of descriptors in V_X or V_Y, N is the total number of concepts in the shared ontology, and n_k is the number of concepts in the shared ontology whose vector V_Y includes the descriptor k. The first factor in the product increases the significance of descriptors that are frequently mentioned in the definition. The second factor increases the significance of descriptors that occur in a lesser number of vectors V_Y. Frequencies in V_X are reduced to the interval [0.5,1.0], since every descriptor of a local ontology concept is important for finding corresponding concepts in the shared ontology.

The weights W_{Xk} and W_{Yk} are normalized to eliminate the dependence on the difference in the length of vectors for different X and Y. If dictionaries or thesauruses containing weight coefficients of words are available, other approaches to determining descriptor weights can be used.

The functions for estimating the correlation between ontological concepts are defined as follows [25, 8]:

$$sim(X, Y) = \frac{\sum_{k=1}^{t}(W_{Xk} \cdot W_{Yk})}{\sqrt{\sum_{k=1}^{t}(W_{Xk})^2 \cdot \sum_{k=1}^{t}(W_{Yk})^2}} \tag{3}$$

$$r(X, Y) = \frac{\sum_{k=1}^{t} min(W_{Xk}, W_{Yk})}{\sqrt{\sum_{k=1}^{t}(W_{Xk})^2}} \tag{4}$$

$$r(Y, X) = \frac{\sum_{k=1}^{t} min(W_{Xk}, W_{Yk})}{\sqrt{\sum_{k=1}^{t}(W_{Yk})^2}} \tag{5}$$

The range of values of the function $sim(X, Y)$ is the real interval [0.0,1.0]. A value of 0.0 means that the concepts are not related to each other, and a value of 1.0 means that the concepts are identical (have identical lists of descriptors). The concept X is considered correlating (similar) to the concept Y if $sim(X, Y)$ is greater than a certain threshold value ℓ; in this case, the value of the function is considered a measure of similarity (force of the relation).

The functions $r(X, Y)$ and $r(Y, X)$ are used to find possible relations of the kind concept/subconcept between different contexts. For these functions to give correct results, the weights of descriptors must be normalized. The above method for weight calculation satisfies this requirement. If $r(X, Y)$ and $r(Y, X)$ are less than a certain threshold value ℓ, the concepts X and Y are not related to each other. If both the values of $r(X, Y)$ and $r(Y, X)$ are greater than ℓ, the concepts X and Y are positively associated with each other, and the correlation coefficient (force of the relation) is the minimum of these values.

If $r(X, Y)$ is greater than ℓ while $r(Y, X)$ is less, then X is a superconcept of Y, i. e., the generalization association is established between X and Y. In this case, the correlation coefficient (force of the relation) is equal to $r(X, Y)$. On the other hand, if $r(X, Y)$ is less than ℓ and $r(Y, X)$ is greater, then X is a subconcept of Y, i. e., the specialization association is established between X and Y. In this case, the correlation coefficient is equal to $r(Y, X)$. If necessary, the results of the automatic mapping of one context into another can later be refined manually by an expert.

4.2 Ontological integration of schemes

After the component ontology contexts have been mapped into the shared ontology, we pass to the ontological integration of schemes. The main purpose of this stage is to detect (among the descriptions of information components) types, classes, and their fragments that are relevant to the specifications of requirements.

Three types of ontological contexts (modules) are involved in the integration process.

- The application ontology module (AOM) contains the ontological definitions for specifications of requirements;
- The resource ontology module (ROM) contains ontological definitions for specifications of components;
- The common ontology module (COM) contains the shared ontology of the particular subject area;

The results of the integration stage of the specifications of requirements and components contexts (AOM and ROM) with the shared ontology (COM) with regard to verbal definitions are presented as positive associations and concept/subconcept associations between the local application and components contexts with the shared ontology of the subject area. To perform the preliminary element integration of the specifications of requirements with specifications of the available components, it is required to find relations between AOM and ROM contexts. These relations can be established on the basis of internal generalization associations and internal positive relations in COM in conjunction with the established intercontext relations that join the ontology concepts of components and requirements with the concepts in COM.

The concepts in ROM that have positive or specialization associations with a concept in AOM (i. e., those that are synonymous with the AOM concept or are its subconcepts) can be related to the AOM concept in question. To reveal such relations, the correspondence paths of AOM and ROM concepts must be analyzed and the concept graph must be complemented with the missing relations.

The search for new relations is performed with the help of the algorithm described in [10] (this algorithm uses the transitivity property of relations). Two sequential positive relations give a new positive relation characterized by the force equal to the product of the forces of the given relations. If a sequence of two relations includes a specialization relation, then the force of the resultant relation equals the minimum of the two given relations, and the kind of this relation depends on the kinds of the relations in the path. If the path contains at least one positive relation, then the resultant relation will be positive as well. If the path consists entirely of specialization relations, then the resultant relation will be a specialization one. In order to calculate the force of a path that includes relations of various kinds, it is necessary to find, first of all, the forces of its subpaths consisting only of synonymy relations; then, the forces of fragments that include generalization/specialization relations can be calculated as well. The purpose of the algorithm is to find the maximum value of the relation force between two given concepts. Relations with forces not exceeding the threshold value ℓ are neglected. The same value ℓ that was used at the stage of the mapping of ontological contexts at the verbal level may be used as the threshold.

The results of this algorithm are used to find correspondences between the specification elements. For this purpose, the concept of the weak ontological relevance of specification elements is introduced.

