A Case Study of Combining Compositional and Object-oriented Software Development

Enn Tyugu, Mait Harf and Pavel Grigorenko
Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia

Keywords: Domain-specific Modeling, Structural Synthesis of Programs, Model Driven Software Development, Compositional Software Design.

Abstract: We analyze an approach to software development where object-oriented and compositional software specifications are written in separate languages and are only loosely connected. It supports compositional design of software in a domain-specific language and automatic model-driven construction of code from classes written in Java. We justify our approach by giving examples of development of large simulation programs and services on large models. We present also an example of using our method in general purpose software development – this is bootstrapping the essential part of a software tool CoCoViLa, i.e. synthesizing CoCoViLa in CoCoViLa itself.

1 INTRODUCTION

Model-driven engineering (MDE) (Kent, 2002; Software, 2003), a.k.a. model-driven software engineering (MDSE) is gaining popularity in software development practice. This raises the level of abstraction of software specifications and simplifies the development of domain-oriented languages (DSL). MDSE tools use compositional approach to software specification. At first glance, it seems natural to use classes as components in MDE. However, in the object-oriented paradigm, classes are still programming concepts and are too closely related to implementation. They are rather inconvenient to use immediately as components. To overcome this inconvenience, one can define components differently from classes. An approach can be to introduce a specification language for specifying models and their components, separating the implementation aspects as much as possible from the domain specific software specifications. This has been done in several visual programming tools like MetaEdit+ (Tolvanen and Kelly, 2009) and CoCoViLa (Grigorenko et al., 2005).

The main result presented here are three case studies justifying the method where object-oriented and compositional specifications are combined in software development with minimal restrictions on both. Object-oriented programming is done in Java, but it could have been C++ or any other OO language. Compositional specification is written in a DSL developed with a tool briefly introduced first in (Grigorenko et al., 2005) and developed further during the following years. It is essential that OO and compositional specifications have separate name spaces, and can be developed almost independently. The presented approach is justified by three applications reviewed in the present paper. The first two use DSLs in simulation and service composition. Both solve large problems that validate the scalability of the approach. The third is an example of model-based development of a rather large software tool. More precisely, it is bootstrapping CoCoViLa in CoCoViLa itself, whose source code is available from the web[1].

This presentation is organized as follows. We discuss background and the related work in Section 2. Section 3 introduces basic concepts and the compositional specification language. The following three sections are case studies demonstrating scalability and applicability of the method. The presentation ends with a brief conclusion part.

2 BACKGROUND AND RELATED WORK

Model-driven approach to computing has a long history in specific domains like integrated circuit design and many simulation applications. The tools required

[1]http://www.cs.ioc.ee/cocovila

Tyugu E., Harf M. and Grigorenko P..
A Case Study of Combining Compositional and Object-oriented Software Development.
DOI: 10.5220/0004698502010208
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 201-208
ISBN: 978-989-758-007-9
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

for these applications are complicated and their development has been expensive. For example, todays network simulation tools like OPNET, OMNeT++ and ns-3 are large software systems and some of them cost hundreds of thousands of dollars. Experience shows that developing a new tool of this kind takes years.

A completely new trend was initiated in the nineties with introduction of UML – a universal modeling language that has become a standard for software specification, and has influenced also research in software engineering. One can say that UML has initiated model-driven software engineering (MDSE) (Brambilla et al., 2012). Considerable attempts have been made recently using UML-based metamodels and model transformations for automating software development (Santos et al., 2010). Generally speaking, metamodels and model transformation rules should be a background knowledge that support automation of program construction in various application domains.

Domain specific languages (DSL) have initially been developed independently of model-based approach to computing. Their history begins early in the middle of the last century with languages for numeric control of machine tools. Only recently, the research in DSLs and visual languages has adopted also model-based approach. The work most closely related to our research is performed by the MetaCase group which has developed MetaEdit+ (Tolvanen and Kelly, 2009). This work is based on domain-specific modeling (DSM). The technology prescribes analysis of a domain and development of a DSL first. This language enables one to specify models that are high-level specifications of systems and problems to be solved. The principal step is the development of a code generator specific to the domain. The book (Kelly and Tolvanen, 2008) presents lots of examples of both – DSL development and code generator development. Finally, a library of reusable code can be added and, as a result, a domain framework is created that speeds up the software development dramatically (up to 500% and 1000% according to (Kelly and Tolvanen, 2008)).

