Implementing Extended Structural Synthesis of Programs

Sven Lämmermann

Department of Microelectronics and Information Technology, KTH
laemmi@it.kth.se

Enn Tyugu

Estonian Business School
tyugu@ebs.ee

Abstract

We discuss logic and specification language of a program synthesizer for Java intended for dynamic synthesis of services. We introduce metainterfaces as logical specifications of classes. They can be used in two ways: as specifications of computational usage of classes or as specifications of new classes composed from metaclasses – the classes already supplied with metainterfaces. The specification language used has a straightforward translation into a restricted first-order constructive logic. New programs are composed by a deductive synthesis method close to the structural synthesis of programs. An example of composition of a transaction service is presented.

Introduction

We discuss logic and specification language proposed in an ongoing project aimed at the development of a program synthesizer for Java intended for dynamic synthesis of network services. A prototype of the synthesizer has already shown good performance. In this project, we prioritize the efficiency of proof search before the generality of the logical language. The method under consideration is deductive program synthesis. In essence, it is an extension of the structural program synthesis (SSP) described already in (Mints and Tyugu 1982), and we call it ESSP – extended structural synthesis of programs.

Our discussion is related to two quite different knowledge domains: software composition and logic. In the introductiuon we briefly look at the features of software that must be representable by the logic of the synthesizer. We explain here as well which features of logic will be used to support the required software features, and show how the complexity of proof search depends on the features covered by the logic. As expected, supporting all desirable features of the software composition leads in principle to exponential time complexity of the proof search. However, our proof search strategy enables us to construct a coarse-grained proof quite efficiently using admissible inference rules that correspond to actual applications of functions and control structures in computations. We are still quite satisfied with the efficiency of the application of the synthesizer to practical problems.

Our synthesis method is aimed at composing software from functions, including higher-order functions (which are taking automatically synthesized functions as arguments, and can so represent control structures, e.g. loops and conditional statements). Higher-orderness guarantees the generality of the method in principle – one can preprogram a small set of higher order functions that are sufficient for representing any computable function. In practice, we use a specific set of preprogrammed functions for every problem domain. These functions are implemented as methods of classes written in Java. We distinguish three kinds of building blocks used in the software synthesis process:

- The smallest building blocks (we call them also modules) are methods; they are selected and connected automatically by the synthesizer.
- The next size blocks are Java classes representing particular concepts.
- The largest are metaclasses that represent independent software components and are specified declaratively.

The software features we are going to express in logic are the following:

- Dataflow between pre-programmed modules
- Order of application of modules (control variables)
- Hierarchical data structures
- Usage of pre-programmed control structures (subtasks)
- Alternative outcome of modules, in particular exception handling
- Implicit linking of modules

1. *Dataflow.* Dataflow dependencies between pre-programmed functions can be expressed simply by implications, where for each function there is an implication. On its left hand side is a conjunction of propositions stating for each input variable (for each argument) that the input is available and on the right had side is a conjunction of propositions stating for each output variable that the output is computed. This conforms with the intuitionistic realizability of implications as functions and propositional variables as data. Consider the following example. Let *f* and *g* be modules with input variables *a*, *b* respectively, and output variables *b*, *c* respectively. Then the logical description of the dataflow will be $A \supset B$ and $B \supset C$ where *A*, *B*, *C* are propositions stating availability

(computability) of variables *a*, *b*, *c*. Computability of *c* from *a* follows from derivability of *C* from *A*.

2. *Order of module application*. We are sometimes confronted with the situation where the order of module application matters, but this is not determined by dataflow. To overcome this problem, a logic must, in addition, include means to force a certain order of module application. For instance, let the modules *f* and *g* from the example above be two methods of a class *D* implemented in an object oriented programming language. Before these two methods can be applied, a constructor *d* of the class *D* must create an object of class *D*. By adding a proposition, let's say *D* (as an additional conjunct) to the right hand side of the implication specifying the constructor *d*, and adding the same proposition to the left hand sides of the implications specifying *f* and *g*, the correct order of application is forced on the modules. Fortunately, this does not require any extension of the logic compared to the logic of dataflow.

3. *Hierarchical data structures.* Hierarchical data structures can be specified by means of structural relations that bind a structure, e.g. x with its components, say $x_1,\ldots,x_k$. The computational possibilities of this relation can be expressed by the implications $X_1 \wedge \ldots \wedge X_k \supset X$ and $X \supset X_i$, for $i = 1,\ldots,k$ where *X* and X_i denote again the computability of *x* and x_i respectively. This does not require extension of the logic used for expressing the dataflow. (Object-oriented environments may require some additional computations for obtaining a structured value, e.g. application of a constructor of a proper class.)