Definition 1. The element I_r of an information component specification is called ontologically weakly relevant to the element I_s of the specifications of requirements of the same kind (type, class, function, attribute, and so on) if I_r is related to a concept C_r, I_s is related to a concept C_s, and there exists a positive relation between C_r and C_s, or C_r is a subconcept of C_s. This can be written as

$$weak\_relevance(I_r, I_s) \Leftrightarrow (\exists\, C_r, C_s :$$
$$inst\_of\,(I_r, C_r) \wedge inst\_of\,(I_s, C_s) \wedge$$
$$\wedge (positive(C_r, C_s) \vee subconcept(C_r, C_s)))$$

In the subsequent development of the information system, the results of the ontological integration of schemes are used (before composing its general scheme) to preliminarily relating elements of component scheme specifications to the specification elements of requirements.

4.2.1 GUI for mapping to COM concepts

Fig. 2 shows the GUI for confirmation of mapping of components and requirements concepts into common concepts. The GUI allows the developer to correct the results (a list of associastions between components/requirements concepts and common conepts) of automatic mapping to COM concepts.

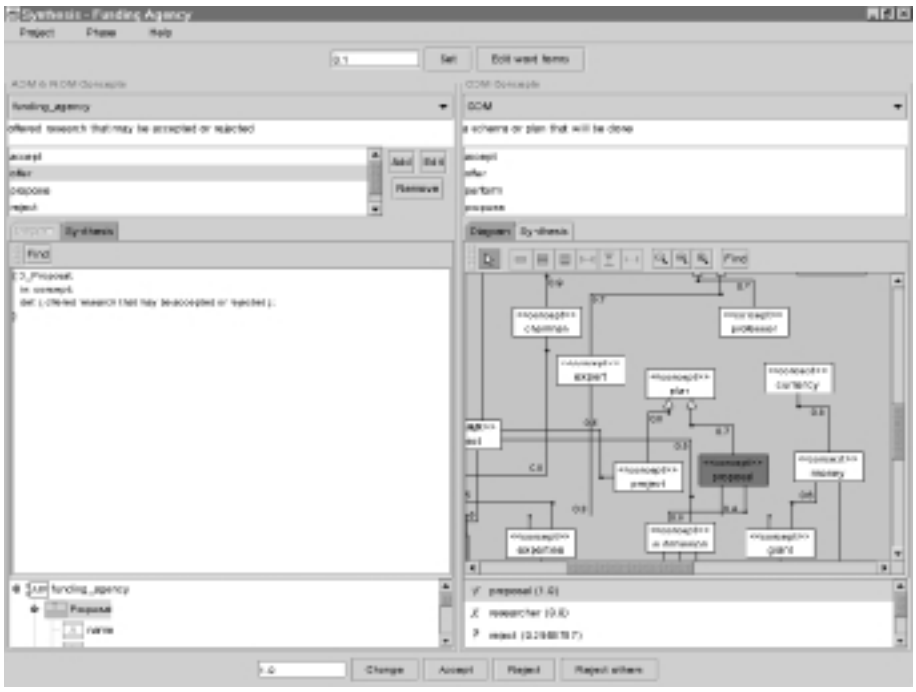

Figure 2: GUI for Mapping to COM Concepts

Expert should accept or reject each finded association. Expert also may establish a new associations (between components/requirements concepts and common conepts) not included into the list.

4.2.2 GUI for confirmation of relevance of requirements and components elements

Fig. 3 shows the GUI for confirmation of relevance of components and requirements elements. It serves to reduce the number of automatically found ontological relevance relations between requirements and components elements.

For each pair of found requirements and components elements the expert should accept or reject their relevance.

4.3 Structural conflict resolution

On construction of the most common reduct of types defined in the specification of requirements and in a component, the various conflicts between the specifications (structured conflicts, value conflicts, behavioral conflicts) should be discovered and resolved. The result of the conflicts resolution is represented

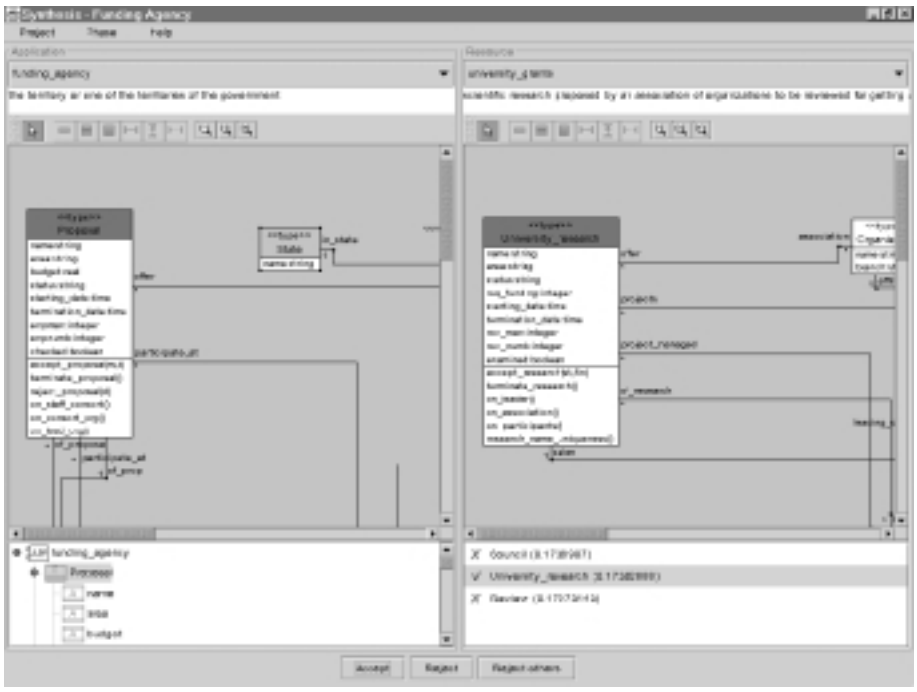

Figure 3: GUI for confirmation of relevance of requirements and components elements

in a transformation of a conjugate reduct into a concretizing reduct. The concretizing reduct includes together with the attributes of a reduced type the mapping of reduct and concretizing reduct attributes and the conflict resolution functions.

