Acta Cybernetica **15** (2002) 567–581.

A distributed program synthesizer*

Vahur Kotkas†

Abstract

This paper describes an architecture of a distributed synthesizer for automated program construction. The objective of the synthesizer is to realize the ideas of Structural Synthesis of Programs in a computer network. The synthesizer handles structural specifications stored into Java classes as meta-interfaces and works on a network using CORBA technology.

Keywords: SSP, Java, Automated Synthesis of Programs, Meta-Interfaces.

1 Introduction

In the last decade Object Oriented programming (OOP) languages (like C++, Java, C#) became dominant providers of software reusability. However, the reuse efficiency greatly depends on developers' experience in programming and their knowledge on existing libraries.

There are several development environments available that assist in the selection of a proper libraries, but none of them generate fully operational code - this remains the developers' task. Searching for the proper library usually means reading the descriptions of the many libraries available. This is time consuming activity and hence software developers still create new libraries without knowing that some other software, having the same functionality, already exists. This indicates that there is a need for automated handling of software libraries.

One way to automate the software design process is to use Structural Synthesis of Programs (SSP) [1]. SSP is a technique of deductive synthesis of programs based on automatic proof search in intuitionistic propositional calculus. This technique uses classes of our problem domain extended with declarative specifications to generate new software automatically. The resulting software is correct (does not need any further checking) with respect to the correctness of the classes it is based on. The solving complexity is hidden from the end-user into the system.

The idea of using SSP for automated program generation is not new. Already in seventies a Priz family of programming languages was developed in the Institute of Cybernetics, that allow engineers to solve their tasks using a very high-level

*This work was partially funded by Estonian Innovation Foundation under the contract No. 6kl/00.

†Institute of Cybernetics at Tallinn Technical University, Estonia, email: vahur@cs.ioc.ee

programming language. A similar approach has justified itself quite well in the Amphion system [2].

Because of its relative robustness and flexibility the Java programming language was chosen for the SSP addition in the current study, but in principle this kind of additions can be provided to other OOP languages as well.

This paper introduces architecture of a software synthesizer that performs automated program construction and describes briefly the declarative specification language of the meta-interfaces added to Java classes that enable SSP.

This work is inspired from the work done in Institute of Cybernetics, Estonia during several decades and related to the work of Sven Lämmerman [3] from Royal Institute of Technology, Sweden.

2 Method

Java classes and objects do not contain sufficient information on their components relations and internal functionality to perform automated program synthesis. Automated program construction is not possible without that information. To overcome the problem, we introduce a meta-interface as an extension to a Java class, where the needed information is provided.

A meta-interface is a declarative specification that:

1. introduces a collection of interface variables of a class
2. defines, which interface variables are computable from others under which conditions.

For instance, having a class $Triangle$ and a method $findSideSine$ for computing size of a side according to the theorem of sine: $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)}$, we can introduce interface variables for all angles (A,B,C) and sides (a,b,c) of the triangle and declare a meta-interface that will specify how one can use the method. This meta-interface looks as follows:

```
var a,b,c,A,B,C : any;
rel a,A,B -> b {findSideSine}
rel b,B,C -> c {findSideSine}
 . . .
```

Here the $findSideSine$ is an implementation of the theorem of sine in Java in the form of a method. The meta-interface just declares how the method can be used.

The usage of a meta-interface is as follows: one writes a request for synthesis of a method with input $x_1, \ldots, x_m$, where $m > 0$, and output y, whereas $x_1, \ldots, x_m, y$ are variables specified in the meta-interface, for instance in the form of the formula $x_1 \& \ldots \& x_m \rightarrow y$ and lets a prover to prove that this formula is derivable from the specification of the meta-interface, i.e. that the goal of synthesis of the requested method is achievable.

The prover returns a sequence of rules applied in the proof, which from the synthesis point of view represents an algorithm. This algorithm is used to generate the program code that solve the initial problem. Thus the algorithm is a co-result (or side-effect) of the proving process [4].

The aim of introducing meta-interfaces is obvious: to make classes as components more flexible. Indeed, if a meta-interface specifies n variables, then one can write 2^{2^n} requests for synthesis of which only a few may appear provable. However, we still get considerable flexibility compared to conventional interface of Java, without the need to implement all the possible programs that one may need during the runtime.

