Synthesis of distributed programs

Vahur Kotkas

Institute of Cybernetics at TTU
vahur@cs.ioc.ee

Abstract. This paper presents a way for distributed program synthesis. We use Java programming language as a base language that is enchanted with declarative specifications. Program synthesizer that performs automated program construction uses these specifications. Several aspects are presented on how in this framework a new program can be synthesized and executed, taking advantage of the distributed computing.

Introduction

Distributed processing is used to speed up computational processes already for decades. Still, these capabilities are used only by a limited number of programmers. There are two main reasons why distributed processing is not widely used are:

1. Availability of the High-Performance Computing (HPC) facilities (super-computers) is limited because of monetary or political reasons.
2. Writing a program to be executed in a distributed manner is far more complex than doing it for sequential case.

The first reason is quickly fading, as it is possible today to build a computer cluster with quite affordable price and take advantage of HPC and GRID computing.

It is more crucial with software development – there are still no widely accepted solutions allowing one to develop software for distributed computing. Still there exists strong demand for an easy to use software development environment.

In this paper we concern mainly issues related to modeling and simulation environments. Scientists and engineers use modeling and simulation often in their research. Generally, these activities are computationally difficult. It cannot be expected that people who need simulations in their work are also computer specialists capable to develop low level complex programs that take advantage of HPC.

[Kotkas02] describes an architecture of a program synthesizer, where the main keywords to highlight the problem domain were

–high level specifications
–fully automatic program construction
–program reusability.

These features are addressed to researchers and engineers who are not computer scientists or skilled programmers and need an easy to use tool for modeling and simulation.

The program synthesizer allows one to specify their problems with high-level declarative specifications and let the synthesizer create automatically a program to solve their problem.

Basically, one has to describe the parts of modeled system (modules or objects), tie them together into a full model, give the necessary input parameters and ask the system to synthesize a program that calculates the values of resulting parameters. This approach gives a significant flexibility as the previously developed models can be very easily reused.

The solving task is commonly computationally difficult and need a lot of computing power; hence there is a need towards distribution of the synthesized programs in order to carry out the computations in a shortened time scale.

In this paper we describe the possibilities to take advantage of HPC in the given framework. The base language for the framework is Java, which is extended with declarative specifications and used as an input to the program synthesizer that is the main part of the framework.

In the next chapter we describe the high level specification language embedded into Java programming language, then describe how the program synthesis is carried out and finally look at the possibilities for the distributed program execution.

Declarative Specifications

Declarative Specifications are embedded into a Java class as an array of Strings. Each element of the array represents one clause of the specification. These specifications can be accesses easily with Java reflective tools whenever an object or class is accessible. Every class that should be incorporated in the program synthesis should have the declarative specification. We call such specification a meta-interface of the class as it describes the interface variables and their interrelations that can be used in the synthesized program.

The specification language consists of two sections -- **var** and **rel** section. The **var** section specifies the components of current class - Java primitives and objects - 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 meta-interface. It is formally specified as follows

```
var a_1, a_2, ..., a_n : 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.

The **rel** section defines relations of the declared components in the form of computability statements or computational constraints. In simpler words, **rel** statements show how some components can be calculated out of other components. The statements are written as follows:

```
rel Label: RelStatement
```

The *RelStatement* specifies an equivalence, equation, inequality or method declarations. The *Label* is a given name to the current specification statement that can be referred for debugging or is used to modify 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 or an instance method. It describes the input and output parameters required for the method invocation and exceptions if needed. In other words, the method declarations show how the methods and components of the current class are interrelated. A general Method declaration construction is the following:

```
[SubtaskSpec][InputSpec] -> OutputSpec { MethodName }
```

Here the square brackets denote that the *SubtaskSpec* and *InputSpec* are optional.

SubtaskSpec defines a comma-delimited list of *Subtask*s. Each *Subtask* is surrounded with square brackets and denote a computational problem of type x -> y, which solvability is treated as input for the method, where x and y are lists of components of class *ClassName*. The Subtask declaration construction is the following:

```
[ClassName |-] [InputSpec] -> OutputSpec
```

In case of Subtask is present in the Method declaration the synthesizer creates a new program that solves the computational problem on a separate object of the given class *ClassName* where x and y are lists of components of this class.

In case the *ClassName* is omitted the current context object (object in which the declaration is present) is used where x and y are components of the object.

The synthesized program can be called from within the method body in the following way:

```
SSP.subtask(1, input, output);
```

Here the *SSP.subtask* is a method from the *SSP* class that automatically finds the corresponding synthesized program matching to the first subtask (see parameter 1 in the method call) of the current Method. If there are more than one subtask specifications in the Method declaration they can be accessed using their number of order in the *SSP.subtask* call. The input and output are structures of input and output parameters for the synthesized program that were defined in the Method declaration *SubtaskSpec* section. Class SSP is a collection of methods that can be called from within the Java program. The name is given referring to Structural Synthesis of Programs paradigm on top of which the programs synthesizer operation lies.

The *InputSpec* consists of two lists of components separated by **&**-symbol. Commas separate the components described on both lists. 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 [compClass|-k,l->m] 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. This method is applicable only in the case a computational problem *k,l-> m* is solvable on class *compClass* or in the other words *m* is computable out of given *k, l* and the default initializations made when an object of class *compClass* is created.

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.

Equations and inequations in the *RelStatement* are useful when one solves an engineering or modeling task. Java programming language does not include any solver that handles equations automatically and the solver for the equations has to be coded imperatively by the software developer. By allowing these kinds of definitions, we can support constraint enriched Java classes and significantly reduce the programming time. However, we need an external solver that handles such kind of computations.

An example of specification language usage

Let us illustrate the usage of the specification language with an example of a simple modeling system *SystA*, which uses 2 subcomponents: *ModelA* and *ModelB* that implement 2 algorithms. Unfortunately these models are built to use different measurement systems – the *ModelA* uses feet's and miles while the *ModelB* uses metric measurement units. In practice algorithms that are developed based on a number of measurements are quite often given in measurement units commonly used. Problems appear when we need to combine several of such models and they may not been prepared to be used with the same measurement system.

In order to solve the modeling task we have to execute the computations in *ModelA* and *ModelB* sequentially so that some of the output parameters of *ModelA* would be input parameters for *ModelB*. As these models are using different measurement units, we have to translate the measurement units in between.

Typically the algorithms (created by engineers) given in the form of program code would look the following :

```
public class ModelA {
     // in1 and in2 given in NMI, out1 and out2 given in feet
     public double in1, in2, out1, out2;
     public double computeOut1(double input1, double input2) {
      /* implementation follows here */
     }
     public double computeOut2(double input1, double input2) {
      /* implementation follows here */
     }
}
public class ModelB {
     // in1 and in2 given in km, out given in cm
     public double in1, in2, out;
     public double computeOut1(double input1, double input2) {
      /* implementation follows here */
     }
}
```

Fig. 1. Initial source of the *ModelA* and *ModelB* implementing some algorithms

We have to add a meta-interface (*SSPspec*) to the class in order to take advantage of the automated synthesis of programs and highlight the measurement units in use

```
import ee.ioc.cs.synthesizer.*
public class ModelA implements SSPInterface {
     public static String[] SSPspec = {
      “var in1_nmi, in2_mni, out1_ft, out2_ft : double“
      “var in1, in2, out1, out2 : any“
      “rel in1_nmi -> in1 {narrow}“
      “rel in2_nmi -> in2 {narrow} “
      “rel out1 -> out1_ft {narrow} “
      “rel out2 -> out2_ft {narrow} “
      “rel O1: in1, in2 -> out1 {ComputeOut1} “
      “rel O2: in1, out1 -> out2 {ComputeOut2} “
     }
     /* The rest of the class follows here */}
```

Fig. 2. Declarative specification of ModelA

Here the relations *O1* and *O2* show what parameters are actual inputs and outputs to the methods *ComputeOut1* and *ComputeOut2* that is not as clearly visible in the initial class text. The keyword narrow stands for type casting that is in our case *double -> double* and need not any special treatment (like catching possible exceptions).

In the similar way we create the meta-interface to the *ModelB*.

The class *SystA* is presented on Fig 3. The purpose of method *run()* is to get values for variables *in1* and *in2* and calculate the value of *out1* and present it.

