

# A Composition Language with Precise Logical Semantics

*S. Lämmermann<sup>1</sup>, E. Tyugu<sup>2</sup>*

<sup>1</sup> KTH/IMIT, Electrum 229  
S-164 40 KISTA, Sweden, laemmi@it.kth.se

<sup>2</sup> Institute of Cybernetics,  
Akadeemia tee 21, 12618 Tallinn, Estonia  
tyugu@ieee.org

**Abstract.** A composition language that supports object-oriented software development and has precise translation into logic is presented. This translation is used in automatic synthesis of scripts needed for composing software from given specifications. The language is intended for representing logical specifications of classes in the form of metainterfaces. Its coverage of required properties of compositional languages is discussed and hints to an implementation of the language in Java environment are given.

## 1 Introduction

In the present paper we consider how logic can support compositional programming. We are going to do this on an example of a composition language with precise logical semantics that we have developed and implemented. We are going to share here our experience with it as well as compare its properties to the requirements on composition languages. Required properties of composition languages have been discussed in several papers, in particular, in [7] from the following sides: a) architectural description, enabling one to specify and reason about component architectures, b) scripting, enabling one to specify applications as configurations of components according to a given architectural style, c) glue, enabling also component adaptation, and d) coordination, needed to specify communication coordination mechanisms and execution policies for concurrent and distributed components. In [1] the discussion is on a rather technological level: connection, aggregation, code reuse, recursive construction, platform independence, and minimal need for new tools, framework support, simple learning, extensibility and versioning support. After introducing a composition language and its translation into logic, we shall discuss how it covers the features required for specifying compositions.

Having had some experience with generative programming (program synthesis) we decided to apply a logic-based program synthesis method in the context of object-oriented compositional programming. This puts some restrictions on a composition language. From one side, it must be possible to give a precise logical semantics of the language. From the other side, the language may be not expressive enough for reasoning about architectural properties of systems other than just their structure. On the technological level there is no language support for versioning, platform

independence, different frameworks. These features can be to some extent supported by the Java platform chosen for the implementation. However, most of properties: component selection, scripting, gluing, coordination, aggregation, extensibility, adaptability and hierarchical composition are supported reasonably well.

In our setting, components are *metaclasses* – classes supplied with metainterfaces. The latter are specifications written in the composition language. Logically, the main part of the component is its metainterface. A class of a component is just an implementation of functionality specified in the metainterface. In particular, a metaclass may consist of a metainterface only (having an empty class), then the whole implementation should be synthesized automatically from a metainterface. This becomes possible when a metainterface refers to other metaclasses with nonempty implementations (classes).

Figure 1 shows a general scheme of software composition from metaclasses. Metainterfaces can be considered as wrappers that 1) provide flexibility to classes and 2) contain information about their usability. One metainterface (an oval shape in Fig.1,a) can wrap one class (a rectangular shape) and several metaclasses that can be bound with each other. This supports structural description and inheritance hierarchy of components. A new class is composed from its specification in the form of a metainterface and a goal, Fig.1,b.

The process of generation of a new class is as follows. One writes a specification in the form of a metainterface using existing metaclasses, and also writes a top-level goal as an implication  $P \rightarrow R$  where  $P$  is precondition and  $R$  is postcondition of the main method of the new class to be synthesized. (We use an arrow  $\rightarrow$  for denoting implications in the specification language just for convenience of typing.) The goal is to find a realization for the implication  $P \rightarrow R$  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  $P \rightarrow R$  using logical formulae of specification as specific axioms, and extract the realization of the goal (a program) from its derivation. 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. The synthesis method used is ESSP – Extended Structural Synthesis of Programs [4]. This method is an extension of structural synthesis of programs implemented in several older systems [5].

## 2 Metainterfaces

Here we define a metainterface and give an example of the specification language. 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.