The most common reducts and their concretizing reducts are specified as SYNTHESIS types. The specific metaslots and attributes are used to represent the specific semantics. The following example shows definition of the `Proposal` reduct intended as a common `Proposal - Submission` reduct.

```
{R_Pr_Su;
    in: reduct;
      metaslot
        of: Proposal;
        taking: {name, area, consortium, budget};
        c_reduct: CR_Pr_Su;
      end
}
```

A list of attributes of the reduced type in the slot `taking` contains names of its attributes that are to be included into the reduct. A starting list of attributes

is taken from the common signature reduct obtained previously. Further the list can be properly adjusted.

A slot `c_reduct` refers to the concretizing reduct based on a resource type. We add a concretizing reduct (above types of pre-existing components) as the following type definition:

```
{CR_Pr_Su;
  in: c_reduct;
    metaslot
    of: submission;
    reduct: R_Pr_Su;
    end;

  simulating: {
    R_Pr_Su.name             == CR_Pr_Su.name &
    R_Pr_Su.area             == CR_Pr_Su.research_area &
    R_Pr_Su.consortium       == CR_Pr_Su.participants &
    R_Pr_Su.budget           == CR_Pr_Su.req_money &
    R_Pr_Su.starting_date    == CR_Pr_Su.starting_date &
    R_Pr_Su.termination_date == CR_Pr_Su.f_termination_date };

  f_termination_date: {in: function;
    params: {+s/Submission, -return/Proposal.termination_date}
    {{ return = s.starting_date+s.duration }}}
}
```

In this case a slot `reduct` refers to the reduct based on an analysis model type. The predicate `simulating` shows how the concretizing state is mapped to the reduct state.

`R_Pr_Su.area = CR_Pr_Su.research_area` defines that values of attribute `area` of reduct `R_Pr_Su` are taken from values of attribute `research_area` of concretizing reduct `CR_Pr_Su`. `f_termination_date` presents the mediating function resolving the conflict.

The conflict resolution in SYNTHESIS method is based on a combination of two approaches - using a set of predefined rules of structural transformations [18], and using a high-level language for transformation specifications [19]. A coorespondence between elements of specifications (types, attributes, functions) is set as a result of ontological specifications integration. Using a language of object calculus for specification of the mediating functions, expert may specify functions for resolving various conflicts. A predefined set of rules is used for resolving structural conflicts to help the construction of mediating functions between specifications of components and application requirements.

As an example we give one of the rules of path conformance. Other rules are similar to those considered in researches on database schemas integration [18].

Rule of path conformance: Structural abstraction.

The application path $Ts_0..Ts_1$ is conformant to resource path $Tr_0..Tr_1$, if

a) application path has the form: $Ts_0 - a1 -> Tx_1..Ts_1$

b) resource path has the form: $Tr_0 - b0 -> Tr_3 - b1 -> Ty_1..Tr_1$

c) application type Tx_1 is relevant to resource type Ty_1

d) application path $Tx_1..Ts_1$ is conformant to resource path $Ty_1..Tr_1$

where

$Ts_0 - a1 -> Tx_1$ – single-valued attribute having a user-defined type (Ts_0 has an attribute $a1$ of type Tx_1)

$Tx_1..Ts_1$ – any path (may be with zero length)

An example of mediating functions for resolving structural conflicts for paths $Organization - in\_city -> String$ $Company - address -> Address - city -> String$ is:

```
get_in_city: {in: function;
params: {+r/Company, -returns/Organization.in_city}
{{ returns' = r.address.city }}}
```

4.3.1 Conformant paths detection algorithm

The main goal of the algorithm is to find the conformant paths in application and resource type composition schemas. The conformant paths are used to resolve the structural conflicts in application and resource schemas. The search is based on the list of relevant elements obtained after the ontology integration step. The process also allows to find additional relevant elements missed in the ontology integration step.

Briefly, the algorithm of finding the conformant paths is the following.

1. Take an arbitrary pair - application type Ts_0 and resource type Tr_0 from the list of ontologically relevant types.

2. Try to find application type Ts_1 and resource type Tr_1 such that the paths $Ts_0..Ts_1$ and $Tr_0..Tr_1$ meet one of the rules of paths conformance (see example for one such rule below). If a rule has been detected then the tool requests a developer (item 3) otherwise the tool goes to the first item.

3. If types Tx_1 and Ts_1 or Ty_1 and Tr_1 are not the same, a developer is requested if these types are relevant or not. If the developer accepts the relevance of types Tx_1 and Ts_1, and of types Ty_1 and Tr_1, these pairs are added to the list of ontologically relevant types.

4. If types Tx_1 and Ts_1, Ty_1 and Tr_1 are relevant or the same, the developer is requested if the paths $Ts_0..Tx_1$ and $Tr_0..Ty_1$ are conformant or not. If the developer accepts the conformance of paths $Ts_0..Tx_1$ and $Tr_0..Ty_1$ (Fig. 4), these pairs are added to the list of conformant paths.