A meta-interface can be written for two different purposes. First, it may specify possible usage of the class, i.e. its derivable methods, like in our example. Second, it can be used as a specification showing how some application software should be composed from components that are supplied with meta-interfaces. In the latter case, a new class can be built completely from the specification of its meta-interface. For this purpose, specifying equality rules between some components of the new class may be needed.

3 Declarative specification language

The meta-interface, that contains declarative specifications, may be introduced to the classes

1. as an addition to the programming language
2. in the form of comments
3. as an array of strings.

The first solution, for adding the specifications, changes the Java programming language. The declarative specifications become native components of classes and are compiled with the rest of program. This avoids recompilations during the runtime. On the other hand, as we changed the language we need also a new Java compiler and virtual machine and we can not use our techniques with other versions of Java.

Introducing the structural specifications into a Java class as specifically formatted comments allows to use existing Java compilers and interpreters, as we do not need to modify existing programs — we just add some more comments to them. However, to find the specifications from the source files during the runtime is time consuming. Even more, this approach is not usable when source files may change or are not accessible during the runtime.

We have chosen the third possibility — to introduce the structural specifications into the class as an array of strings. In this case we can always access the specifications during the runtime as they are present in the component of the object in use. The Java programming language remains unchanged, hence, the solution

works with all versions of Java. However, this approach needs additional resources for compiling these specifications on the request of program synthesis.

The specification language of meta-interfaces consists of two sections — **var** and **rel** section. The **var** section specifies the components used in the **rel** section. Multiple instances of these sections can be used in a specification in random sequence.

The **var** section is an obligatory section in the specification of every class, without it the specification is incomplete. The **var** section is formally specified as follows

$$\texttt{var}\ \ a_1, a_2, \ldots, a_n\ \ \texttt{:}\quad \mathit{type},$$

where $a_i (i = 1..n)$ are declared variables. If *type* is represented with the keyword **any**, the declared component already exists in the Java class and the exact type of that component is applied during the compilation of the specification. Otherwise, if the names of Java primitive types or classes are used, new components are added to the synthesized program.

Some of the components in the **var** section may be defined as virtuals. In this case keyword **vir** is used instead of **var**. Virtual components are not taken into account when the state of the object they belong to is evaluated. This issue is further explained in the chapter 6.3.

The **rel** section defines relations of the declared components in the form of computability statements or computational constraints. These relations define usage of Java methods, equivalences, equations and inequalities. The statements are written as follows:

$$\texttt{rel}\ \mathit{Label}\texttt{:}\quad \mathit{RelStatement}.$$

The *RelStatement* specifies an equivalence, equation, inequality or method declarations. The *Label* is a name given to the current specification statement that can be referred for debugging or is used for modifying inherited specifications.

As Java classes inherit properties from their superclasses, also the specifications of meta-interfaces are inherited. Meta-interface in a subclass overrides the specification statements of the superclass, if it defines a new relation with the same label.

Method declaration presents either a class method, an instance method or a special narrowing method denoted with the keyword **narrow**. It describes the input and output parameters required for the method invocation and exceptions if needed. A general method declaration construction is the following:

$$[\mathit{InputSpec}]\ \texttt{->}\ \mathit{OutputSpec}\ \{\ \mathit{MethodName}\ \}$$

Here the square brackets denote that the *InputSpec* is optional.

In case the $MethodName$ is equal to keyword **narrow**, there should be exactly one component defined as method input and one for output. Keyword **narrow** represents a conversion of one type to another, if possible.

The *InputSpec* consists of two lists of components separated by `&`-symbol. The components described on both lists are separated by commas. The components of

the list before `&` are handled as method's formal parameters and the components after `&` respectively define instance variables that the method uses. If the list after the symbol `&` is empty, `&`-symbol must be discarded.

The *OutputSpec* has a similar structure to the *InputSpec*. The only difference is that before the `&`-symbol only one component is allowed, because an arbitrary method in Java may return only one value in time. In case there is no element specified before `&`, the method is of type void.

Additionally, the output parameter list may end with a `|` separated list of components that defines a set of exceptions, which could be thrown by the method.

