Model-Based Technology of Software Development in Large

Jaan Penjam and Enn Tyugu

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
email: {jaan | tyugu}@cs.ioc.ee

Abstract. The present work describes a technology for developing software in unique and large projects. The present model-based technology supports the projects where a single software product is developed. This is different from the block languages and model-based software tools on the market, which provide a set of components where the reusability of the components is an important requirement. A distinguished feature of the technology is a support that it gives to the software design at an early stage of the design process. The design process begins on the architectural level where implementation details can be ignored. Components are introduced considering their functionality, but the implementability of a component is taken into account at the early stage of the design process only based on an experience of a designer.

1 Introduction

The present paper describes a technology for developing software in unique and large projects. Contrary to the model-based software tools on the market, which support the development of a set of components where the reusability of the components is an important requirement, the present tool and technology support the projects where a single software product is developed. The reusability is a beneficial, but not necessary property of the components designed and developed with this technology.

A distinguished feature of the technology is a support that it gives to the knowledge-based design of software at an early stage of the design process. One can say that *the design process begins on the architectural level where implementation details can be ignored.* Instead of classes, components are introduced. The implementability of a components can be taken into account only based on an experience of a designer. This design technology is intended to be analogous to the architectural design in other engineering areas like civil engineering or mechanical engineering.

Implementation of a component results in a class, but an implemented component has also a formal specification used in composition of the software system – the metainterface. Beside that, a component may support (local) protocols for communicating with other components. A component can be considered as a knowledge module, or even an agent, operating in a coordinated way with other components.

2 Visual description of software architecture

We present here a definition and a notation of knowledge module that can be used for describing software architecture on the knowledge level. A *knowledge module* is considered as a pair of sets: a set S of notations (objects) and a set M of denotations (meanings of notations) together with a notation-denotation relation between these sets. (This gives interpretation of the notations.) Also means to perform operations on the set S must be given, although we do not specify these means here, see details in [14]. They are specific to every knowledge module, and can be abstractly represented as inference rules. An abstract representation of a knowledge module is a deductive system with interpretation, see S. Maslov [11].
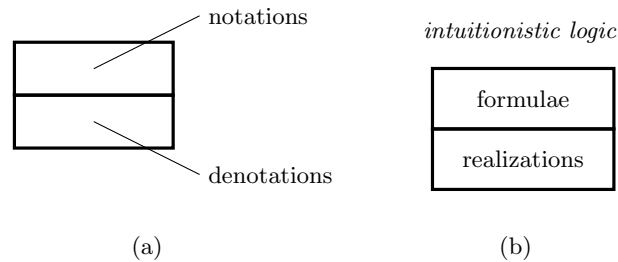

Fig. 1: Visual notation of a knowledge module (a) and of knowledge module of logic (b)

Visual representation of a knowledge module is a pair of rectangles as shown in Fig. 1a. An example of a meaningful knowledge module is given in Fig. 1b. It is a knowledge module of intuitionistic logic.

Knowldege modules can be bound in various ways: hierarchically, semantically and operationally [14]. Let us have two knowledge modules K_1, K_2 with sets of objects S_1, S_2, sets of meanings M_1, M_2 and notation-denotation relations R_1, R_2 respectively.

Hierarchical connection. We say that knowledge modules K_1 and K_2 are hierarchically connected, iff there is a relation R between the set of meanings M_1 and the set of objects S_2, and strongly hierarchically connected, iff there is a one-to-one mapping between the elements of a subset of M_1 and of a subset of S_2, see Fig. 2. A hierarchical connection of knowledge modules can be observed quite often in real life. An example is deductive program synthesis. The knowledge system of logic and the calculus of computable functions (CCF) are strongly hierarchically connected, because there exists a Curry-Howard isomorphism of proofs and formulae as types, see Fig. 2b.