## A Composition Language with Precise Logical Semantics



Figure 1. Classes, metaclasses and metainterfaces

*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. In particular, interface variables denote in preconditions that the variable must have a value before the method is applied, and in postcondition -- that its value will be computed. (There are other kinds of preconditions that we shall discuss later.) For instance, having some class written in Java with two methods `getExcRate` 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 calculate the exchange rate. Here is a fragment of the metainterface:

```
// These are interface variables:
currency, localCurrency : String
exchangeRate : float
// Here follow two axioms with their realizations
//given in curly brackets:
currency, localCurrency -> exchangeRate {getExcRate}
-> 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 second axiom by

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

Usage of the metavariable (`LocalCurrency`) forces the synthesizer to search automatically for local currency that may depend on the location where the service

will be used. It can be detected (by unifying metavariables of different metainterfaces), if it is specified in some accessible metainterface. 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 {getExcRate}
     -> localCurrency {getLocalCurrency}
     -> (Exchange) {Exchange}";
  float getExcRate(String currency, String localCurrency)
    {...};
  String getLocalCurrency()
    {...};
}
```

A metainterface can be used for specifying how an application should be composed from metaclasses whose interface variables are bound by equalities. In such a case, a new class can be built completely from a specification of its metainterface. Here we present a metaclass for calculating exchange rates that also uses another metaclass as a component (pay attention to the bindings for currency and exchangeRate).

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

The keyword spec of the specification currency->exchangeRate{spec} denotes an unimplemented method that must be synthesized, if the implication currency -> exchangeRate will be used in a proof. 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.

```
// 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();
}
```

## A Composition Language with Precise Logical Semantics

```
float ExchangeRate_exchangeRate = 0;

ExchangeRate_exchangeRate =
    ExchangeRate_exch_obj.getExchangeRate
    (currency, ExchangeRate_exch_localCurrency);
return(ExchangeRate_exchangeRate);
}
```

### 3 Composition Language

The composition language is syntactically very simple. There are three types of statements in it:

- 1) specification of an interface variable  
*id* : *type* [*binding*, ...]
- 2) axiom  
*precondition* -> *postcondition*{*implementation*}
- 3) equation  
*exp1* = *exp2*

The *type* can be either a primitive type, a class, or a metaclass. Bindings (that may be missing) in a specification of an interface variable are equalities  $a = b$  where  $a$  is an interface variable of the metaclass given by *type* and  $b$  is any interface variable. Let us note that, on the topmost level, interface variables represent components, and bindings bind the components with each other. On lower levels, interface variables are used as parameters of components and single assignment variables that describe dataflow. This dual role of the interface variables is a consequence of the usage of specification language both for specifying components and for specifying composition of components.

Axioms are written in a logical language which will depend on the power of the prover used in the synthesizer. We can handle axioms with preconditions which are conjunctions of the following: propositional variables, implications of conjunctions of propositional variables and unary predicates called metavariables. The postconditions are disjunctions of conjunctions of propositional variables and unary predicates. *Implementation* is a name of a method of the class whose metainterface is being specified, or it is a keyword *spec* telling that the axiom states a new subgoal. In the latter case the axiom can have only conjunctions of propositional variables as pre- and postconditions.

Equations are given by arithmetic expressions:  $exp1 = exp2$ . We have found equations especially useful for gluing components together and adjusting their data representation (units etc.). Which expressions are acceptable in equations depends on the equation solver that has to be a part of the supporting software.

Let us note that interface variables with types given by metaclasses have the role of components that may be used in the generated software, but a component will be used only if needed. Here is a simple example in our specification language:

```

x1: Oscillator1
x2: Oscillator2
process: RungeKutta modell1 = x1, model2 = x2, from = 0,
to = 115
goal: -> process.graph

```

Either `Oscillator1` or `Oscillator2` may be used in actual computations, depending on the presence of resources for them detected at the synthesis time.

## 4 Logical Semantics

Complexity of the language lies in the logic of specifications. Every statement of a specification is translated into a set of logical formulae. All these formulae together constitute specific axioms of a theory in which the goal has to be proved. The three kinds of statements of the specification language are translated into the logic as follows.