4. *Usage of predefined control structures.* As we have noted already, some building blocks that have to be specified are higher order functions that take other functions as input. A functional input *f* of a function *g* can be specified by an implication on the left hand side of the implication specifying the function *g*. For instance, the formula $(I \supset A) \wedge N \supset S$ specifies a function *g* with arguments *f* and *n* that computes *s*, where the function *f* can be any function computing *a* from *i* that is composed from given blocks. (The capital letters again denote the computability of the variables denoted by the respective small letters.) Accepting nested implications as formulae is an essential extension of the logical language. Indeed, any propositional formula of the intuitionistic logic can be encoded by formulae of this language so that its derivability can be checked in a theory within this implicative language, see for instance (Mints and Tyugu 1982).

5. *Alternative outcome of modules*. Even considering only dataflow, we can meet a situation where a module may produce one of several alternative outputs every time it is applied (represented by branching in a dataflow). In particular, this is the case with exception handling – we expect that a normal output will be produced, but must be ready to handle exceptions, if they appear. This requires introduction of disjunctions on the right hand side of implications.

6. *Implicit selections and linking of software components.* Our intention is to use the synthesizer for putting together large programs from components. We mean by a component not just a function, but also a software configuration providing some useful functionality. We expect to have libraries of components supporting various problem domains, like packages in Java. So the problem of automatic selection components appears. This requires a language where we can state that a component with particular properties exists or, vice versa, it may be required. Hence, we need existential and universal quantifiers. However, we can restrict the usage of quantifiers so that we will not need the whole power of first order logic.

Extended structural synthesis covers the features discussed above as follows:

- Non-nested implications of conjunctions of propositions cover 1, 2, and 3,
- Nested implications cover also 4,
- Disjunctions cover 5, but can be reduced to nested implications,
- Restricted first-order logic covers 6.

The proof search for non-nested implications has linear time complexity. Adding nested implications gives us the general case of intuitionistic propositional logic, hence exponential time complexity in general; however, we can use a modal logic for nested implications and get simpler proof search, see (Mints 1991). In the case of first-order logic we use quantifiers in a very restricted way, and hope to keep the search manageable.

We have chosen Java as the language we extend, because it is a widely used object-oriented language suitable for implementing services in a network. Components of services are represented as classes and interfaces. Special attention is paid to handling exceptions in Java in a logically sound way. Also handling constructors requires special attention. This has required some extension of the logic used in (Lämmermann 2000). Considerable part of the presentation takes an example of synthesis of a banking service implemented in our project. This example is introduced gradually by presenting Java classes on various parts of the paper.

Specifications as Metainterfaces

On the specification language side we introduce metainterfaces as logical specifications of classes. They can be used in two ways: as specifications of computational usage of classes or as specifications of new classes composed from classes already supplied with meta-interfaces. A **metainterface** is a specification that:

- Introduces a collection of **interface variables** of a class.

- Introduces **axioms** showing the possibilities of computing provided by methods of the class, in particular, which interface variables are computable from other variables under which conditions.
- Introduces **metavariables** that connect implicitly different metainterfaces and reflect mutual needs of components on the conceptual level.

Interface variables are abstract variables that get implementations as class- or instance variables, only if they will be used in computations by a synthesized program.

Axioms that we use are implications with preconditions for applying a method on the left- and postconditions on the right-hand side. Interface variables denote in preconditions that the variable must have a value before the method is applied, and in postconditon – that its value will be computed. (There are other kinds of subformulas in axioms that we are going to discuss later). For instance, having some class written in Java with two methods `getExchangeRate` and `getLocalCurrency` for calculating the exchange rate of a given currency and the local currency, we can introduce interface variables `currency` for a source currency, `localCurrency`, and `exchangeRate`, for the local currency and exchange rate respectively, and declare an axiom that specifies how one can use calculation of exchange rate. Here is a fragment of the metainterface: these are interface variables:

```
currency, localCurrency : String
exchangeRate : float
```

Here follows the specification of computability (the arrow -> is used in our language as an implication sign for convenience of typing):

```
currency,localCurrency -> exchangeRate {getExchangeRate}
-> localCurrency {getLocalCurrency}
```

If we wish to specify that the local currency can be found from a class `LocalCurrency`, we can use a metavariable `(LocalCurrency)` and replace the specification:

`-> localCurrency {getLocalCurrency}` by
`(LocalCurrency) -> localCurrency {assign}`

The last extension enables the synthesizer to detect automatically local currency that may depend on the location where the service will be used, if it is specified in some accessible metainterface (by unifying metavariables of different metainterfaces). The class `Exchange` extended with a metainterface as a static component `spec` is as follows:

```
class Exchange implements java.io.Serializable {
  static String[] spec = {
    "currency, localCurrency : String",
    "exchangeRate : float",
    "currency,localCurrency->
                 exchangeRate{getExchangeRate}",
    "-> localCurrency {getLocalCurrency}"
    "-> (Exchange) {Exchange}"
  };
  float getExchangeRate(String currency,
                        String localCurrency) {…}
  String getLocalCurrency() {…}
}
```