Semantic connection. Knowledge modules that have one and the same set of meanings are *semantically connected*. This is the case, for instance, with classical
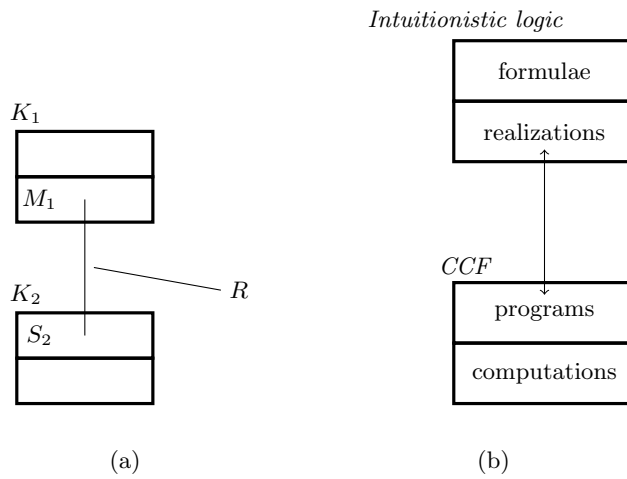
Fig. 2: Notation of hierarchical connection (a) and example of strongly hierarchically connected knowledge modules of deductive program synthesis (b)

logic systems that have different sets of inference rules, or even with natural languages that belong to closely related cultures (i.e. that have the same set of meanings). Graphical notation of semantic connection is shown in Fig. 3a.
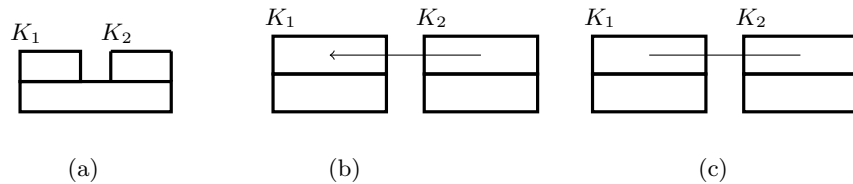

Fig. 3: Semantic connection (a), operational dependency (b) and operational connection (c)

Operational dependence and operational connection. Knowledge module K_1 is *operationally dependent* on a knowledge module K_2, if some of its derivation rules use K_2 in deriving a new object, i.e. the result of derivation in K_1 depends on knowledge processing in K_2, Fig. 3b.

Knowledge modules K_1, K_2 are *operationally connected*, if K_1 is operationally dependent on K_2, and K_2 is operationally dependent on K_1. Graphical notation of this connection is shown in Fig. 3c. The notations presented here are used for the architectural design of software at the first stage of a software project.

3 Architectural design of software

The first stage of design of a software system is its architectural design. At this stage, only the most general structure of the system is developed and specified by the knowledge architectural means. It is important to decide, which knowledge modules are needed, and how they will be connected. The input for this stage is a specification of functional requirements.
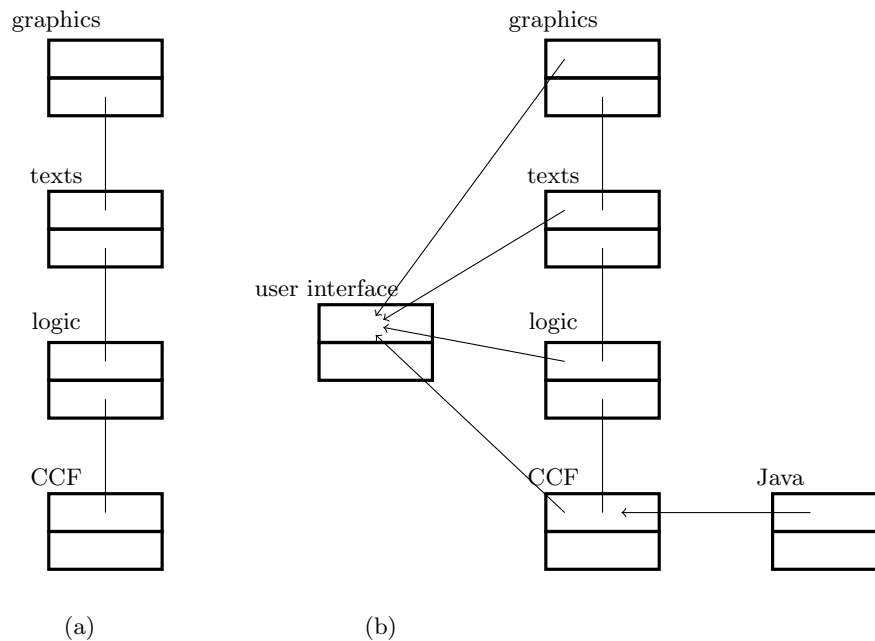

Fig. 4: Basic knowledge modules of CoCoViLa (a) and complete knowledge architecture of CoCOViLa (b).