The statement

$x : type$

where *type* denotes a primitive type, is translated into the empty set of formulae.

Let a metainterface of a class  $Q$  already have a semantics given by a set of formulae  $F_Q$ . Then the translation of a statement

$x : Q$

is the set of formulae obtained from  $F_Q$  by prefixing all propositional variables occurring in formulae with the prefix “ $x$ .” For example, if `Oscillator1` has a formula  $f.c \supset z.im$  in its semantics, then the new formula will be  $x.f.c \supset x.z$ . This translation “flattens” the specification – all axioms are on the same level, but preserves its structure in compound names. Actually, this translation is instantiation and unfolding of the metainterface of  $Q$ . Binding  $a = b$  of a statement  $x : Q \ a = b$  is translated into two axioms  $x.a \supset b$  and  $b \supset x.a$  denoting that  $b$  is computable from  $x.a$  and  $x.a$  is computable from  $b$ .

Translation of axioms is straightforward. Interface variables become propositional variables (denoting computability of respective interface variables), commas are translated as conjunctions, and metavariables as quantified unary predicates. More precisely, a metavariable ( $Q$ ), where  $Q$  must be a metaclass name, will become  $\exists u Q(u)$  where  $u$  is a new variable, and the predicate  $Q(u)$  means that  $u$  is an instance of the class  $Q$ .

An equation  $exp1 = exp2$  is translated into a collection of axioms showing which variables are computable from other variables of the equation, e.g.  $u = i * r$  is translated into three formulae:

$i \wedge r \supset u$

$u \wedge r \supset i$

$u \wedge i \supset r$ .

We have introduced semantics of structures as follows. A specification of a structure  $x$  with components  $x_1, \dots, x_n$  is translated into the axioms  $x \supset x_1 \wedge \dots \wedge x_n, x_1 \wedge \dots \wedge x_n \supset x$ . In particular, if we consider a metaclass  $Q$  as a structure constituted of its

## A Composition Language with Precise Logical Semantics

instance variables, then these axioms must be added to the semantics of the statement  $x : Q$ .

Let us discuss now how various software features are expressed in our logic. (This, together with the goal of keeping the logic as simple as possible for efficiency of proof search, is a justification of the logic we are using.) These features are the following:

- Dataflow between program modules
- Order of application of modules
- Availability of resources
- Hierarchical data and component structures
- Usage of pre-programmed control structures
- Alternative outcome of modules, in particular exception handling
- Selection and implicit linking of components.

1. *Dataflow.* Dataflow dependencies between pre-programmed functions can be expressed simply by implications, where for each functional module (e.g. a method of a class) 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 hand 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  $A$ . (We use capital letters for propositional variables here in order to avoid ambiguity.)

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, the 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. *Availability of resources* is expressed by propositions in the same way as the computability of variables. One just has to introduce a proposition for a resource stating that the resource is available.

4. *Hierarchical data structures* can be specified by means of structural relations that bind a structure, e.g.  $x$  with its components, say  $x_1, \dots, x_k$ . The computational possibilities of this relation are expressible by the implications  $X_1 \wedge \dots \wedge X_k \supset X$  and

$X \supset X_i$ , for  $i = 1, \dots, k$  where  $X$  and  $X_i$  denote 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.)

5. *Usage of predefined control structures.* As we have noted already, some building blocks that have to be specified may be 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 [6].

6. *Alternative outcome of modules.* 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.

7. *Implicit selections and linking of software components.* Our intention is to be able to put together large programs from components. We mean by a component not just a function, but also a software configuration flexibly providing useful functionality. We expect to have libraries of components supporting various problem domains, like packages in Java. So the problem of automatic selection of 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 restrict the usage of quantifiers so that we will not need the whole power of first order logic. We use only subformulae of the form  $\exists u Q(u)$  in axioms to denote a fact that an instance  $u$  of a class  $Q$  is available (can be created).