```
import ee.ioc.cs.synthesizer.*
public class SystA implements SSPInterface {
     public static String[] SSPspec = {
      "var in1, in2, out1 : any"
      "rel [ModelA |- in1_nmi, in2_nmi -> out1_ft, out2_ft],
         [Translator |- ft -> km],
         [ModelB |- in1_km, in2_km -> out_cm]
         in1, in2 -> out1 {methodX} "
     }
     public double in1, in2, out1;
     public double methodX(double par1, double par2) {
      /* Here we form the structures of input and output
parameters for the subtasks and store the values of par1 and
par2 to input 1. */
      SSP.subtask(1, input1, output1);
      SSP.subtask(2, output1, output2);
      SSP.subtask(3, output2, output3);
      Return output3;
     }
     public void run() {
      in1 = getValue();
      in2 = getValue();
      String progID = SSP.synthesize ("in1,in2->out1", this);
      SSP.execute(progID, this);
      presentValue(out1);
     }
}
```

Fig. 3. Class SystA

The second subtask refers to class *Translator* that could be the following:

```
import ee.ioc.cs.synthesizer.*
public class Translator implements SSPInterface {
     public static String[] SSPspec = {
      "var m, km, cm, mm, dm : double"
      "var ft, kft, inch, nmi : double"
      "rel km = m/1000"
      "rel km = nmi*1.852"
      …
     }
}
```

Now when the method run of class *SystA* calls *SSP.synthesize()* the program synthesizer is executed and searches for a sequence of applicable methods that solve the given task. Note that *in1*, *in2* and *out1*, used in the program synthesis request, are the components of the class *SystA*. The usage of *SSP.synthesize()* method call is discussed in more detail in chapter "Distributed Program Execution".

Distributed Program Synthesis

The *Synthesizer* (see Fig. 4) handles the specifications and performs program construction. The Synthesizer is composed of the following 6 components: *Manager*, *Compiler*, *Decorator*, *Knowledge Base & Component Repository* (*KBCR*), *Planner* and *Code Generator*.

All components of the *Synthesizer* are running under Object Management Group's Common Object Request Broker Architecture (OMG CORBA), which provides a flexible communication and activation for distributed heterogeneous object-oriented computing environments [Vinoski97].

While using CORBA for component interconnection we have to serialize (translate into a byte stream) all the data we want to send over the network, as CORBA is inter platform and inter programming language communication architecture and entities cannot be sent over a network.

Manager is the central component of the *Synthesizer* that coordinates the work of other components and manages the computational resources. When receiving a request for program synthesis from a *Program* the *Manager* first checks whether the *Program* has issued a similar synthesis request before. In this case there is no need to perform synthesis again and the suitable (solver) program can be fetched from the *KBCR*.

KBCR is a database-like structure that allows to store compiled declarative specifications for each class used in the *Program* as well as generated solutions for different problems. This enables the reuse of already synthesized programs.

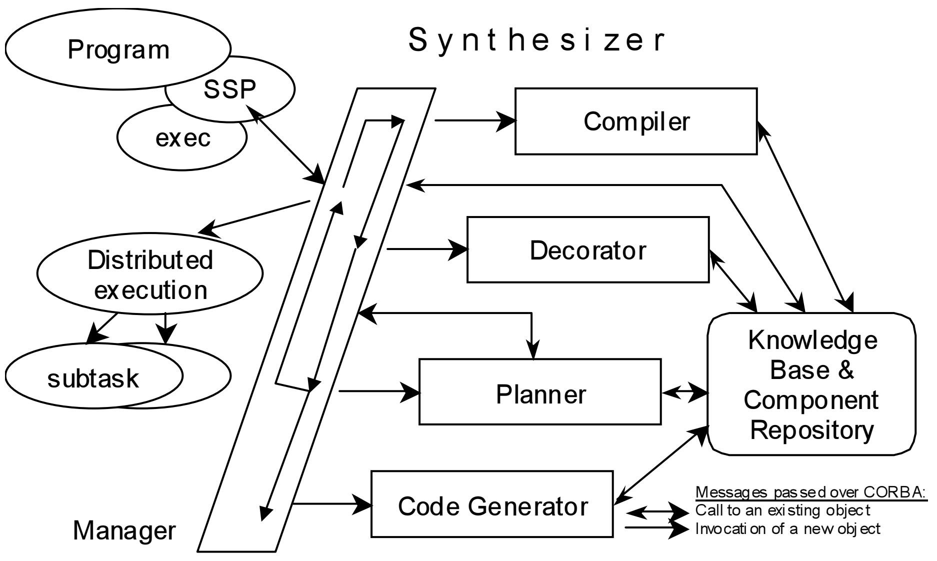
Synthesizer
Program
SSP
exec
Distributed execution
subtask
Manager
Compiler
Decorator
Planner
Code Generator
Knowledge Base & Component Repository
Messages passed over CORBA:
Call to an existing object
Invocation of a new object

Fig. 4. The automated program synthesis environment