Our software technology is similar to DSM of the MetaCase group. However, we use deductive program synthesis for code generation. This makes the implementation of a new DSL much faster, because no generator development is needed.

The research in deductive program synthesis has a long history, beginning with the works of C. Green (Green, 1980), Z. Manna, R. Waldinger (Manna and Waldinger, 1993) and R. Constable (Constable, 1971). Our work is based on a tool that uses a special kind of the deductive synthesis – structural synthesis of programs (SSP) (Matskin and Tyugu, 2001). SSP is oriented at the efficiency. Its weak expressiveness must be compensated by a specification language that hides a large number of axioms needed in practical applications. As the logic of program synthesis is out of scope of our presentation, we will not discuss it here more.

Eclipse Modeling Project[2] provides tools for the model-based development. It includes Eclipse Modeling Framework (EMF), a modeling and code generation framework for specifying and managing (Ecore[3]) metamodels (written as XML files) and building applications from such models, Graphical Editing Framework, a facility for building interactive user interfaces, and Graphical Modeling Framework, bridging EMF and GEF by providing runtime infrastructures and generative components for building rich graphical editors. EMP does not provide automatic code generation, however there are complementing frameworks that provide such facilities via templates.

Microsoft has created a Visualization and Modeling SDK[4] (DSL Tools) for defining domain-specific languages, building visual designer environments and implementing code generators in Visual Studio. It uses UML-like notation for diagrams and stores concept definitions in XML format. Code generators are defined using template languages. The framework does not support n-ary relationships and its graphical capabilities are quite poor, only supporting variations of rectangles as shapes of objects.

3 COMBINING OBJECT-ORIENTED AND COMPOSITIONAL ARCHITECTURES

There are some principal difficulties in combining the two paradigms. Object-oriented approach widely uses encapsulation and hiding, and relies on parameter passing. Structural composition uses ports for binding components, it can provide considerable flexibility, and it is more fitted to represent ontology of a problem domain. We present CoCoViLa framework as an example of combining object-oriented and compositional architectures. The main design principles of CoCoViLa are the following:

- Two loosely connected specifications of software product are used – compositional (CA) and

[2]http://www.eclipse.org/modeling
[3]A variant of MOF http://www.omg.org/mof/
[4]http://code.msdn.microsoft.com/vsvmsdk

object-oriented (OO); the first specifies the logic and structure of software and the second specifies implementation of software components.

- OO and CA have separate namespaces.
- OO and CA have common type system.
- Inheritance in CA is consistent with inheritance in OO.
- Each software component has two parts: a Java class (this is an OO specification) and a compositional specification called metainterface. A component is called metaclass.
- Implementation of logical formulas that define functional dependencies between specification variables is described by methods of a corresponding Java class.

A metainterface has precise logical semantics given as a set of formulas – axioms with realizations given by methods specified in the OO part, and equations (as well as some other language constructs) written in the metainterface. CoCoViLa uses a constructive logic – intuitionistic propositional calculus for synthesizing a program from a model presented as a logical theory, therefore all axioms must have realizations. Components can be defined hierarchically, i.e. a metainterface of a component may contain components whose type is given by metaclasses, i.e. by other components. A metaclass may consist of a metainterface only, e.g. in a case when computations are specified by equations. Metainterface is written in a specification language. It is included as a comment in a Java class whose methods are implementations of the axioms of the metainterface.

Figure 1 shows three dimensions of software composition from metaclasses: compositional specification, object-oriented implementation, and common platform supporting both paradigms. On the compositional specification level (shown in the horizontal plane), instances of metaclasses (i.e. classes with metainterfaces) are bound by equalities between their components. One metainterface (an oval shape) can wrap one class (a rectangular shape) or several metaclass instances that can be bound with each other. Metainterfaces are wrappers that provide flexibility to classes and contain information about their usability. We see four metaclasses in Figure 1. Three of them have implementations as Java classes. The fourth includes their instances as components and has no own implementation. This supports the hierarchical description. In another dimension, conventional object-oriented inheritance and aggregation are supported as shown by arrows between the classes in Figure 1. A new class for performing computations must be synthesized from its specification given as a metainterface and a goal.