A metainterface can be used as well for specifying how an application should be composed from components that are supplied with metainterfaces. In the latter case, a new class can be built completely from a specification of its metainterface. For this purpose, specifying equality relations between some interface variables of the new class may be needed. An extended example of the usage of metaclasses is presented in the section Example. Here we present a metaclass for calculating exchange rates that uses another metaclass as a component.

```
class ExchangeRate implements java.io.Serializable {
  static String[] spec = {
    "currency : String",
    "exchangeRate : float",
    "exch : Exchange",
    "exch.currency = currency",
    "exch.exchangeRate = exchangeRate",
  };
}
```

The class `ExchangeRate` can be used for several purposes. The actual usage will be determined by a goal. Goals can have different forms, for instance, `ExchangeRate|-currency->exchangeRate` gives a program for computing the `exchangeRate` of a given `currency` and the local currency. The following definition of the method implementing the given goal is fully automatically built for the class `ExchangeRate`, and the name `spec_1` is automatically generated for it.

Realization of `currency -> exchangeRate {spec}`

```
float spec_1 (String currency) {
  Exchange ExchangeRate_exch_obj = null;
  ExchangeRate_exch_obj = new Exchange();
  String ExchangeRate_exch_localCurrency = null;
  ExchangeRate_exch_localCurrency =
      ExchangeRate_exch_obj.getLocalCurrency();
  float ExchangeRate_exchangeRate = 0;
  ExchangeRate_exchangeRate =
    ExchangeRate_exch_obj.getExchangeRate(currency,
                ExchangeRate_exch_localCurrency);
  return(ExchangeRate_exchangeRate);
}
```

The number of possible different computations described by a class or an interface is the number of its methods. Using metainterfaces, a user gives a goal of computation, and names data items to be computed or given as arguments. This enhances flexibility in two ways: first, the number of possible computations is at least the number of available methods, but can be much larger and often is. Second, component deployment does not require a user to know names of methods. For example, in Java one has to compose streams from a large number of classes for programming input and output. It is easy to build a new class `InputOutput` including streams as components and with a metainterface that will enable one to do the composition of streams automatically. To specify some input or output in a program, one has to give a goal, e.g. to put a string into a file one can write

```
InputOutput |- stringValue, filename -> file
```

that can be read as “take `stringValue` and `filename` and create `file` using the class `InputOutput`”.

General Scheme of Synthesis of Classes

We are using a concept of **metaclass** in order to distinguish a class supplied with a metainterface from a conventional class. In principle, it is unimportant, whether the metaclass is considered as one single text or its metainterface part is kept separately like different parts of a Java bean. In our implementation we keep the specification of a class as a component of the class in the form of an array of strings. This allows us to implement a metaclass as a regular Java class.

Our language of specifications has a straightforward translation into a restricted first-order constructive logic (see section Logical Semantics of Specifications). Axioms have a realization given by a method of the class to whose specification the construction belongs. In particular, an implication like `->localCurrency` may have an implementation in the form of a method, e.g.

```
String getLocalCurrency().
```

Then we denote it in the specification by showing a name of the respective methhod in curly brackets:

```
-> localCurrency {getLocalCurrency}
```

Development of a synthesized primary class `C` proceeds as follows. One writes a specification of a new metaclass, using existing metaclasses, and also writes a top-level goal in the form of an implication `P->R` where `P` is precondition and `R` is postcondition of the main method of the new class to be synthesized. So, the goal is to find a realization for the implication `P->R` in the form of a new method of the new class `C`. This is achieved by a conventional scheme of the deductive program synthesis: derive `P->R` using logical formulae of specifications as specific axioms, and extract the realization of the goal from its derivation, see (Manna and Waldinger 1992). The realization of the goal is executed in the method `main` of the class `C`. New subgoals may appear during this derivation, hence, more methods of the class `C` may be synthesized completely automatically. Also instance variables of the new class may be introduced. In particular, any proposition with the meaning "the interface variable `x` has a correct value" may get a realization as an instance variable of the new class.

Summarizing this section we can say the following:

- We compose specifications manually and synthesize software automatically in the form of new classes using the composed specifications.
- Software components are metaclasses, each of which includes two parts: 1) a realization in the form of an ordinary OO class, and 2) a metainterface, i.e. a specification that describes the possible computational usage of methods of this class.
- Correctness of the realization of a component with respect to its specification has to be guaranteed by the developer of the component.
- Correctness of the synthesized class with respect to its specification is guaranteed by the correct implementation of the synthesizer.