For example `rel a, d.y & c.x, d.x -> c.y & d.y | e { doIt }` illustrates the usage of constructions, where *InputSpec* = `a, d.y & c.x, d.x` and *OutputSpec* = `c.y & d.y | e`. Component a and d.y are formal parameters for local method *doIt* with signature $type(c.y)$ $doIt(type(a), type(d.y))$, and $c.y$ is the output of that particular method. Global components $c.x$ and $d.x$ are used in the computations and $d.y$ is modified as a side-effect of this method. The component e represents an exception that may be thrown by the method.

Equivalence defines a pair of components that should stay equal at any stage of an executed synthesized program. One can think of equvalent components as of objects that are stored into the same memory location. Equivalences are used as connectors between components enabling to build larger systems from smalles components. An example of an equivalence definition may be the following: `rel a.x == b.y`. One component may be present in many equivalence definitions. This forms a group of components that should stay equal during the execution.

Equation or inequation in the *RelStatement* is useful when one solves an engineering task. Java programming language does not include any solver that handle equations automatically and the solver for the equations have to be coded imperatively by the software developer. By allowing these kind of definitions, we can support constraint enriched Java classes and significantly reduce the programming time. However, we need a general purpose solver that handles these kind on specifications. Recent results in genetic algorithms show already today acceptable performance in optimization tasks, hence we are looking optimistically toward them.

There is a special use of having only one **var** component defined in the meta-interface — one can avoid from writing the full path to the subcomponent used in relations. For example instead of writing (`rel x = length.m + 7`) we can just write (`rel x = length + 7`), if the component *length* contains only one **var** component. In the synthesized program the component *length* would be automatically substituted to *length.m*, hence the computations go correctly. This allows to write the specifications in more convenient and more meaningful way.

Substitution is also useful when one does not know the name of the **var** component in the specific class, but is sure that there exists only one such component. This may happen when we prepare a superclass and the real types and components are defined in the subclasses.

4 Example of the meta-interfaces

Let us have triangulation as a sample problem. We have two triangles that have one side in common (see Fig. 1) and we have measured values for one side (a) and two angles (C and A) of the first triangle (t1) and two angles (C and A) from the second triangle (t2). Our task is to compute the total area of the two triangles.

Please note that the variable names on the figure do not denote the equivalence between different angles and sides — only the mapping between an object on the figure and its class is presented there.

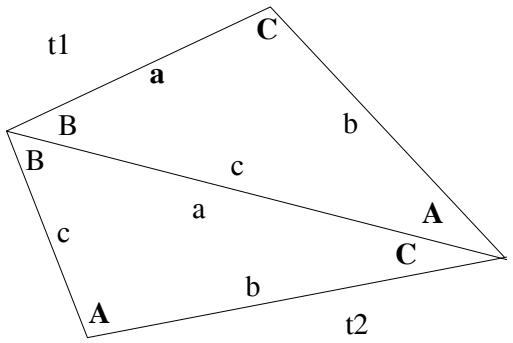

Figure 1: Graphical representation of the triangulation example.

While using an OOP language (like Java) we need to create two classes for the problem description. The first class describes a triangle and another composes the problem of triangulation from two triangles. For both classes we need to add a meta-interface (see Fig. 2 and the array of String SSPspec on Fig. 3) in order to specify the relations among the components.

```
var a,b,c,S : any
vir A,B,C : any
rel X: a,B,C -> S {calcArea}
rel Y: a,A,C -> c {findSideSine}
rel Z: A+B+C=Math.PI
```

Figure 2: Declarative specification of class Triangle.

All components (variables and objects) of the class (statements starting with var and vir), that are used in the relations (statements starting with rel), must be redefined in the specifications. Let us note that the actual type of components, specified here as of type any, is determined from the object during the synthesis.

Because of the encapsulation reasons private components can not be used in the declarative specification.

We can define also new components in the specification by giving their type instead of keyword any, but they do not become real components of the class. We may consider them as temporary components that are available only during the execution of the synthesized program.

The *calcArea* and $findSideSine$(see Fig. 2) are methods implemented in the class Triangle. The method *calcArea* realizes computation of the area, knowing one side and its two nearby angles. The method $findSideSine$ computes a side, knowing another side and their opposite angles. The third *rel* statement describes relation among the angles of the class in the form of equation — the fact that the sum on inner angles of a triangle is equal to 180 degrees.