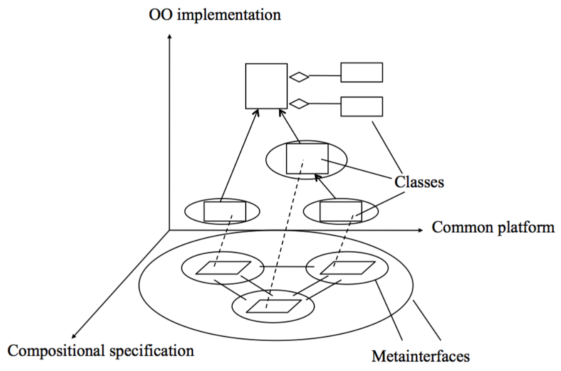


Figure 1: Classes, metaclasses and metainterfaces.

We have used several slightly different versions of a language for specifying metainterfaces in different versions of the system. A composition language has to include at least the following constructs.

1. Specifications of interface variables
   `type id, [id, ..]`
2. Bindings
   `var1 = var2`
3. Axioms
   `precondition -> postcondition{impl}`

Types in CoCoViLa can be primitive Java types (`int`, `double`, etc), Java classes or metaclasses. Bindings are used for structural composition, to specify the equality of interface variables `var1` and `var2` (these can be components of other interface variables declared in the specification).

Axioms are written in a logical language, the choice of which depends on the availability of a prover to be used in the synthesizer. In the existing version of CoCoViLa we use the logic of intuitionistic propositional calculus as described in (Matskin and Tyugu, 2001). Names of interface variables are used as propositional variables. Derivability of a propositional variable means computability of the respective interface variable. The preconditions of axioms can be conjunctions of propositional variables and implications of conjunctions of propositional variables. The postconditions are conjunctions of propositional variables. An implication in a precondition denotes a new goal – a new computational problem whose algorithm has to be synthesized before the method with this precondition can be applied.

Program synthesis plays an essential role in applying this logic. However, we only refer to the synthesis method SSP and do not explain it in any detail here. This is because the method, called also propositional logic programing, has been described in a number of papers already long ago (Matskin and Tyugu, 2001; Mints and Tyugu, 1990), and has been successfully used in software development, in particular, in PRIZ (Mints and Tyugu, 1990) and other similar systems.

The specification language includes also equations and more means for binding components, but these features can be modeled in the core of the language that is presented here. The following example is a metaclass of a complex number, where possible computations are described by equations and no Java methods are needed except the `sin` function imported from `java.util.Math` and used in an equation.

```
import java.util.Math;
class Complex {
  /*@ specification Complex {
    double re, im, arg, mod;
    mod^2 = re^2 + im^2;
    mod * sin(arg) = im;
  }@*/
}
```

4 MODEL-BASED DEVELOPMENT OF SIMULATION LANGUAGES

CoCoViLa includes a graphical interface that facilitates developing and using visual domain-oriented languages. This is especially suited for solving of simulation problems.

Here we present an example of modeling and simulation of complex fluid power systems (Grossschmidt and Harf, 2009). At the current state, the language includes about 100 problem-oriented concepts, each represented by a component. These concepts are either simple elements as hydraulic resistors, tubes, connecting hydraulic elements, or hydraulic subsystems like regulators, servo-drive with feedback, electro-hydraulic servo-valve. There are three levels of hierarchy of hydraulic components and subsystems.

Figure 2 shows a part of a model for simulation of dynamic behavior of a hydraulic servo-system. Each component has a menu button in the scrollable menu bar. The model includes 48 objects (only 16 objects are visible in the figure). However, the objects of this model are themselves specified by models, hence the total number of interface variables involved in the program synthesis is several thousands. The model is, in essence, a system of ordinary differential equations of order 28. Automatically synthesized Java code for solving this problem contains 8920 lines. The simulation gives behavior of the system under different disturbances. Results of some simulations are presented in Figure 3. The synthesis takes about one second and the simulation time on a PC for this example with a million iterations is three minutes. The graphs show movement of an actuator and a flapper under a step disturbance of the load force. One can see that this is a stiff dynamic system, hence a large number of simulation steps is needed for detecting oscillations of both low and high frequencies.