Specification Language

Our aim in designing the specification language has been to make it as convenient as possible for a software developer. This language should allow a simple and exact translation into a language of logic used in the synthesis process, but we have tried to avoid excessive use of logical notations. The following design decisions have been made.

- We have separated interface variables from instance- and class variables (attributes of classes). This gives flexibility to specifications and enables one to specify existing classes developed initially without considerations about their appropriateness for synthesis. Moreover, by separation of interface variables from attributes of metaclasses the creation of new side effects is avoided.
- Inheritance is supported in specifications, but without overriding in the specification part. This may cause inconsistencies, if an implementation of some relation is overridden. Fortunately, this situation can be detected automatically by introspection of classes.

Core of the Language

Specification of an interface variable is a declaration of the form `<identifier>:<specifier>`, where the identifier is a name of the new variable declared, and the specifier is one of the following phrases:

- Any of the primitive types of the underlying OO language, `int`, `float` for instance
- A class name or a metaclass name.

It denotes that, if the variable is used in computations, it will require a realization as an instance variable of the given type.

Binding is an equality of the form `<name`$_1$`>=<name`$_2$`>` denoting that realizations of the two interface variables must have equal (may be partial) values. By the partial values we mean that even if only some subcomponent of the variable given by `name`$_1$ has a value, the respective subcomponent of the other variable has the same value and vice versa.

Name may be an identifier or a compound name, e.g. `a.b.c` denoting the component `c` of the component `b` of the interface variable `a`.

Our aim is to support compositional software development and to enable one to specify new software by synthesizing it from specifications that do not include other declarations than component specifications and bindings, where components are represented by interface variables of suitable classes. The experience shows that one has to be able to add some "glue" to components – by adding new relations that bind them. This can be done by manually programming some methods of the new class and specifying them by means of axioms. The axioms are used also for specifying classes that have a role of components. In this case, the aim is to specify all computational possibilities that a class provides.

Axiom is a logical formula specifying in some way the possibilities of computing provided by methods of the class. The concrete syntax of axioms depends on the logic used for program synthesis. Here and in the examples we shall use the logic implemented in (Lämmermann 2000). Here an axiom is a declaration of computability in the form of an implication with a list of preconditions separated by commas interpreted as conjunction symbols on its left-hand side and postconditions on its right-hand side, followed by its realization – a method or constructor name in curly brackets, e.g.

```
currency, localCurrency ->
exchangeRate {getExchangeRate}          (1)
```

An axiom states that the method can be applied, if all its preconditions are derivable. Postconditions are separated by commas as conjunction symbols, and by disjunction symbols |. Postconditions are interface variables or propositional variables taken in square brackets. The meaning is that if a postcondition is derivable then the variable with this name is computable. Disjunction specifies a possibility of alternative results of computation, e.g.

```
amount, currency, [currency->exchangeRate] ->
[debit_t] | exception {debit}           (2)
```

- A precondition is either an interface variable or a logical formula in square brackets. In this way we keep the language logically extensible — new forms of formulae can be used in square brackets. At present, a formula in square brackets can be just a propositional variable or an implication with lists of interface variables on its left- and right-hand sides. The semantics of preconditions is the following:
- A precondition in the form of an interface variable means that a value of this variable must be computable.
- A precondition in the form of an implication denotes a subgoal: computing values for the interface variables on its right-hand side from given values of the interface variables on its left-hand side. We call this a **subtask**. Realization of a subtask is a synthesized method.
- A precondition in the form of a propositional variable in square brackets does not have any computational meaning, but must be derivable when the axiom is used in a proof.

The specification (2) uses the propositional variable `[debit_t]` and interface variable `exception` in postconditions. The method `debit` may throw an exception. This is specified by the disjunct `exception` in postconditions. The specification (2) asserts that if the method `debit` does not throw an `exception` then the transaction to debit the bank account associated with the current computation was successful.

Realization of an axiom is a method or constructor of the class to which the specification belongs. Input parameters of the realization are associated with its preconditions positionally. For instance, the realization of `currency` of the specification (1) is passed as the first parameter to the method `getExchangeRate` and the realization of `localCurrency` as the second parameter. Realization of a subtask is passed as an object that implements an interface called `Subtask` that has a synthesized method `subtask` for solving the subtask. For instance, the realization of the subtask specified by `[currency->exchangeRate]` of the method `debit` of the specification (2) will be passed to the method `debit` as object of type `Subtask`.

Extensions

We have introduced several extensions to the core language. Here we consider two of them. The first concerns first-order features and requires extending the logic of axioms, all other extensions can be translated into the logic of the core language.