The first *rel* statement of class Problem (see Fig. 3) defines equivalence between the components. Equivalence means that the values of components stay always equal during the execution of the synthesized program.

```
import ee.ioc.cs.synthesizer.*;

public class Problem implements SSPInterface {
    public static String[] SSPspec = {
        "var t1, t2 : any",
        "var S : any",
        "rel U: t1.c == t2.a",
        "rel V: S = t1.S + t2.S"};
    public Triangle t1, t2;
    public double S;

    public void run() {
        t1.a = new Length('km',2);
        t1.C = new Angle('deg',90);
        t1.A = new Angle('deg',45);
        t2.C = new Angle('deg',45);
        t2.A = new Angle('deg',90);
        String progID = SSP.synthesize(this, "->S");
        SSP.execute(progID, this);
    }
}
```

Figure 3: The class Problem.

The labels X, Y, Z, U, V may be omitted. They may be used for debugging purposes, as they are added to the algorithm provided by the Planner. These labels are used here to make later in this paper references to these relations. The Planner and the algorithm are described in more detail in chapter 6.4.

As a result of the call *synthesize* of the class SSP (see the method *run*() on the Fig. 3) a new method will be synthesized (e.g uniquely identified as $xf17634$,

see Fig. 4) that realizes the requested computational problem. The name of the synthesized method is returned as its ID and can be used later for the method invocation.

The method *exec* executes the synthesized method and as a result modifies the object p by assigning proper value to the component S.

```
public void xf17634(Problem p) {
    p.t1.c = p.t1.findSideSine(p.t1.a,p.t1.A,p.t1.C);
    p.t2.a = p.t1.c;
    p.t1.B = Math.PI-p.t1.A-p.t1.C;
    p.t1.b = p.t1.findSideSine(p.t1.a,p.t1.A,p.t1.B);
    p.t1.S = p.t1.calcArea(p.t1.a,p.t1.B,p.t1.B);
    p.t2.c = p.t2.findSideSine(p.t2.a,p.t2.A,p.t2.C);
    p.t2.B = Math.PI-p.t2.A-p.t2.C;
    p.t2.b = p.t2.findSideSine(p.t2.a,p.t2.A,p.t2.B);
    p.t2.S = p.t2.calcArea(p.t2.a,p.t2.B,p.t2.C);
    p.S = p.t1.S + p.t2.S;
    return;
}
```

Figure 4: The synthesized method.

5 Distributed components

Software developers have long held the belief that complex systems are easier to be built from smaller components. There are two engineering drives in the development of a component-based system [5]:

1. Reuse - the ability to use existing components repeatedly

2. Evolution - development and maintenance of a highly componentisized system is easier and cheaper.

Composing a distributed application - an application composed of distributed components - adds the following features

1. Higher flexibility - as the components are not compiled into the application, the changes in the components do not affect the consistency of the distributed application if the interfaces of these components are fixed and semantics of their inputs and outputs remains unchanged.

2. Higher fault-tolerance - if a distributed component is developed for a certain environment and is well tested to work properly in it, then composing an application using these distributed components, that reside always in their native environment, cuts down in programming and testing time.

In a distributed application the intercomponent communication is fairly expensive in terms of time and other resources [6]. Thus components are encouraged to be larger than smaller. However, larger components have more complex interfaces and are changed more probably during the system development phase. The larger components are, the less flexible is the whole distributed application. To achieve optimum we should consider the level of abstraction of components, their likelihood to change, complexity etc. These ideas are followed while composing the architecture of the synthesizer, which is described in the next chapter.

6 Architecture of distributed synthesizer

By using the CORBA technology [7] in the synthesizer we get more flexibility for computing in a network and possibilities for concurrent computing, hence we call it a distributed synthesizer. Using CORBA also forces us to follow the ideas of modular programming and supports the program componentization.

While using CORBA for component interconnection we have to acknowledge that objects as entities can not be sent over a network, as CORBA is inter platform and inter programming language communication architecture. It means that every object should be serialized before delivery or in other words turned into a byte stream.

Serialization has two drawbacks on performance: firstly we need additional computing power for the serialization and secondly the serialized data takes always more memory than the original object, thus we may need wider bandwidth network for communication. This draws a more concrete granulation criteria to the decomposition process i.e. one should avoid loops in which there is a lot of data exchange over the network.