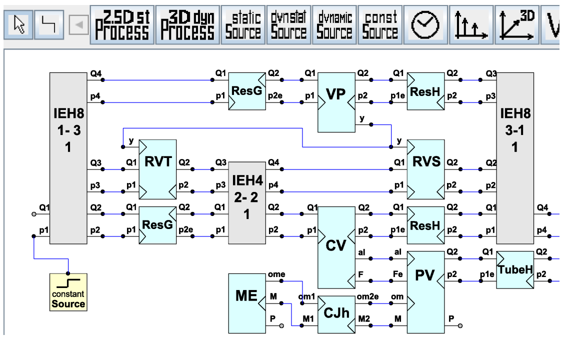
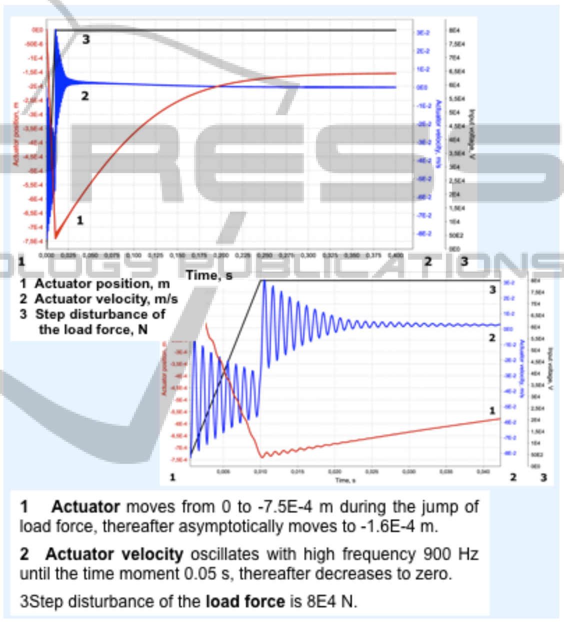


Figure 2: Part of a model of dynamic behavior of a hydraulic servo-system.



1 **Actuator** moves from 0 to -7.5E-4 m during the jump of load force, thereafter asymptotically moves to -1.6E-4 m.

2 **Actuator velocity** oscillates with high frequency 900 Hz until the time moment 0.05 s, thereafter decreases to zero.

3Step disturbance of the **load force** is 8E4 N.

Figure 3: Simulation results of step disturbance of load force.

5 COMPOSITION OF SERVICES ON LARGE SERVICE MODELS

As has been shown in (Matskin et al., 2007), the logic of CoCoViLa is well suited for describing web services and workflows. This inspired us to make experiments with synthesis of public services for Estonian authorities providing services over Internet. The first attempt was made by collecting all available services into a single metaclass in CoCoViLa that was called a service model (Maigre et al., 2009). This resulted in a large model containing more than 600 atomic services developed by different ministries. In order to achieve compatibility of services, a large number of data transformers were needed, e.g. transformation of date or address from one format into another. This model was used for synthesis of compound services including data from different authorities. We discovered that it was almost impossible to keep this single model up-to-date, because of the permanent changes done by different authorities on the large number of services. Currently the Estonian e-government information system integrates more than 2000 atomic services from about 100 organizations (Maigre and Tyugu, 2011).

A solution to the complexity problem was introducing hierarchy in the service model. Services of each ministry were based on its own database and were kept up-to-date and compatible by the ministry itself. Services of a ministry were collected into a metaclass that became a component of a general model (Grigorenko and Tyugu, 2012). Figure 4 shows a model that includes components for *Vehicles Registry*, *Business Registry*, *Population Registry*, *Migration Registry* and *Statutory Pension Insurance*. Only selected data were made public for each component, and the general service model became comprehensive. Figure 4 includes in the left upper corner a BPEL component. This is a superclass needed for control of the synthesis of a service in BPEL format. (It was possible also to create WSDL format by using a component WSDL as the superclass.) In CoCoViLa, schemes are also metaclasses. For every scheme it is possible to define a superclass that is a metaclass which will be able to manipulate scheme objects and pass the data between objects using special constructs. We present here in an abbreviated way an example of synthesis of a service taken from (Grigorenko and Tyugu, 2012). Figure 4 is just a specification of this problem. The goal is to find a notary's home address and number of his cars not older that 10 years from a given document number that was related to the notary. This goal is formalized as

```
DocumentNumber.data ->
    PopulationReg.address,
    Count.size
```