Metavariables as pre- and postconditions. A metavariable `(Q)` as a precondition is satisfied if any object of the class `Q` is found. A metavariable as a postcondition denotes that an object of the given class `Q` will be computed. Instead of a component name we use a class name in parentheses in this case, e.g.

```
no -> (Account) | (Exception) {getAccount}      (3)

amount,currency,[currency->exchangeRate]->
[debit_t]|(Exception) {debit}                   (4)
```

The second disjunct of both specifications is the metavariable `(Exception)`. The formula that specifies the method `debit` is the same as we used before in (2), but the interface variable `exception` has been replaced with the metavariable `(Exception)`. Both specifications state that exception handling must be provided by some other metaclass, which is detected automatically during synthesis if a suitable metaclass for exception handling exists. In our example we shall use the following metaclass for handling exceptions:

```
class ExceptionHandler {
  static String[] spec = {
    "(Exception) -> [any] {handleException}"
  };
  static void handleException(Exception e) {…}
}
```

This metaclass provides a method that takes as input any object of class `Exception`. The method `handleException` displays an error message depending on the given exception. The propositional variable `[any]` in the postcondition of the specification of method `handleException` can be used outside this metaclass to specify logic of computations after exception handling. For instance, the binding (as used in the metaclass `Account`, which we shall describe in section Logical Semantics of Specifications)

```
[debit_t] = [ExceptionHandler.any]
```

specifies that realizability of the propositional variable `[debit_t]` is the same as realizability of the propositional variable `any` of the metaclass `ExceptionHandler`. This binding is needed to handle exceptions in a logically sound way.

Equations. The synthesizer and the run-time environment support the usage of linear equations in calculations. We have experience in using the equations in earlier synthesizers (Tyugu 1996) and find this feature very convenient for gluing together components.

Instance value is a special concept needed for distinguishing an instance of a metaclass from a structure composed from realizations of all its interface variables. We write `x.obj` instead of `x` to denote that a value computed for the interface variable `x` is an instance of the class of `x`, but not the structure constructed from its instance variables. The name `obj` is a keyword of the specification language. For examples we refer to the definition of method `spec_1` in section Specifications as Metainterfaces and to the example in section Example.

Logical Semantics of Specifications

We implemented initially the same semantics of specifications that was in the PRIZ and NUT systems (Mints and Tyugu 1982) (Mints 1991) and were able to use the structural synthesis of programs (SSP) as described already in (Tyugu 1996). This guaranteed good scalability and high performance of the synthesis. However, the logic of SSP was too restricted for expressing some important properties of Java programs, first of all – throwing exceptions. Second, context-awareness that is an important feature of the synthesis of services required another extension – introduction of predicates that describe the appropriateness of some computable objects for binding independently developed components. Therefore we have extended the logic with disjunctions and metavariables. Here we are going to describe the translation of constructions of the specification language into a language of logic and then outline the usage of the logic in the program synthesis process.

Structural properties of objects are represented in our logic computationally. We avoid the usage of description logic (Premkumar, Devanbu, and Jones 1994) for representing structures, although it has been designed for this purpose. In this way we keep the control over proof search and preserve its high performance. The semantics of a metainterface `X` having interface variables $x_1,\dots,x_k$ is expressed computationally by the axioms $X \rightarrow x_1,\dots,x_k$ and $x_1,\dots,x_k \rightarrow X$ where identifiers denote the computability of variables with the same names. (This conforms to our usage of names of variables in axioms in section Specification Language.) These formulae express that the structural value is computable when all its components are given, and vice versa - all its components are computable when the structural value itself is known. Besides that, we have to take into account the structures when equalities are present. In the case of equality of two structures, e.g. `x=y` (`x:X`, `y:X`) we have to be able to infer the computability of any of the components of any of them as soon as a respective component of the other is given. An easy way to guarantee this property in our logic is to unfold the equality explicitly for all components, i.e. to write automatically `x.z=y.z` for all components `z` of `x` and `y`.

Axioms written in a specification are translated into logic in a straightforward way. They are implications with preconditions on the left and postconditions on the right side. If a method may throw an exception, then it is described by an implication with disjunction on its right side. One disjunct specifies normal execution and the other – throwing an exception, e.g.

```
url -> (Bank) | (Exception) {Naming.lookup}
```

Handling disjunctions requires introduction of a new search strategy – search of proof of the innermost goal. This extension is thoroughly described in (Lämmermann 2000). As we have said it earlier, the nested implications are **subtasks**. The general form of an axiom for a method or constructor with `m` subtasks, `n` input variables, `s` output variables and throwing exceptions is:

$$(u_{1,1}\wedge\dots\wedge u_{1,k1} \rightarrow v_1)\wedge \dots \wedge(u_{m,1}\wedge\dots\wedge u_{m,km} \rightarrow v_m)\wedge x_1\wedge\dots\wedge x_n \rightarrow y_1 \wedge\dots \wedge y_s \vee z$$

Metavariables are translated into the logic with quantifiers: as a universal quantifier among the preconditions and an existential quantifier among the postconditions. Examples are as follows:

- $(C)\wedge x\rightarrow y$ gives $\forall w(C(w)\wedge x\rightarrow y)$
- $u\rightarrow v\wedge(C)$ gives $u\rightarrow v\wedge\exists w C(w)$.