We present our technology on an example of design and development of a model-based software tool CoCoSynth that includes also a program synthesis functionality. For a specification of functional requirements, we refer to [4] and the documentation of an existing tool CoCoViLa in web[1]. This means that we are applying our technology to the development of a new version of CoCoViLa.

It follows from the documentation of CoCoViLa that there must be two hierarchically connected knowledge modules *logic* and *CCF* that support the deductive program synthesis as it has been shown in Fig. 2b. This is the core of the software tool. We see also from the documentation that the input of the tool is not in a logical language, but in a textual domain-oriented specification

---

[1] http://cocovila.github.io/

language and/or in a visual language. This adds two more knowledge modules: *texts* and *graphics* to the architecture of the tool. These knowledge modules are hierarchically connected with the logical module, Fig. 4a. Operation of the tool is controlled by a user through a visual *user interface* that is also a knowledge module. We can describe this connection by operational dependence of this module on other modules. If we wish to show also the functionality that reflects the usage of Java at runtime, then we have to add a *Java* knowledge module tool as shown in Fig. 4b.

4   Design of components

This stage includes a conceptual analysis of the domain, and can be called also domain engineering, although the domain is presented by a single new software product in this case. Having an architectural description of software, one can take the knowledge modules of this description as components in the first place. However, these components may be too large or too small. The knowledge modules can be sometimes divided into smaller components, or collected into a single component, considering their functionality and realisability. A separate component may be required for representing a hierarchical relation between the knowledge modules. This will be demonstrated on the example below.

Table 1: Table of components

Notation	Input	Output	Comments
specification			a data structure
problem			a data structure
algorithm			a data structure
code			a data structure
Controller			will be a superclass
EDITOR		specification	
PARSER	specification	problem	
PLANNER	problem	algorithm	
GENERATOR	algorithm	code	
EXE	code		
ALGORITHM VISUALISER	algorithm		

In the present example, we keep the user interface knowledge module as a separate component *controller*. The knowledge modules *graphics* and *texts* will be joined into a single component *editor*, in order to facilitate their usage by *controller*, because the latter will produce the text in parallel with the graphics. We keep the knowledge module of logic as a separate component *planner*. This name reflects the purpose of the logic component that synthesises an algorithm of the software product. The hierarchical connection between *editor* and *planner*

will be represented by a separate component *parser* which transforms a text into logical formulae. The computational knowledge module CCF gives a component called *exe*. Also a hierarchical relation between *planner* and *exe* will be represented by a separate component generator which generates a Java program that has to be run. We introduce an additional component *algorithmVisualiser* for the visualisation of synthesised algorithms.

5   The CoCOViLa system overview

The tool used in our technology must support convenient implementation of components, preferably visual specification of software models and automatic code generation from a model. The tool CoCoViLa [4] used in our technology consists of two almost independent programs: *Component Editor* and *Specification Editor*. The first is a relatively small program for developing visual images of components, specifying their general properties and collecting them in domain-oriented packages.

The specification editor, referred further as CoCoViLa itself, uses a package of components for specifying tasks in respective domain or, in the present case, specifying a software model, and it supports code generation from the model. Essential working principles of this tool are the following:

- besides a visual representation, each software component has two parts: 1) logical specification of the component (LO), called metainterface, 2) object-oriented (OO) realisation of the component – a Java class;
- a component is called metaclass, because after program synthesis it may be transformed into several different classes;
- object-oriented and logical parts have separate namespaces, except for names of methods from OO used in LO – this enables one to write specifications of components almost independently of their Java realisations;
- OO and LO have a common type system.

Metainterface has a precise logical semantics given as a set of formulas – axioms with realisations given by methods of its Java class. These formulas constitute a theory in intiutuonistic logic that is used by structural synthesis of programs [12] for automatic construction of programs in CoCoViLa.