Considering the ideas described above we propose the following architecture for a distributed synthesizer, decomposed into 7 logically separated components (see Fig. 5): Object Factory, Compiler, Decorator, Knowledge Base (KB), Planner, Code Generator and Component Repository (CR)

6.1 Object Factory

The Object Factory (OF) is the central process that maintains the work of the other components. It does the performance evaluations on the network and sends a new task to the host that has least load at the moment. The Program uses a predefined local class SSP that finds the location of the OF on the current network and forwards the request to it. This allows to hide the whole machinery of the synthesizer from the end-user.

6.2 Compilation

Knowledge Base (KB) is a database-like structure that allows storing compiled declarative specifications for each class used in the program. When Compiler re-

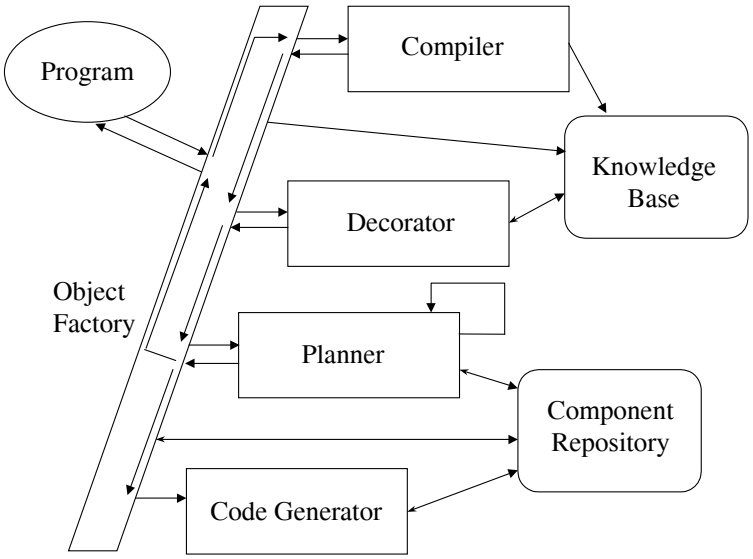

Figure 5: Architecture of the distributed synthesizer.

ceives a request for program synthesis from the OF, it first checks whether all the descriptions of classes that are used in the declarative specification are represented in the KB. The data received by the Compiler is consisting of serialized object, its serialized class and the computational problem specification i.e. the goal.

As we can never be sure that the classes stay unchanged the Compiler calculates a hash number based on the declarative specification for each class and compares it to the hash number stored in the KB. If the hash numbers are matching the compilation of that class can be skipped, otherwise if they do not match or the class description does not exist in the KB, the Compiler parses the declarative specifications of the class and stores the resulting description into KB.

Computing and checking the hash number assigned to class descriptions allows us to avoid compilation of the declarative specification every time a program synthesis request is made. Thus it speeds-up the program execution when similar problems have to be solved often.

6.3 Decorator

The Decorator creates a bipartite graph like structure out of the class descriptions stored in the KB. The bipartite graph has two types of nodes - components and relations [8]. The reason of building such a structure is to get rid of the object-oriented hierarchy, thus making it more suitable for the search. Fig. 6 presents

an example of a bipartite graph that corresponds to the sample specifications described above (see Fig. 2 and Fig. 3), where the smaller circles denote objects and components, and bigger circles represent to relations.

To still remember the hierarchy of the initial object, new relations are added to the graph (see unlabeled relation nodes on Fig. 5) that tie objects to their components.

From these additional relation is also clearly visible the distinction of the var and vir components of the declarative specification. Only the var components are included into the added relations and the vir components are left out. That means if there is a computational problem where the goal is an object having many components, then when all its var components are "known", meaning that the value of the component is known, the object gets the status "known" using that relation.

And vice versa - if an object is "known" the status transforms only to its var components.

The next task of the Decorator is to paint the object and variable nodes in the graph. A painted node in the graph represents to a component with the state "known". Considering our triangulation example the components a, C and A of Triangle t1 and components C and A of Triangle t2 are known, thus their nodes on the graph should be painted (dashed vertically on Fig. 6).