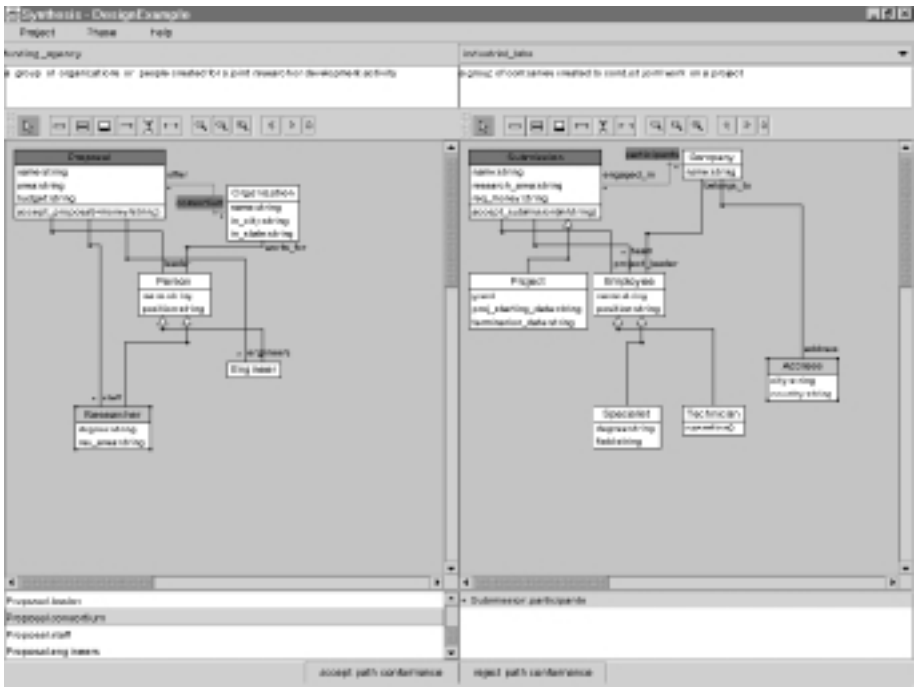

Figure 4: GUI for confirmation of paths conformance

4.3.2 GUI for confirmation of paths conformance

The tool automatically generates a list of conformant paths which the developer should accept or reject using the GUI shown in Fig. 4.

4.4 Most common reduct identification

For each pair of type specifications Ts and Tr (each Tr should be ontologically relevant to Ts) we try to construct their common reduct. We start with identification of their *common signature reduct.* This procedure takes into account only signatures of types ignoring their complete specifications. After that most common reducts are constructed. This makes possible to identify the common reduct for Ts, Tr and the respective *concretizing* reduct of Tr that can be imagined as a conjugate of a common reduct that incorporates also necessary conflict reconciliation with Ts specifications. Finally, if required, we can justify the concretizations constructed so far by formal proofs.

Here and further the notion of *concretizing* specification (of reduct, type, class) means that the refining specification includes required mediating functions resolving value, structural or behavioural conflicts with the respective

specification of the analysis level.

To identify common signature reducts, we should find for each pair of ontologically relevant types Ts, Tr a maximal collection A of pairs of attributes (a^i_{Ts}, a^j_{Tr}) that are also ontologically relevant and satisfy the type constraints so that a^j_{Tr} could be reused as a^i_{Ts}. General approach to form A for the pair of ontologically relevant types Ts and Tr is the following:

1. All ontologically relevant pairs of immediate state attributes (a^i_{Ts}, a^j_{Tr}) of the types belong to A if type of a^j_{Tr} is a subtype of type of a^i_{Ts}. This requirement can be completely checked for built-in attribute types. For the user defined types a pair of ontologically relevant attributes is conditionally included into A: final check is postponed until user defined attribute types relationship will be clarified.

2. All ontologically relevant pairs of immediate functional attributes (a^i_{Ts}, a^j_{Tr}) of the types belong to A if signatures of functions a^i_{Ts} and a^j_{Tr} satisfy the following requirements:

   (a) they have equal numbers of input parameters and of output parameters pairwise ontologically relevant;

   (b) for each ontologically relevant pair of input parameters their types should be in a contravariance relationship;

   (c) for each ontologically relevant pair of output parameters their types should be in a covariance relationship.

3. A pair of immediate attribute a^i_{Ts} of type Ts and of immediate attribute $a^j_{Tr'}$ of a component type Tr belongs to A if they do not satisfy conditions 1) or 2) above but Ts, Tr, a^i_{Ts}, $a^j_{Tr'}$ are included into reusable structures (paths) suggested by the reusable path detecting rules.

   Structural conflicts deserve specific attention: they are resolved in a process of reusable path detecting process (4.3.1). This process resembles the processes developed for the database schema integration approaches [18].

On analysis of attribute pairs we recognize and resolve the various conflicts between the application and component type specifications. This resolution is reflected in specification of a concretizing reduct that is a common reduct mediated to the specification of requirements level.

For common reducts the refinement condition is established using mapping of participating types and their reducts into AMN, specifying and proving the respective refinement machines. The mapping and proving procedure used for that were illustrated in [5].