In the case the *Program* contacts with the *Synthesizer* first time it "introduces" itself by delivering its classes in addition to the synthesis request. These classes are forwarded to the *Compiler*.

The *Compiler* first fetches the declarative specifications out of the classes and calculates a hash number based on the each specification. Then it compares the results to the hash numbers stored in the *KBCR*. If the hash numbers match the compilation of these classes can be skipped, otherwise the *Compiler* parses the declarative specifications of those classes, which hash code did not match the ones in *KBCR*, and stores the resulting description into *KBCR*.

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

Then the *Decorator* paints the object and variable nodes in the graph. A painted node in the graph represents to a component with the state "known". A different "color" is used to mark the goal(s). Evaluated or "known" is the state of a component when we consider that it has a value i.e. it is not *null* in case of Java.

This structure is the search space delivered to the *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 *KBCR* whether a solution already exists for it. In the case the solution does not exist, the *Planner* starts solving it.

If the solution is found, the algorithm is passed to the *Code Generator*, the problem specification in the *KBCR* is marked as solvable and the caller *Program* is noticed about it.

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 *KBCR*. The component is stored into the *KBCR* with its problem specification that makes it possible to use that component repeatedly for solving many similar tasks.

Planning Strategies

In principle there are two kinds of relations that may occur in the declarative specification [Tyugu81]:

1. Unconditional relations implementing unconditional computability statements of Structural Synthesis of Programs (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 (without subtasks) 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.

The problem specification for Planner is of form x->y, where x denotes the set of "known" objects and y denotes the set of objects to be computed (the goal). While synthesizing programs for sequential execution in a non-distributed environment the Planner creates an algorithm (a sequence of relations) that describes how to compute y from x using the following proof search strategy of Structural Synthesis of Programs [Harf80]:

- An assumption-driven forward search (linear planning) selects unconditional relations. At each step, if all the input objects of the relation are "known" and at least one output object is "not known", the relation is added to the algorithm. When adding a relation into the algorithm all its output objects are set as "known" objects. The search is completed when all the nodes marked as "goal" is also "known" or there are no nodes with "known" inputs left. The search is a simple flow analysis on the network of unconditional relations.
- Goal-driven backward search selects and solves 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 KBCR 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.
- 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.

Well-known search-strategies like depth-first or width-first can be applied here as well as some more-advanced strategies using for example a heuristic criterion. For larger problems the choice of right search-strategy is critical, as one may need to go through the whole search space in a case of wrong strategy choice.

Hence, in the presence of satisfactory amount of distributed computing power, we propose the following changes to the planning strategies.