Inheritance is straightforward – all axioms of a superclass are added to the axioms of the class. Metainterfaces of Java classes and Java interfaces are handled in one and the same way. This leads to multiple inheritance in metainterfaces.

Unfolding is used for representing the semantics of interface variables whose type is specified by a metaclass. Let us consider the metaclass `ExchangeRate` described in section Specifications as Metainterfaces. Its logical semantics is expressed by a collection of formulae. If a new interface variable `exchR:ExchangeRate` is specified (as it is in the class `Account` shown below), then these formulae are explicitly inserted into the logical specification and prefix `exchR` is added to all names in these formulae. This leads to expansion of the specification, but avoids excessive usage of universal quantifiers. An example is a Java remote interface Account:

```
interface Account extends java.rmi.Remote {
  static String[] spec = {
    "amount : int", "currency : String",
    "exchangeRate : float", "exch : Exchange",
    "exch.localCurrency = currency",
    "exchR : ExchangeRate",
    "exchR.currency = currency",
    "exchR.exchangeRate = exchangeRate",
    "[debit_t] = [ExceptionHandler.any]",
    "amount,currency,[currency->exchangeRate]->
                           [debit_t]|(Exception) {debit}"
  };
  void debit(int amount,String currency,Subtask sub)
      throws RemoteException;
}
```

The metaclass Account we have presented here provides a remote transaction debit to debit an account. Its metainterface declares five interface variables: amount, currency, exchangeRate, exch, and exchR. We use the propositional variable [debit_t] to specify the final state of a complete transaction that involves the method debit and exception handling. The binding [debit_t]=[ExceptionHandler.any] is needed to synthesize a branch for handling exception. Let us use Account to illustrate the change in specifications if we unfold this metaclass. The first prefix to be added to all names of interface variables and propositional variables (excluding metavariables) is the name of the metaclass, which is Account. For instance, all occurrences of the interface variable amount become Account.amount. The axiom specifying the method debit is replaced by:

```
Account.amount,Account.currency,
[Account.currency->Account.exchangeRate]->
[Account.debit_t] | (Exception) etc.
```

Metavariables are left unchanged, e.g. (Exception) remains as it is. The unfolding is done hierarchically, if needed. Hence, long compound names occur. We use compound names also in generated source code, but substitute an underscore (_) for a dot (.) in a compound name in order to conform to the Java programming language syntax, for instance, see automatically built method spec_1 described above.

Java classes may have instance methods and class methods. Class methods are invoked on a class, not on an instance. Our specification language does not reflect this fact. Fortunately, we can gather information about methods by introspection of classes while unfolding, and call an appropriate constructor of a class first and create its instance, if its instance methods are used in a synthesized program.

Example

The example is synthesis of a transaction program implemented in a distributed way using Java RMI. The program implements a service — debiting a bank account. The metaclasses needed for specifying this service are Bank, Account, Exchange, ExchangeRate, and ExceptionHandler. Fortunately, we have already specified these metaclasses except Bank. The metaclass Bank is a remote interface:

```
interface Bank extends java.rmi.Remote {
  static String[] spec = {
    "no : long",
    "url : String",
    "url -> (Bank) | (Exception) {Naming.lookup}",
    "no -> (Account) | (Exception) {getAccount}",
  };
  Account getAccount(long no) throws RemoteException;
}
```

The metaclass Bank we present here provides only one remote method getAccount, but reuses the class method lookup of the class Naming. The method lookup is used to obtain a remote reference to a bank object that is associated with a given url (Uniform Resource Locator). This remote reference is then used to obtain a remote reference (by invoking the method getAccount on it) to an account object that is connected to a given account number no. Having access to an account object we can debit the corresponding account by invoking the method debit on the remote reference if we can solve the sub-problem to calculate the exchangeRate of the local currency and the currency of the location of the bank. Some methods of our metaclasses may throw an exception, which is specified by using disjunction in the postconditions of the specifications of the respective methods. Due to disjunction, we have to solve additional sub-problems that take over the computation in case of exception.