The synthesized algorithm is:

```
documentNumber->
  notaryNationalIDCode {toimiku_dokument};
```

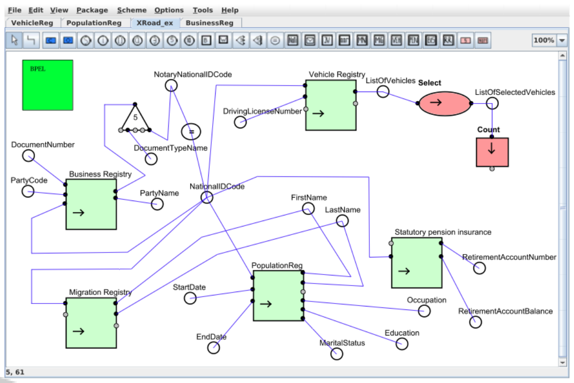


Figure 4: Hierarchical service model of the Estonian e-government.

```
(BusinessReg)
nationalIDCode->
  firstName, lastName {RR405IsikNimi};
(PopulationReg)
nationalIDCode, firstName, lastName ->
  address {RR57}; (PopulationReg)
nationalIDCode->
  listOfVehicles {paring22}; (VehicleReg)
listOfVehicles->
  listOfSelectedVehicles {select}; (Main)
listOfSelectedVehicles->
  size {count}; (Main)
```

The experiments showed that model driven approach could be used in synthesis of public services by experts responsible for creation of new services. Practical application of the method would require the application of some essential organizational measures.

6 BOOTSTRAPPING CoCoViLa

It is natural to use MDE in development and application of domain-specific languages, and there are numerous tools, especially simulation tools, that use this approach (Tolvanen and Kelly, 2009). Examples presented above demonstrate that CoCoViLa is not an exception in this sense. However, our goal is to expand the application domain of CoCoViLa for general-purpose software development. This requires a different software technology. In particular, when software components are developed for one project, one cannot expect that these components will be reused many times, because they may be needed only in this ongoing project, although the reuse is recommendable in the future projects. The effort for developing components should be minimal. Besides that, no restrictions can be accepted on using a programming language in implementation of components, because the technology must be generally applicable. We are in process

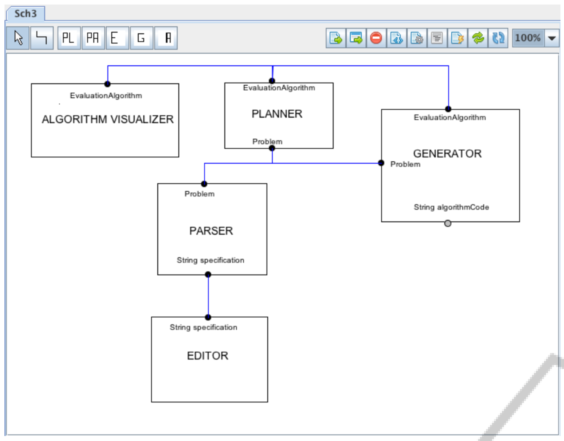


Figure 5: Model of CoCoViLa.

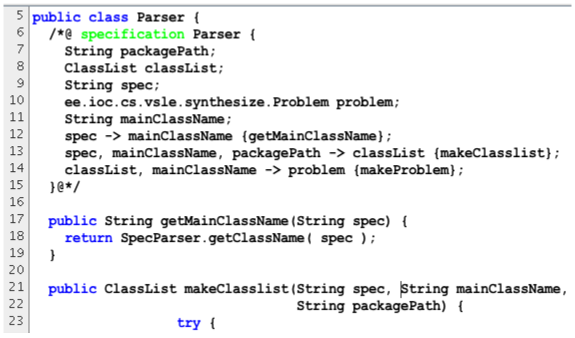
of fine-tuning and documenting the respective technology. As a test case for the technology we have chosen the development of a successor of CoCoViLa itself that we will call CCV.