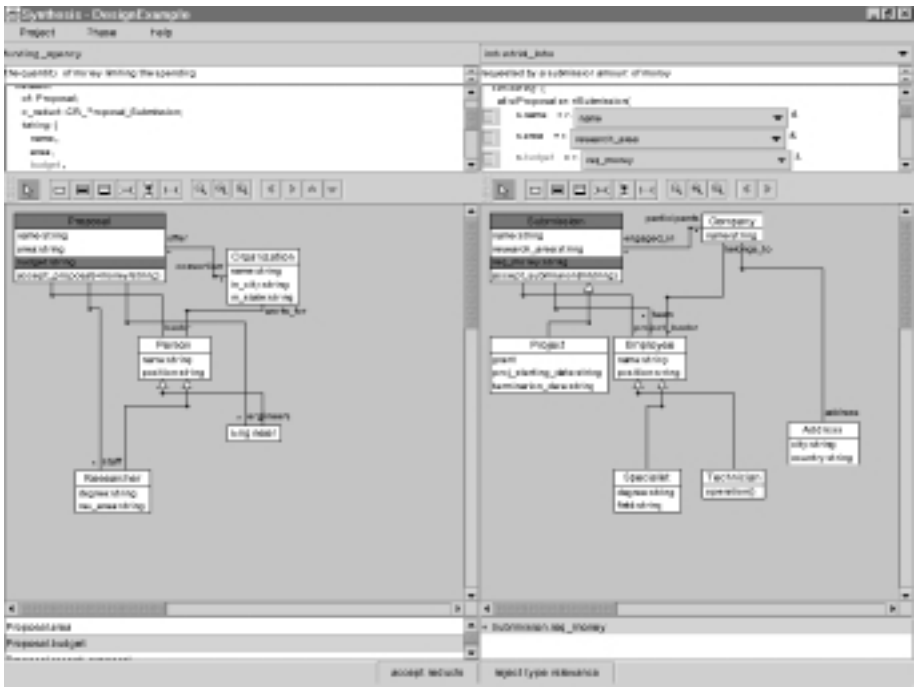

Figure 5: GUI for reducts construction

4.4.1 GUI for most common reducts construction

For each pair of the relevant application type *Ts* and component type *Tr* a developer should refine the common reduct and its concretizing reduct definitions. The developer may edit these specifications using the GUI shown in Fig. 5.

The tool generates the initial filling of this form. Tool detects the conflicts and constructs the corresponding mapping functions. For structural conflicts the specifications of mapping functions are generated. For other conflicts tool fill the mapping function specification with helping information.

4.5 Composition of specification fragments

The composition of discovered fragments (reducts) into specification concretizing the requirements is realized by applying the operations of type composition (meet and join) and by construction of view over classes [13].

An example of using operation `join` over types `Proposal` and `Submission` is:

```
IProposal = Proposal[name, budget, status] |
            Submission[name, req_money/budget, agency_name]
```

Views are defined by functions having only one return parameter: *returns/ < class_name > as_class*. The function specify a rule of view formation. The view formation rule defines the class composition by operations over classes, such as union, intersect, join. The specification of view is the following:

```
{<view name>;
    in: class;
    metaslot
        view_gen:{ in: function;
            params: {-returns/<view name> as_class};
            enforcement: on_access;
            {{ <view formation rule> }}
            }
    end;
}
```

4.5.1 Process of composition

Depending on the kind of a system being developed the various scenarios of construction of concretizing specifications are applied.

Developing of software systems, only types are specified. The reduct compositions are constructed in order that as much attributes of a type of specifications of requirements could be covered as possible. The composition of reducts is realized by `join` operation. If some attributes remain uncovered, new type including such attributes is created. Later this type should be implemented by the developer.

Developing of information systems, classes are the main entities considered. Depending on the requirements, such operations over classes as, e.g., `union` or `join` are used. For `union` the instance type of the constructed view is formed by `meet` operation over instance types of the classes - arguments of the operation. For `join` the instance type of the constructed view is formed by `join` operation over instance types of the classes - arguments of the operation.

4.5.2 GUI for composition of specification fragments

Fig 6 shows a GUI for construction of compositions over types and classes.

5 Semi-structured data in compositional development

Term 'semistructured data' denotes data representations possessing some of the following characteristics [11]:

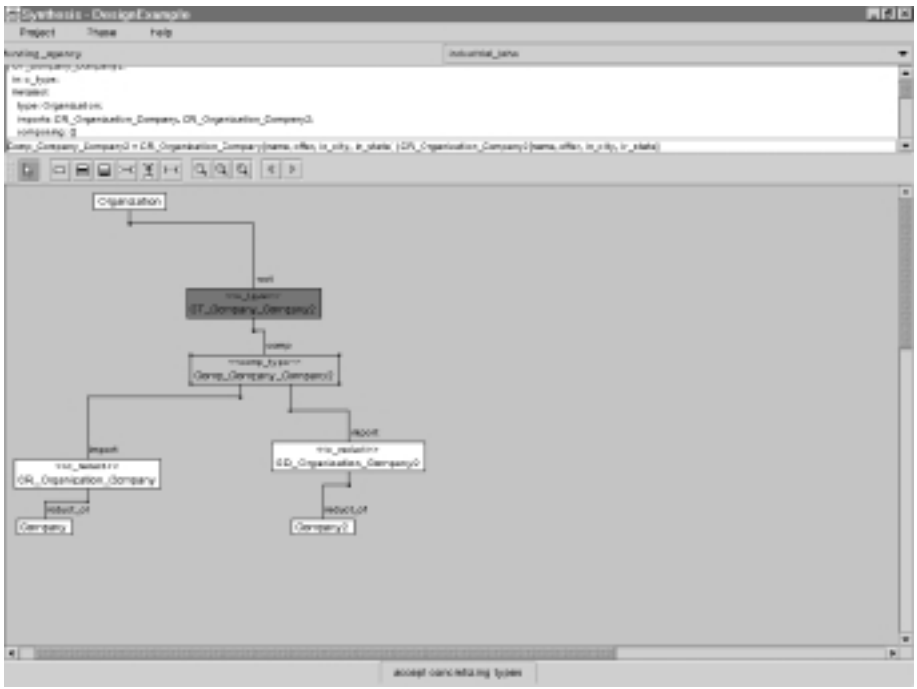

Figure 6: GUI for reducts construction

1. the schema of data is not given in advance and may be implicit in the data,
2. the schema is relatively large (w.r.t. the size of the data) and may be changing frequently,
3. the schema is descriptive rather than prescriptive, i.e., it describes the current state of the data, but violations of the schema are still tolerated,
4. the data is not strongly typed, i.e., for different objects, the values of the same attribute may be of different types.