1. A number of *Planners* is executed in parallel that use different search strategies. *Manager* handles the execution of *Planners* and collects the results (see Fig. 5).

2. In addition to the *Planners*, the *Manager* spawns a *SubtaskSolver* (see Fig. 5) that finds from the given search space (composed by the *Decorator*) all independent subtasks (subtasks to be executed on a separate object) and requests their solving from the *Manager*.

3. Whenever some of the independent subtasks are found to be solvable, their solvability is marked in the *KBCR*, the algorithm is passed to the *Code Generator* and the *Planners* are informed about it. The work of the *Planners* is limited to the given search space. It is inefficient to let them check the solvability of independent subtasks from the *KBCR*.

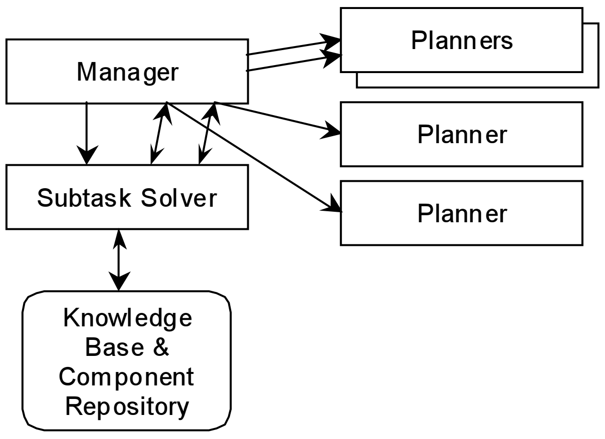

Fig. 5. Distributed planning

4. The independent subtasks are considered "similar" to the ordinary inputs of a method i.e. whenever the independent subtasks are marked solvable and the input variables are "known" the method can be applied included into the algorithm. Hence the methods without and with only independent subtasks can be applied in the phase of linear planning.

5. The dependent subtasks (subtasks to be executed on the context object) are considered only after there are no more independent subtasks to solve and no more relations can be applied by linear planning.

6. The planning is considered finished whenever first of the Planners has finished. The other Planners are then terminated.

In order to solve an independent subtask the *SubtaskSolver* has to create a new instance (object) of the class *ClassName* and then make similarly to the *Program* a synthesis request to the *Synthesizer*. This is necessary because the constructor of the

ClassName may evaluate some of the components that may appear important for solvability.

Distributed Program Execution

Depending on the source data the synthesized program can be executed locally, partially remotely or remotely. In the current framework we consider remote execution usable only on a distributed platform. The local execution is needed in the case our source data cannot be serialized and there are no independent parts of the program that can be executed remotely.

There are three possible cases where distributed execution of the synthesized program can be applied:

1. Execution of the subtasks
2. Solving equation and inequation systems
3. Parallel execution of the main synthesized program

We aim to make the environment as user-friendly as possible and try to hide all the distribution related tasks into the system. However, in case of subtask execution, user still has to decide how the execution is performed.

In engineering computation and especially in case of modeling and simulation one needs quite often to perform similar computations on an array of data. It may be a number of measurement results, different parameters of given model or a number of case inputs for an optimization task.

In those cases such computations can be declared as subtasks. The benefit of the subtasks is that we do not need to implement needed programs beforehand and let the synthesizer to do it. In the program we have to just pass proper data to the synthesizer program and get our computations done.

However, passing the data to the subtask program is somewhat complicated. As we need to make a call to an existing Java method, which then forwards the input data to the real program, the subtask call is generalized to the form:

```
SSP.subtask(specNo, InputPars, OutputPars);
```

Here the *InputPars* and *OutputPars* are structures that must be composed before the call for subtask execution. In case we want parallel execution of the same subtask on multiple data the subtask call is slightly different:

```
SSP.subtasks(specNo, InputPars[], OutputPars[]);
```

In other words we need to send an array of input parameters and receive an array of results. Hence the two following Java program fragments yield to the same results:

```
for (int i=0;i<inputs.length();i++) {
 input = inputs[i];
 SSP.subtask(1, input, output);
 outputs[i] = output;
}
SSP.subtasks(1, inputs, outputs);
```

The only difference of these program fragments is that the first one is executed in sequential order but the second one may be executed in parallel. One of the following criteria has to be fulfilled to enable parallel execution of subtasks:

1. The subtask is independent (can be executed on a separate object)
2. The subtask is dependent (must be executed on the context object) and the context object is serializable.

These criteria are detected automatically and if the current situation does not match to any of these criteria, sequential execution is used.

Considering our example about *SystA* we can think of a modeling case where we need to calculate a number of out1's based on a number in1's and in2's. In this case the methods *methodX* and *run* should be only slightly changed:

```
    public double[] in1, in2, out1;
    public double[] methodX(double[] par1, double[] par2) {
     /* Here we form the structures of input and output
parameters for the subtasks and store the values of par1 and
par2 to input 1. */
     SSP.subtasks(1, input1, output1);
     SSP.subtasks(2, output1, output2);
     SSP.subtasks(3, output2, output3);
     Return output3;
    }
    public void run() {
     in1s = getValues();
     in2s = getValues();
     String progID = SSP.synthesize (“in1s,in2s->out1s“,
this);
     SSP.execute(progID, this);
     presentValues(out1s);
    }
```

Selection of proper solver for a given equation or inequation system may be crucial task. We should consider the speed and accuracy of the solvers as well as their capabilities. In the current framework the selection of solvers can be automated.

The easiest way to solve task is to select a number of solvers and execute them in parallel. When one of the solvers has finished its result is used and the others are just terminated.

More complex ways of solver usage can be introduced. The results of different solvers can be collected and most common solution selected or partially sequential execution of solvers can be used when none can handle the problem alone [Monfroy03][Petrov02][Caseau01].

It is the task of Planner to find appropriate parts of the synthesized program that can be executed in parallel. Search for these parts is made during the minimization stage. However, in practical tasks one should not expect significant speedup from distribution of the main synthesized program, as it is impossible to predict the execution times of given relations unless some additional information is available. This kind of information can be learned while solving similar tasks and executing similar programs many times.

Summary

This paper discusses the possibilities to take advantage from distributed execution of program synthesizer and synthesized program parallel execution. We describe declarative specifications embedded to Java programming language to support automated program construction, the program synthesizer that performs automated program construction uses these specifications and several aspects how in this framework a new program can be synthesized and executed in a distributed manner.

Changes to the planning algorithm are proposed to take advantage of the distributed execution of the synthesizer and speedup the program construction process.

As the input data and synthesized programs can be transferred to the location providing satisfactory computational resources we believe that this approach could be useful for GRID computations.

References

[Kotkas02] Kotkas, V.: A distributed program synthesizer. Acta Cybernetica 15 (2002), pp 567-581

[Vinoski97] Vinoski, CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communications Magazine, vol. 14 (1997).

[Tyugu81] Tyugu, E.: The structural synthesis of programs. Lecture Notes in Computer Sciences, Vol. 122, 1981, pp. 290-303.

[Harf80] Harf, M., Tyugu, E.: Algorithms of structured synthesis of programs. Programming and Computer Software, 6, (1980), pp 165-175.

[Monfroy03] Monfroy, E., Castro, C.: Basic Components for Constraint Solver Cooperations. In Proceedings of the 18th ACM Symposium on Applied Computing (SAC'2003), ACM Press (2003).

[Petrov02] Petrov, E. Monfroy, E.: Automatic analysis of composite solvers. In Proceeding of the 14th Int. Conf. On Tools with Artificial Intelligence (ICTAI), IEEE Computer Society (2002).

[Caseau01] Caseau, Y., Silverstein, G., Laburthe, F.: Learning hybrid algorithms for vehicle routing problems. TPLP 1 (2001) pp 779-806.