When beginning this project, we had already source code of CoCoViLa, that consisted of 240 Java classes, totally about 30K lines of code. These classes had been developed in Java without any restrictions on programming. Our intention was to use these classes as much as possible for the new version of CoCoViLa.

We had no intention of changing the functionality of the system. So we had an essential part of the requirements specification of CCV in the form of documentation of the existing version of CoCoViLa. The first design step of the project was developing a model of CCV. This step was easy to perform, because we had already a conceptual description of the existing system which could be used for CCV as well. We decided to start with a high-level model of CCV, including only a small number of components with well-defined functionality. Information flow between the components was also made precise. This process showed some design faults in the existing system (hidden data paths, etc.). The high level components of both CoCoViLa and CCV were `Editor`, `Parser`, `Planner`, `Generator` and `AlgorithmVisualizer`.

The design step gave a model that was drawn in the window of the existing CoCoViLa scheme editor as shown in Figure 5. This scheme shows components, types of their inputs and outputs, and connections between the components. Developing graphics of components, and writing their metainterfaces was an easy task using the existing CoCoViLa class editor.

`Editor` is a large component responsible for interaction with a user, visual interface and transformation of graphical representation of the software model into a textual specification. It will be reasonable to



```
public class Parser {
  /*@ specification Parser {
    String packagePath;
    ClassList classList;
    String spec;
    ee.ioc.cs.vsle.synthesize.Problem problem;
    String mainClassName;
    spec -> mainClassName {getMainClassName};
    spec, mainClassName, packagePath -> classList {makeClasslist};
    classList, mainClassName -> problem {makeProblem};
  }@*/

  public String getMainClassName(String spec) {
    return SpecParser.getClassName( spec );
  }

  public ClassList makeClasslist(String spec, String mainClassName,
                                 String packagePath) {
                 try {
```

Figure 6: Metainterface of Parser.

develop a detailed model of this component in the future. The specification has a well defined type with syntax briefly introduced in Section 2, but for the implementation it is just a Java type `String`. Our technology prescribes showing both types in such a case, as can be seen from Figure 5.

`Parser` is a component that transforms the specification from a textual specification into an internal data structure specified by a Java class `Problem`. This data structure is mainly a large graph that is a representation of metainterfaces of all unfolded components. It includes also a goal – names of the data items from the specification that must be computed by the program. The `Problem` class has been designed for the needs of `Planner`. If planning algorithm changes, then this data structure may have to be changed as well.

`Planner` is the component that synthesizes the algorithm for computing the goal. It uses only `Problem` instance as the input. An algorithm of the planner has been developed gradually through the years, and its present version is described in (Grigorenko and Tyugu, 2012). Type of the output of `Planner` is described by Java class `EvaluationAlgorithm`. This output represents only a structure of the synthesized algorithm.

`Generator` transforms the `EvaluationAlgorithm` into Java code. It uses the instance of `Problem` to obtain and add input arguments into the generated program.

`AlgorithmVisualizer` is included in the model of CCV for supporting the debugging process, and it is not needed for the code development itself.

The next step of the design was to write metainterfaces of components. It was reasonable to do this already in Java, introducing empty methods with correct signatures in the metaclasses, if needed. Figure 6 shows a metainerface for `Parser`, and also one simple method `getMainClassName` for extracting a name of metaclass from a specification. The metainterface shows that problem can be constructed in three steps specified by the following axioms:

```
spec -> mainClassName{getClassName};
spec, mainClassName, packagePath ->
  classList{makeClassList};
classList,mainClassName->problem{makeProblem};
```

When designing metainterfaces, we had to decide how to use the existing code. We decided to follow the design of CoCoViLa as much as possible. In the case of `Parser`, this meant the usage of an intermediate data structure of type `ClassList`. This decision, in its turn, permitted to use the methods `makeClassList` and `makeProblem` in the implementation of the `Parser` metaclass without any changes.