To work with semistructured data the following facilities of the canonical model can be used:

- collections of self-defined objects or collections of frames (worlds) without pre-defined associations;
- worlds with pre-defined frame associations;
- classes containing partially typed objects self-defining their own individual attributes that were not specified in a type of the class instances;
- classes of aggregates (associations of objects and frames);

- type *union*.

Two various sets of semi-structured data modeling facilities are considered in this section. The first set contains facilities that are intrinsically semi-structured. These are frame-based facilities. Another set contains specific data types that are characteristic for semi-structured data. Union and Link data types are representatives of this facility class.

Compositional development works on the level of types, not on the level of type instances. Frame type leading to partially typed world instances, Union type, Link type are used in compositional development. An example of compositional development over Web sites for the purpose of personalization is given in [17]. To involve these types into the compositional development a refinement condition should be defined for these types. These conditions are defined below after brief introduction of each type.

5.1 Frame types

SYNTHESIS has been developed as the hybrid (structured/semi-structured) data model [16]. Frames provide facilities for unstructured and semistructured data and objects - for the structured ones.

A frame is considered to be a symbolic model of a certain entity or a concept. A frame is represented as an ordered collection of attributes called slots used for description of properties of an entity or a concept. To each slot a collection of values can be associated; each value being generally an abstract value of arbitrary type defined in the SYNTHESIS language (another frame may be used for that value). Any frame can acquire a unigue unmodifiable identifier. In the frame language a frame base is considered as a collection of frames interrelated by means of binary associations. Frames appear as objects that exist autonomously, without types and classes to describe and create them. To deal with frames as an unordered collections of attributes, specific kind of frames called frame_set has been added.

Generally, frames are abstract values, their type (Tframe) is a subtype of the abstract value type Taval. 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 containing a corresponding 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.

Frames 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.

5.1.1 Partially typed objects

Objects in SYNTHESIS 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 (types can be used for their definitions if required).

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 respective 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 information about information resources. In particular, the frame language is used for the representation of unstructured or semistructured information about application domain entities (including textual data). Such information can be keeped separately or can be related directly to objects on the basis of object - frame associations.

Path expressions in SYNTHESIS may correspond to database path that may involve structured or semistructured data only, or there may be crossover points to navigate from structured to semi-structured data or vice-versa.

5.1.2 Frame applications

To work with semistructured data on the Web, each Web site page may be represented in SYNTHESIS by a frame embedding sub-frames representing the fragments of the page. An instance of a world is a collection of frames that can be reachable through hypertext-based frame relationships from the 'root' frame referenced by an URL. The frames involved can be represented differently (as frames with slots or without slots or combining both approaches).

Such world is interpreted as a labeled graph with nodes containing frames (that look as aggregates (or can be structured as those), as semistructured objects or as unstructured data) and arcs formed by hypertext links. In SYNTHESIS attributes of structured types can be typed with the frame type and frame slots can be instantiated with any SYNTHESIS type values (dynamic type capability).

For XML a DTD-less document can be interpreted as a frame with the name reflecting unique name of the document. Each element is a slot with a name corresponding to the element tag and value - to the element content. Attributes are represented as slots of the metaslot related to the attributed element. In another possible interpretation each element is a frame with a name correspond-

ing to the element tag and frame content - to the element content. Attributes are represented as slots of metaframe related to the attributed element frame. IDREF is mapped into the slot having Link properties.

5.1.3 Frame types

Frames are partially typed with specifications of binary associations that can be used for interrelating frames of the respective world. To set an association between frame A and 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.

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

{< frame identifier >;
in : 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: {< formula >}. An absence of the slot domain or the slot range means that the corresponding 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).

5.1.4 Frame types refinement

A binary association type is a relation type $A \subseteq D \times R$ where D and R are sets of instances of types T_d and T_r, respectively. This relation should conform to the association type predicate denoted further as P_A and imposing a constraint on set of pairs $(d, r) \in A$ where $d \in D, r \in R$.

Definition 5.1 *Association type refining A is a relation $A' \subseteq D' \times R'$ where D' and R' are sets of instances of types $T_{d'}$ and $T_{r'}$ refining types T_d and T_r, respectively. This relation should conform to the refining association type predicate denoted as $P_{A'}$ and imposing a constraint on set of pairs $(d', r') \in A'$ where $d' \in D', r' \in R'$.*

A and A' should conform to the following refining invariant:

$$\forall\, a \in A\, \exists\, a' \in A'(P_A(a) \wedge P_{A'}(a') \wedge ref(d, d') \wedge ref(r, r')),$$

where a denotes (d, r) pair, a' denotes (d', r') pair, ref is a refining relation defining how values of an association domain and range are mapped into the refining values.

Definition 5.2 *Frame type F'* refines *frame type F if for each binary association type A in F there exists binary association type A' in F' that refines A.*

Using this definition, operations of taking common reducts and making meet and join compositions are applicable to the frame types.

5.2 Object (frame) links modeling

Links lead to object (frame) sharing. Links are considered to be unique in the hierarchical context. Attributes (slots)that are links have to be "double typed":
1) as a usual attribute;
2) as a link that may have various semantics (e.g., URL, URI, Ref, etc.).

To cope with that links are introduced as members of the respective metatypes (e.g., Link, URL, URI, etc.)

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

If an attribute is specified as own object then the corresponding component object is an ownership 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 corresponding 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:

< attribute identifier >:< class identifier >,

then it means that the component is completely independent object and deletion of a value of this ADT does not lead to 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.

Refining of an attribute belonging to the Link-kind metatype requires refining of both – the attribute type and the Link type. Different kinds of link types can be related by refining relationships.

5.3 Union type

A union type provides for representation of set of values that is a labelled union of a given collection of types. Each value of a union type can be represented by a pair $<$ type label, value $>$.