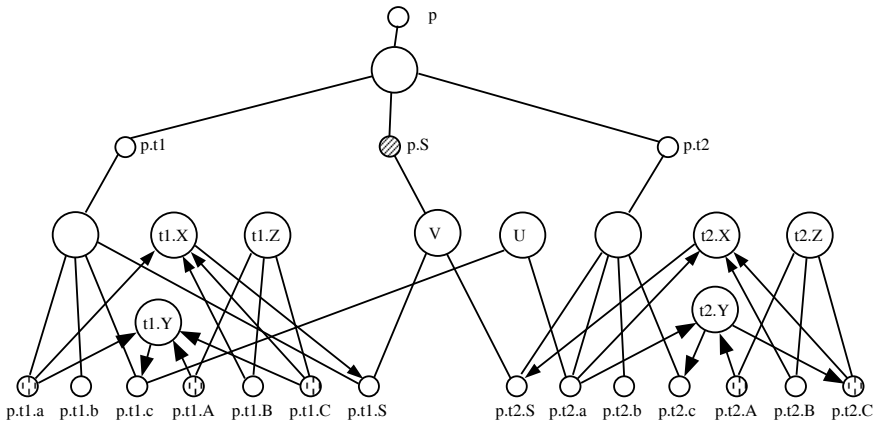

Figure 6: The search-space graph of the Triangulation example.

A different "color" is used to mark the goal. As the problem was to find the total area of the two triangles the node designating component p.S should be painted as goal (filled with upward diagonal lines on Fig. 6).

This structure is the search space delivered to the Planner.

6.4 Planner

The main function of the Planner is to find a solution for the problem specified by a user. Before starting with problem solving the Planner checks from the Component Repository (CR) whether a solution already exists for it. In the case the solution does not exist, the Planner starts solving it.

In principle there are two kind of relations that may occur in the declarative specification:

1. Unconditional relations implementing unconditional computability statements of SSP. In such relations computability of some (output) object depends only on some other (input) object(s). Unconditional relations of several types as equations, equivalencies, Java methods etc. are available in the specification language.

2. Conditional relations or relations with subtasks, implementing conditional computability statements of SSP, describe more sophisticated dependencies where output objects depend not only on input objects but also on solvability of some other computing problems. This kind of relations is unfortunately not present in our toy-example.

The problem specification for Planner is of form `x->y`, where `x` denotes the set of "known" (nodes painted with vertical dashes on the Fig. 6) objects and `y` denotes the set of objects to be computed (the goal). The Planner has to construct an algorithm (a sequence of relations) that describes how to compute y from x [4].

The proof search strategy of SSP applied in the Planner is

1. an assumption-driven forward search to select unconditional relations (linear planning). The algorithm works in the forward direction with unconditional relations only. At each step a relation, which input objects are "known" and at least one output object in "not known", is located and added to the algorithm. When adding a relation into the algorithm all its output objects are set as "known". The search is completed when all the nodes marked as "goal" are also "known" or there is no relations left with all inputs "known".

2. a goal-driven backward search to select and solve subtasks. The search is applied if the linear planning cannot be continued. Only such relations with subtasks are considered which input objects are "known". First the CR is checked for the existence of the solution of every subtask the relation have. If existing solutions are not found, the Planner is recursively used for solving every subtask of the relation considered. If all the subtasks of the relation are solved, the relation is added to the algorithm. Linear planning is used after every invocation of a relation with subtasks in the algorithm.

3. a minimization is applied to the resulting algorithm of the two previous search strategies. The search strategies above do not guarantee that we have built the shortest possible algorithm for computing the desired goal. Even more, the synthesized algorithm may contain relations that are not necessary for

computing the goal. Minimization is used to exclude such relations from synthesized algorithm. As a result of planning we get an algorithm that is not necessarily the shortest, but it does not contain unnecessary relations.

The algorithm is passed to the Code Generator and the problem specification in the CR is marked as solvable.

6.5 Code Generator

The code generation is a straightforward process, where the algorithm is translated into the class of the appropriate programming language i.e. to a Java class when the source language was Java. The class is compiled and a component including a newly created instance of this class is summoned and added to the CR. The component is stored into the CR with its problem specification that makes it possible to use that component repeatedly for solving many similar tasks.

If the computational problem were solvable and the algorithm is delivered to the Code Generator the Planner informs the OF about the solvability of the assigned problem and the OF delivers also the identification of the component stored in the CR to the Program.

Such an approach supports so-called Case Dependent Software Reuse (CDSR). It is called case dependent, because the planning process does not depend only on the declarative specifications given with particular object, but also on the current state of an object. The state of an object is determined by the states of its components (variables and objects), that can be marked as "known" or "not known".