As we can see from the example of `Parser`, the design of metainterfaces required some analysis of the CoCoViLa source. It included a private class `Synthesizer` that controlled the whole program construction process, and used other classes where needed. It was obvious that the existing classes could not serve immediately as implementations of components of the new model. An interesting question was, how much changes were needed. It came out that only minor changes of code were needed for implementation of methods after careful design of metainterfaces. This work was performed in two days by a programmer who had an acquaintance with the CoCoVila code.

7 CONCLUSIONS

We have shown here on three examples how a tool supporting the usage of Java classes supplied with logical specifications can be used in compositional programming. The first example concerns a practically important case of development and usage of a domain specific visual language. It demonstrates also the scalability of the method – a hierarchical specification that after unfolding has thousands of interface variables gives 8920 lines of synthesized code in one second. The second example is about automatic composition of services. Atomic services that are components have to be collected in large service models first, and the models are used then for specifying the services to be synthesized. From our point of view, the most important is the third example. This demonstrates the feasibility of using the tool in a software development project where domain specific language has only a secondary role. The main domain specific asset in this case is a system model that can be used as a specification for composing the software. This enables us to separate compositional specification from implementation, and is useful for organizing a software project.

ACKNOWLEDGEMENTS

This research has been partially supported by the European Regional Development Fund (ERDF) through Estonian Center of Excellence in Computer Science (EXCS), target-financed research theme No. 0140007s12 of the Estonian Ministry of Education and Research, and the project No. 3.2.1201.13-0026 Model-based Java software development technology.

REFERENCES

Brambilla, M., Cabot, J., and Wimmer, M. (2012). Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182.

Constable, R. L. (1971). Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland.

Green, C. (1980). *The application of theorem proving to question-answering systems*. Outstanding dissertations in the computer sciences. Garland Pub.

Grigorenko, P., Saabas, A., and Tyugu, E. (2005). Visual tool for generative programming. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 249–252. ACM.

Grigorenko, P. and Tyugu, E. (2012). Higher-order attribute semantics of flat declarative languages. *Computing and Informatics*, 29(2):251–280.

Grossschmidt, G. and Harf, M. (2009). Coco-sim–object-oriented multi-pole modelling and simulation environment for fluid power systems. part 2: Modelling and simulation of hydraulic-mechanical load-sensing system. *International Journal of Fluid Power*, 10(3):71–85.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-specific modeling: enabling full code generation*. Wiley. com.

Kent, S. (2002). Model driven engineering. In *Integrated formal methods*, pages 286–298. Springer.

Maigre, R., Küngas, P., Matskin, M., and Tyugu, E. (2009). Dynamic service synthesis on a large service models of a federated governmental information system. *International Journal On Advances in Intelligent Systems*, 2(1):181–191.

Maigre, R. and Tyugu, E. (2011). Composition of services on hierarchical service models. *Information Modelling and Knowledge Bases XXIII, Frontiers in Artificial Intelligence*, 237:110–129.

Manna, Z. and Waldinger, R. (1993). *The deductive foundations of computer programming: a one-volume version of the logical basis for computer programming*. Addison-Wesley Longman Publishing Co., Inc.

Matskin, M., Maigre, R., and Tyugu, E. (2007). Compositional logical semantics for business process languages. In *Internet and Web Applications and Services, 2007. ICIW'07. Second International Conference on*, pages 38–38. IEEE.

Matskin, M. and Tyugu, E. (2001). Strategies of structural synthesis of programs and its extensions. *Computers and Artificial Intelligence*, 20(1).

Mints, G. and Tyugu, E. (1990). Propositional logic programming and the priz system. *The Journal of Logic Programming*, 9(23):179 – 193.

Santos, A. L., Koskimies, K., and Lopes, A. (2010). Automating the construction of domain-specific modeling languages for object-oriented frameworks. *Journal of Systems and Software*, 83(7):1078–1093.

Software, I. (2003). *Special issue on model-driven development*, volume 20.

Tolvanen, J.-P. and Kelly, S. (2009). Metaedit+: defining and using integrated domain-specific modeling languages. In Arora, S. and Leavens, G. T., editors, *OOPSLA Companion*, pages 819–820. ACM.