Type specification

$<$ union type $>$::= {union; type_of_label :$<$ label type $>$;
$<$ union element list $>$}
$<$ label type $>$::=$<$ type of integer $>$|$<$ character type $>$|
$<$ Boolean type $>$|$<$ enumeration type $>$| $<$ type identifier $>$
$<$ type of integer $>$::=$<$ type short $>$|$<$ type long $>$| $<$ type unsigned integer $>$
$<$ union element list $>$::=$<$ union element $>$|
$<$ union element $>$; $<$ union element list $>$
$<$ union element $>$::=$<$ label list $>$:$<$ attribute specification $>$|
default :$<$ attribute specification $>$
$<$ label list $>$::=$<$ label $>$|$<$ label $>$:$<$ label list $>$
$<$ label $>$::=$<$ occurence_of_label_type $>$
$<$ attribute specification $>$::=$<$ type identifier $>$| $<$ type specifier $>$

A label is an occurrence of a type_of_label type. 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

Definition 5.3 Union *type U' refines* union *type U if for each value type V_t in U there exists refining value type $V_{t'}$ in U' and a label type L_t in U is refined by the respective label type $L_{t'}$ in U'.*

Based on this definition, taking a common reduct of union type can be easily defined (in accordance with the definitions of common reduct (2.3) and most common reduct (2.4)).

A procedure of identification of most common reducts for application type Ts with attribute a_s and component type Tr with attribute a_r. Two cases are considered: when both types of attributes a_s and a_r are union types and when type of attribute a_r is a union, but type of attribute a_s is a simple type.

Both types of attributes a_s and a_r are union types To identify common reduct a maximal collection A of pairs of ontologically relevant labels $(l^i_{a_s}, l^j_{a_r})$ should be found such that label type $l^j_{a_r}$ refines label type $l^i_{a_s}$ and type of $l^j_{a_r}$ refines type of $l^i_{a_s}$.

Type of attribute a_r is a union, but type of attribute a_s is a simple type If for attribute a_s there is a relevant label $l^j_{a_r}$ of union type of attibute a_r, and type of the attribute a_s refines a type of the label $l^j_{a_r}$, then attributes a_s and a_r are included to common reduct of types Ts and Tr.

Example

```
{AppType;
    in: type;
    app_attr: {union;
        type_of_label: boolean;
        true: integer;
        false: {sequence; type_of_element: string}
        }
}
{ResType;
    in: type;
    res_attr: {union;
        type_of_label: integer;
        1: integer;
        2: string;
        3: {sequence; type_of_element: string}
        }
}

{R_AppType_ResType;
    in: c_reduct;
        metaslot
        of: AppType;
        taking: {app_attr};
        c_reduct: CR_AppType_ResType
        end;
}
{CR_AppType_ResType;
    in: c_reduct;
        metaslot
        of: ResType;
        reduct: R_AppType_ResType
        end;
```

```
    res_attr: {union;
        type_of_label: integer;
        1: integer;
        3: {sequence; type_of_element: string}
        }

    simulating: {
        R_AppType_ResType.app_attr == CR_AppType_ResType.res_atr
        }
}
```

6 Implementation phase

Support for implementation phase in the tool addresses mainly code generation issues. Code generation involves the transformation of metainformation into a target programming language. During implementation the results obtained at the design phase are transformed into the form of source files in the programming language.

Our implementation model introduces three kinds of entities: implementation type, adapter (wrapper) and "well-known" object. Every type constructed on the design phase (concretizing reducts and types) have a separate implementation. Adapter denotes a group of type implementations and serves as a distribution unit (adapters may reside on different machines). "Well-known" object is an object we know about at implementation time. These objects serve as entries at server sides of a system and usually are registered through Naming Service (if target architecture is a CORBA).

The tool allows a user to view information about these entities through graphical diagrams and allows the user to make decisions regarding some features of a system being developed. Also user may alter information that affects code generation. For example, a user is provided with an opportunity to make decisions about distribution of type implementations among adapters.

Before the code generation a user usually have to choose target programming language and technology that address specific aspects of a system implementation. Current version of the tool is quite limited: it is determined that Java is used as implementation language and distribution is addressed by CORBA technology.

The following language constructs are generated: IDL interface definitions, Java class definitions, method signatures, make files, and registration of "well-known" objects in Naming Service and binding to them.

7 Related work

Research on type lattices and respective algebras has already quite long history. Having different motivations, the respective developments may be characterized as a compromise between reasonable expressiveness and tractability of the modeling facilities. In the compositional method tractability is sacrificed in favour of specifications completeness. Due to the latter we get rigorous way of identification of common fragments of type specifications leading to their justifiable compositions for reuse.

The paper [3] was one of the first that introduced a calculus of record-like structures based on a type subsumption ordering forming a lattice structure.

In [22] a structure of component repository as an information retrieval system is considered. Component retrieval problem is considered for functions as components modeled by relations containing all admissible pairs for the function input/output. Refinement ordering of functions has lattice properties. The lattice is formed by join and meet operations on relations representing functions. Join (meet) represents the sum of (common) specification information that is contained in both relations.

In [27] a signature matching as a mechanism for retrieving software components from a component repository is proposed. Further they considered specification matching for functions represented by their pre- and post- conditions.

Subtyping relation is considered here analogously to [20].

A problem of resolving structural conflicts for component-based design is similar to a problem of conflict resolving for database schema integration. There are two basic approaches for conflict resolving: using of predefined transformation rules or using of a high level language for transformation definition.

Predefined transformation rules [18] are intended for automatic structural conflict resolving and integration schema generation. The developer gives a correspondence between paths in the schemas. On the basis of the predefined rules the integration schema is generated (classes and functions of structural transformation between local and integration schema classes). Main advantages of this approach consist in its simplicity and possibility to prove the correctness of the rule application. The main deficiency of the approach consists in using of fixed rules that restricts capabilities of the method.