6.6 Using synthesized components

When the distributed component is added to the CR, the user program may use it in two ways - by accessing the distributed component from the CR sending data to it and receiving output, or by retrieving the component's class and using its instance locally.

The aim of CR is to maintain a set of components implementing a certain interface. Thanks to the usage of this fixed interface we can use later all these components in program's initial context even when a certain component is not yet available at the moment of program creation.

An Object Factory that on a request creates the appropriate component and forwards the input parameters to it maintains the processing components of the distributed synthesizer. To take full advantages of the network, we may create multiple instances of the processing components on different servers and execute them in parallel.

Furthermore, the fault-tolerance of application is increased through redundancy provided by multiple instances of same component on different servers. Hence, if something happens with one server providing particular component, others backup it and we can distribute the system's load between different hosts in the network while solving different problems simultaneously.

We would not achieve any speed-up to the work of synthesizer when running just one user program, but the benefit appears when several user programs access the synthesizer simultaneously.

All components of the distributed synthesizer are implemented using the Object Management Group's Common Object Request Broker Architecture (OMG CORBA) [7], which provides a flexible communication and activation substrate for distributed heterogeneous object-oriented computing environments. CORBA enhances application flexibility and portability by automating many common development tasks such as object registration, location and activation; demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching.

Several aspects related to CORBA, Quality of Service, reliability of applications, and performance are studied in [9, 10, 11].

7 Conclusions

In the current paper meta-interfaces for Java classes are introduced and an architecture of a distributed program synthesizer is presented. The synthesizer is the main part of SSP realization that allows automatically construct programs out of declarative specifications provided by the meta-interfaces.

Our main objective is to extend a distributed programming language (like Java) with SSP capabilities that automates software reuse and helps users to design programs. The distributed synthesizer is a supporting tool that handles declarative specifications provided in the meta-interfaces of classes and does the actual program construction hidden from the end-user.

Here, an extension to the Java programming language is presented, but in principle such architecture of the synthesizer would suit also to other OOP languages, which classes are extended with structural specifications.

The efficiency of the synthesizer is low when considering only single user program execution, but the reuse of already synthesized programs gives a significant effect on the network where multiple agents are solving similar tasks.

For handling equation systems and additional constraints like inequations that enrich the specification language, a general solver has to be developed. We are looking optimistically towards genetic algorithms that have shown very promising results in solving different optimization tasks.

References

[1] E. Tyugu. The structural synthesis of programs, Lecture Notes in Computer Sciences, Vol. 122, 1981, pp. 290–303.

[2] M. Stickel, R.Waldinger, M.Lowry, T.Pressburger, I.Underwood. Deductive Composition of Astronomical Software from Suroutine Libraries. In 12th Con-

ference on Automated Deduction. A.Bundy, ed., Springer-Verlag Lecture Notes in Computer Science, Vol.814.

[3] S. Lämmermann. Automated Composition of Java Software, Lic. thesis, Department of Teleinformatics, Royal Institute of Technology, Sweden, Technical Report TRITA-IT AVH 00:03, ISSN 1403-5286, ISRN KTH/IT/AVH-00/03-SE, May 2000.

[4] M.Harf, E.Tyugu. Algorithms of structured synthesis of programs. Programming and Computer Software, 6, 1980, pp 165-175.

[5] J. Hopkins. Component Primer. Communications of the ACM, October 2000, vol. 43, no. 10, pp. 27-30.

[6] D. Budgen, P. Brereton. Component-Based Systems: A Classification of Issues. Computer (IEEE CS), November 2000, vol. 33, no. 11, pp. 54-62.

[7] Vinoski, CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communications Magazine, vol. 14, no. 2, February 1997.

[8] E.Tyugu, T.Uustalu. Higher-Order Functional Constraint Networks. Constraint Programming. NATO ASI Series F. Springer-Verlag 1994, pp 116-139.

[9] J. Zinky, D. Bakken, R. Schantz. Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, vol. 3, no. 1, April 1997, pp. 1-20.

[10] S. Maffeis, D. C. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communications Magazine, vol. 35, no. 2, February 1997, pp. 56-60.

[11] A. Gokhale, D. C. Schmidt. The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks. In Proceedings of GLOBECOM '96, London, England, November 1996, IEEE Press, pp. 50-56.