It is our goal to derive a program to debit an account at a bank. The input for this program is the url of a bank, the account number no, and the amount of how much the respective account should be debited. To synthesize this program we state the following goal:

```
Bank, Account |- Bank.url, Bank.no, Account.amount ->
                      [Account.debit_t]
```

The list of metaclasses (Bank, Account) of the antecedent of this goal does not contain all metaclass that are needed to synthesize our program. The synthesis involves also metaclasses Exchange, ExchangeRate, and ExceptionHandler. The deployment of the metaclass ExceptionHandler is implicitly specified by the metavariable (Exception), where the deployment of the metaclasses Exchange and ExchangeRate is explicitly specified in metainterfaces.

After unfolding the metainerfaces of our metaclasses we obtain a flat representation of all formulae that specify methods and constructors of classes. The list of all needed formulae, after unfolding, is the following:

```
Exchange
C1  ≡ -> (Account.exch.obj)
getLocalCurrency
C2  ≡ [(Account.exch.obj)] ->
      Account.exch.localCurrency
ExchangeRate
C3  ≡ -> (Account.exchR.obj)
spec_1
C4  ≡ [(Account.exchR.obj)],
      Account.exchR.currency ->
      Account.exchR.exchangeRate
handleException
C5  ≡ (Exception.obj) -> [Exception.any]
Naming.lookup
C6  ≡ Bank.rul -> (Bank.obj)|(Exception.obj)
getAccount
C7  ≡ [(Bank.obj)],Bank.no ->
      (Account.obj)|(Exception.obj)
debit
C8  ≡ [(Account.obj)],Account.amount,
      Account.currency,[S] ->
      [Account.debit_t]|(Exception.obj)
S   ≡ Account.currency -> Account.exchangeRate
E1  ≡ [Account.debit_t]=[Exception.any]
E2  ≡ Account.exch.localCurrency=Account.currency
E3  ≡ Account.exchR.currency=Account.currency
E4  ≡ Account.exchR.exchangeRate=Account.exchangeRate
```

Each implication specifies input/output conditions of a component `C1`, …, `C8` (which encapsulate a method or a constructor). For convenience we have named the bindings by `E1`, …, `E4`. We replaced the subtask specification of the formula that specifies component `C8` by the proposition `S`, where `S` is a formula. The schema in Figure 1 depicts our synthesized transaction program.

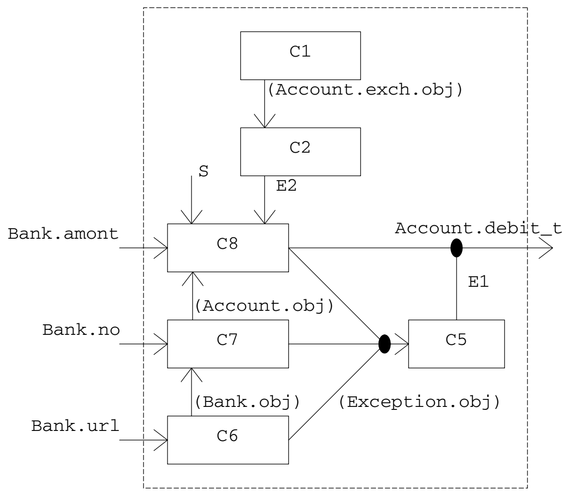

Figure 1. Program schema.

Each box represents a component. Arrows leading from one component to another represent dataflow in computation and component composition in synthesis.

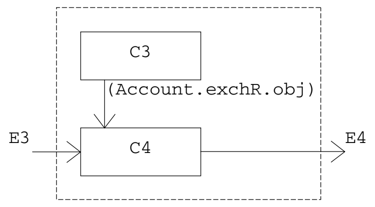

Figure 2. Subtask schema.

The component `debit` receives as input the subtask `S`, which calculates the exchange rate. A separate schema, Figure 2, presents the subtask `S`.

Concluding Remarks

On the practical side, early Java environments suffered from the difficulties in the reuse of classes (first of all, related to GUI). As a result, JavaBeans were introduced. This was followed by the introduction of Enterprise Java Beans (EJB) as components of domain-oriented applications. In both cases, classes were supplied with additional information, and mechanisms (e.g. beanboxes) were developed for using this information. The practicality of program synthesis, not especially related to object-oriented software development, has been demonstrated by several software development systems, e.g. (Blaine et al. 1998) (Stickel et al. 1994).

We introduce an extension of classes that is supported by a domain-independent program synthesis technique, already tested in practice to some extent. As this work is performed in the context of a larger project Personal Computing and Communication (PCC), we have a practical goal to apply our technique to just-in-time synthesis of services from preprogrammed components. In this work we use a logic and a specification language close to those that have been tested in practice (Tyugu, Matskin, and Penjam 1999). The logic is expressive enough for describing, first, structure of hierarchical configurations, second, dataflow between the components, and third, mutual needs of the components of a service – bindings between objects of separately implemented components. The proposed composition model uses a logical proof as a justification of correct deployment of components in the context of their use. We are well aware of tradeoffs between the expressiveness and efficiency of automatic usage of logic, and have chosen in some sense minimal logic that is still universal, i.e. enables us in principle to specify any computable function. This gives the efficiency of search needed in the composition process.