Using of high level language [19] provides for a developer rich facilities for transformation rules definition. However, every conflict resolving function should be programmed and its correctness should be proved separately. Similarly to the previous approach, the developer should define correspondence between schema elements. This approach provides more flexible facilities of forming the integrated schema making possible to resolve any kinds of conflicts.

8 Conclusion

In the paper an approach for compositional information systems development is considered. This is a new approach intended for reuse of pre-existing components. The approach consists in identification of relevant fragments of such components and constructing their compositions satisfying specifications of requirements. Fundamentals of the method include the canonical model having formal interpretation and applied for specification of requirements and components definition; the theory of refinement, making possible to justify component reuse; specification calculus forming a basis for compositional constructing of the systems from components; ontological approach used for semantic contexts integration. Due to the applied methods of component specification and retrieval, the compositional method is scalable with respect to the number of existing components, provides for IS construction using various kinds of components (software, information, process ones), provides for provable justification of the design.

The architecture of a prototype is presented. The prototype combines conventional OAD methods support facilities with the facilities of the new compositional development method. The prototype is intended for developers assistance in course of the information systems compositional development.

References

[1] Abrial J.-R. *The B Book: assigning programs to meaning,* Cambridge University Press, 1996

[2] Abrial J.-R. B-Technology. Technical overview. BP International Ltd., 1992

[3] Ait-Kaci H. An algebraic semantic approach to the effective resolution of types equations. Theoretical computer science, 45, 1986, 293 - 351

[4] R.-J.Back, J. von Wright. Refinement Calculus: A systematic Introduction. Springer Verlag, 1998

[5] Berg K. and Kalinichenko L.A. Modeling facilities for the component-based software development method. In *Proceedings of the Third International Workshop ADBIS'96*, Moscow, September 1996

[6] F. Boumphrey, O. Direnzo, J. Duckett et al. XML Applications, Wrox Press, 1998

[7] D. O. Briukhov, L. A. Kalinichenko. Component-Based Information Systems Development Tool Supporting the SYNTHESIS Design Method. Proc. of the East European Symposium on "Advances in Databases and Information Systems", Poland, Springer, LNCS No.1475, 1998

[8] Briukhov D.O., Shumilov S.S. Ontology Specification and Integration Facilities in a Semantic Interoperation Framework, In *Proc. of the International Workshop ADBIS'95,* Springer, 1995

[9] EIA Interim Standard: CDIF-Framework for Modeling and Extensibility. EIA, 1991

[10] P. Fankhauser, E. J. Neuhold. *Knowledge Based Integration of Heterogeneous Databases.* Integrated Publication and Information Systems Institute (GMD-IPSI), Darmstadt, 1993

[11] Florescu D., Levy A., Mendelzon A. Database techniques for the World-Wide Web: A survey ACM Sigmod Record, N 3, 1998

[12] M.Fowler *UML Distilled*, Addison-Wesley, 1997

[13] L. A. Kalinichenko. *Compositional Specification Calculus for Information Systems Development.* In Proc. of the East-West Symposium on Advances in Databases and Information Systems (ADBIS'99), Maribor, Slovenia, Lect. Notes Comput. Sci., 1999

[14] Kalinichenko L.A. Composition of type specifications exhibiting the interactive behaviour. Proc. of EDBT'98 Workshop on Workflow Management Systems, March 1998, Valencia

[15] Kalinichenko L.A. Method for data models integration in the common paradigm. In *Proceedings of the First East European Workshop 'Advances in Databases and Information Systems'*, St. Petersburg, September 1997

[16] Kalinichenko L.A. SYNTHESIS: the language for desription, design and programming of the heterogeneous interoperable information resource environment. Institute for Problems of Informatics, Russian Academy of Sciences, Moscow, 1995

[17] Kalinichenko L.A.,Skvortsov N.A.,Briukhov D.O., Kravchenko D.V., Chaban I.A. Designing Personalized Digital Libraries "Programming and Computer Software" Vol. 26, No. 3, 2000, pp. 123-133

[18] Wolfgang Klas, Peter Fankhauser, Peter Muth, Thomas Rakow, Erich J. Neuhold. Database Integration using the Open Object-Oriented Database System VODAK Omran Bukhres, Ahmed K. Elmagarmid (Eds.): Object Oriented Multidatabase Systems: A Solution for Advanced Applications. Chapter 14. Prentice Hall, Englewood Cliffs, N.J., 1996

[19] Kosky A. Transforming Databases with Recursive Data Structures. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, November 1995.

[20] Liskov B., Wing J.M. Specifications and their use in defining subtypes. Proc. of OOPSLA 1993, ACM SIGPLAN Notices, vol. 28, N 10, October 1993

[21] Lumpe M. A Pi-Calculus Based Approach to Software Composition, Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999

[22] Mili R., Mili A., Mittermeir R. Storing and retrieving software components: a refinement based systems. IEEE Transactions on Software Engineering, v. 23, N 7, July 1997

[23] Object Management Group. CORBA 2.3.1 /IIOP Specification. OMG document 99-10-07, 1999.

[24] PLATINUM Paradigm Plus Reference Manual. PLATINUM technology, 1996

[25] G. Salton, C. Buckley. *Term-Weighting Approaches in Automatic Text Retrieval.* Readings in Information Retrieval, K. S. Jones and P. Willett, Kaufmann, 1997

[26] Workflow Management Coalition. Workflow reference model. Workflow Management Coalition Standard, WfMC-TC-1003, 1994.

[27] Zaremski A.M., Wing J.M. Specification matching of software components. ACM Transactions on Software Engineering and Methodology, v. 6, N 4, October 1997