The limiting factor in applying the specification language is the efficiency of proof search needed for synthesizing a program. Proof search for non-nested implications has linear time complexity, and can be applied for an unfolded specification with tens of thousands of interface variables. Adding nested implications gives us the general case of intuitionistic propositional logic, hence high time complexity in general; however, we can use a modal logic for nested implications and get simpler proof search, see [6]. In the case of first-order logic we use quantifiers in a very restricted way, and hope to keep the search manageable as well. Indeed, here we have to use only a very simple unification.

## 5 Concluding Remarks

From the presentation above should be obvious that we have been interested not only in a scripting and component description language. Our goal has been to develop a language that gives sufficient information for automatic synthesis of scripts as well as for automatic search of components. These activities require good algorithms. It has been a common belief that automatic synthesis of algorithms (including synthesis of scripts) is not feasible due to its logical complexity. Experience with structural synthesis of programs has shown that in problem domains like simulation and engineering calculations this is not true [8]. Quite successful application of this synthesis method is also in simulation of complex hydraulic systems [2], [3]. Now we have extended the application domain to dynamic synthesis of services [4] – an application that is becoming a hot topic of research.

Having based our composition technique on object-oriented paradigm, and having taken Java environment as a platform, we should compare our components with Java beans. These components are very successfully applied in development of graphic user interfaces, and less successfully in other areas, although at their introduction time the expectations were for wider usage of beans. The reason is obviously in weak support for scripting. Another kind of components is enterprise Java beans (EJB) - better scripting support is needed also here.

We are quite satisfied with the design decision of using one and the same language for specifying components and specifying compositions. Partially this decision has been justified by usage of common logic for both purposes, but it works well in general and supports hierarchical composition and extensibility.

We would like to point out that components can be any Java classes supplied with metainterfaces in our implementation. Another possibility had been to restrict the design of classes. Obviously it would have been much simpler to implement a software generator for static Java methods and class variables and restricted exception handling. Although we do not specify events in logic, and synchronization is the responsibility of class developers (completely in Java style), the logic enables one to specify a connection between an event source and an event handler – the registration of the event handler. Finally, we would like to point out that our language is not dependent on any specific features of Java, and could be used on other object-oriented platforms.

## References

- [1] Birngruber, D.: CoML: Yet Another, But Simple Component Composition Language. In: Jean-Guy Schneider and Markus Lumpe (eds.): Proc. Workshop on Composition Languages WCL 2001. Vienna, September 11 (2001) 15-24
- [2] Grossschmidt, G., Harf, M.: Modelling and simulation of hydraulic systems in NUT programming environment. In: Viertes Deutch-Polnisches Seminar on Innovation und Fortschritt in der Fluidtechnik. Sopot (2001) 329-348.

- [3] Grossschmidt, G., Harf, M.: Simulation of an electro-hydraulic servo-valve in NUT programming environment. In: Proc. of the 13th European Simulation Symposium "Simulation in Industry" ESS'2001, October 18-20, Marseille (2001) 229-233
- [4] Lämmermann, S.: Runtime Service Composition. Ph.D. Diss. KTH, Stockholm, ISSN 1403-5286, ISRN KTH/IT/AVH, 02/03-SE (2002)
- [5] Mints, G., Tyugu, E.: Justification of structural synthesis of programs. Science of Computer Programming, 2(3) (1982) 215--240
- [6] Mints, G.: Propositional logic programming. In: Hayes, J. E. et al. (eds.): Towards an Automated Logic of Human Thought. Machine Intelligence, 12. Clarendon Press, Oxford (1991) 17-37.
- [7] Nierstrasz, O., Meijler, D.: Requirements for a Composition Language. ECOOP Workshop on Models and Languages for Coordination of Parallelism and Distribution, Springer Verlag, LNCS 924 (1995) 147-161
- [8] Tyugu, E., Matskin, M., Penjam, J.: Applications of structural synthesis of programs. In: Wing, J., Woodcock, J., Davies, J. (eds.): FM'99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, Vol. I, LNCS No. 1708, Springer (1999) 551 – 569.