In the systems of structural synthesis (Tyugu, Matskin, and Penjam 1999) (Tyugu 1996) the potential components must be explicitly visible from specifications (after unfolding a specification), and selection of components actually included into a synthesized program is performed on the basis of higher-order dataflow. This is very restrictive in the case of synthesis of services, because of changing context of a service, but it facilitates the proof search. In the present work, we have extended the logic by introducing metavariables in such a way that components of synthesized software can be selected without explicit reference to them. One has to circumscribe, however, the context where the search of components is performed in every particular case. In other words, one must be able to decide for each metaclass, whether this metaclass may be used in the synthesized service. This process is not supported by our system. We hope to use our synthesis together with the component selection method proposed in (Penix and Alexander 1997) using the latter for the preselection of potential components and using our synthesis for adaptation and binding the selected components.

Some words have to be devoted to the performance of the synthesizer. The performance of compilation is not critical. Support of introspection and dynamic compilation of classes in Java has helped the implementation. Some experimental performance evaluation can be found in (Lämmermann 2000). The critical phase is proof search. Here we can partially rely on the experience of the structural synthesis. In particular, if one does not use metavariables, the performance is as good as for the structural synthesis, and programs including thousands of steps can be synthesized in a reasonable time, see (Tyugu, Matskin, and Penjam 1999). When metavariables are used, one has to restrict the search space by preselection of candidate components, and here we intend to use methods developed in (Penix and Alexander 1997) as we have noted it above. The logic of specifications can be extended without significant changes of the specification language. However, a more expressive logic may create more difficulties with the performance of proof search.

Implementation of the developed language is still continuing. In particular, one can solve only linear equations at present, although experience with the NUT system has given us sufficient know-how for implementing more elaborate equation solvers. Also work is going on with the aim of adding a visual interface (a scheme editor) for supporting visual development of metainterfaces.

Our practical application experience is still limited, but we have experimentally synthesized software for a context aware printing service, file downloading and uploading, composing streams, and for a bank transaction services (Lämmerman and Tyugu 2001). One part of the latter, debiting an account at a bank, has been presented in this paper.

Acknowledgements

The Swedish Foundation for Strategic Research (Stiftelsen för Strategisk Forskning, SSF) provided support for this work within the project Personal Computing and Communication (PCC) under contract PCC-0101-06. Discussions with researchers involved in this project are gratefully acknowledged.

References

Blaine, L., Gilham, L., Liu, J., Smith, D. R., and Westfold, S. 1998. Planware - Domain-Specific Synthesis of High-Performance Schedulers. In Proceedings of the Thirteenth Automated Software Engineering Conference, 270-280. IEEE Computer Society Press.

Lämmermann, S. 2000. Automated Composition of Java Software. Lic. diss., TRITA-IT AVH 00:03, ISSN 1403-5286. Dept. of Teleinformatics, KTH, Sweden.

Manna, Z., Waldinger, R. 1992. Fundamentals of Deductive Program Synthesis. TSE 18(8): 674-704.

Mints, G. and Tyugu, E. 1982. Justification of structural synthesis of programs. In Science of Computer Programming 2(3):215-240.

Mints, G. 1991. Propositional Logic Programming. In J. Hayes, D. Michie et al (eds.), Machine Intelligence 12:17-37. Clarendon Press.

Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I. 1994. Deductive Composition of Astronomical Software from Subroutine Libraries. In Automated Deduction. A. Bundy, ed., LNCS 814. Springer.

Penix, J. and Alexander, P. 1997. Toward Automated Component Adaption. In Proceedings of the 9th International Conference on Software Engineering & Knowledge Engineering (SEKE-97), Madrid, Spain.

Tyugu, E., Matskin, M., Penjam, J. 1999. Applications of structural synthesis of programs. In J. Wing, J. Woodcock, J. Davies (Eds.) FM`99. World Congress on Formal Methods in the Development of Computing Systems, vol. I, LNCS 1708: 551-569. Toulouse, France. Springer.

Tyugu, E. 1996. Classes as program specifications in NUT. *Journal of Automated Software Engineering* 1:315-334.

Premkumar, T., Devanbu, and Jones, M. A. 1994. The use of description logics in KBSE systems. In Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy.

Lämmerman, S., Tyugu, E. 2001. A Specification Logic for Dynamic Composition of Services. In Proceedings of the 21st IEEE International Conference on Distributed Computing Systems Workshops, 157-162. Mesa, Arizona. IEEE Computer Society Press.