Components can be defined hierarchically, i.e. a metainterface of a component may contain components whose types are given by metaclasses, i.e. by other components. Also equations, as well as some other language constructs can be used in the metainterface. 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, and it is included as a comment in the Java class of the component between `/*@ ... @*/`. We give an example of a metaclass now. The metaclass `And` represents a logical element (and-gate) for signals represented by 0 and 1. It includes a Java method calc as a realisation of the axiom `in1, in2 -> out`.

```
01  public class And {
02    /*@
03        specification And {
04              int in1, in2, out;
05              in1, in2 -> out {calc};
06        }@*/
07        public int calc( int x, int y ) {
08          return Math.min(x, y);
09        }
10  }
```
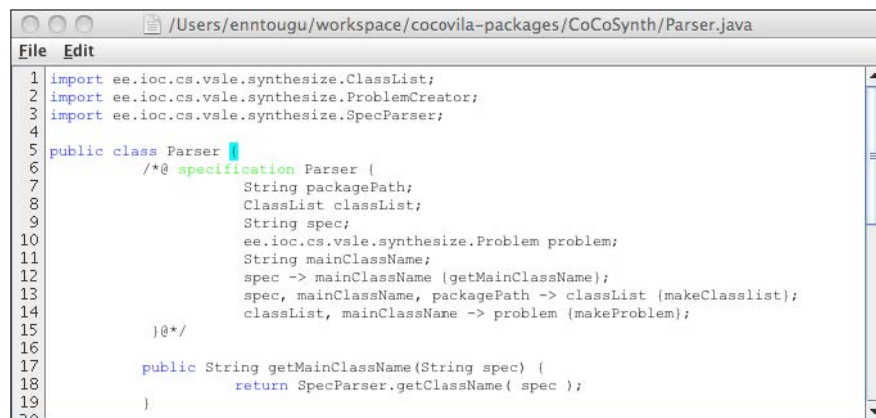
Lines 02 to 06 of the example are a metainterface. Line 04 is a variable specification of integer variables, and line 05 is an axiom. All Java classes and metaclasses can be used as type specifications. Lines 07 and 08 are in Java, and describe the method calc referred to in the axiom. An axiom is a formula of a conjunctive-implicative fragment of intuitionistic propositional logic, where commas represent conjunction symbols.

6 Implementation of components

Each implemented component has graphics, metainterface and Java class. It is reasonable to start with writing a metainterface, although the development of all three components can occur in parallel.

Writing metainterfaces. A metainterface is written in the specification language and included in the class of a component as a comment. Lines 6 to 15 of Fig. 5 show a metainterface for the component PARSER as an example. The metainterface shows that *problem* can be computed in three steps specified by the axioms in lines 12, 13 and 14.

Fig. 5: Metainterface of PARSER

Developing graphics. For developing graphics of a component one uses the *Class Editor* program of CoCoViLa that supports the development of visual representation of the component, definition of ports for binding components, as well as description of properties of component as it is described in the documentation of the Class Editor.

Implementing methods. Class of a component is created already when a metainterface is written. This class must be completed by writing all methods referred to in axioms of the metainterface. In the case of the component PARSER in our example, the methods are

> getMainClassName ,
> makeClassList ,
> makeProblem .

It is obvious that in these methods, other methods may be used that need to be implemented as well. This is a usual program development in Java. For instance, we see that he method `getClassName` of a class `SpecParser` is used in addition in the method `getMainClassName`.

As our example is in essence redeveloping CoCoViLa, it is reasonable to use its classes as much as possible for the implementation of methods. We have used the source code of CoCoViLa, that consists of 240 Java classes, totally about 30K lines of code. These classes were developed without any restrictions on programming in Java. Our experiment has shown that most classes could be used as is, or with only minor changes, depending on the developed metainterfaces.

Static model of software. When components are implemented, a high-level structural model of software can be written immediately in CoCoViLa specification language or drawn as a diagram. This is called static model. For the synthesis, the *static model* is automatically translated into a metainterface of a new Java class.

The static model is a specification for the tasks that the system has to perform. Programs for the tasks are synthesized automatically. When translated into logic, the static model describes a theory representing all possible computations on the model. Let us denote by G the set of the goals that describe the tasks solvable on the model. Each goal is written in the form

$$x_1, x_2, \ldots, x_m \rightarrow y_1, y_2, \ldots, y_m,$$

where $x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_m$ are variables of the static model, e.g. *specification*, *problem*, *algorithm* etc. in our example. These variables are considered in logic as propositional variables of the theory, and commas as conjunction symbols. Fig. 6 shows the static model where all components described in the previous section are present.

Dynamic model. The static model describes only tasks that the software can perform, but not a user interface to invoke these tasks. In order to describe the
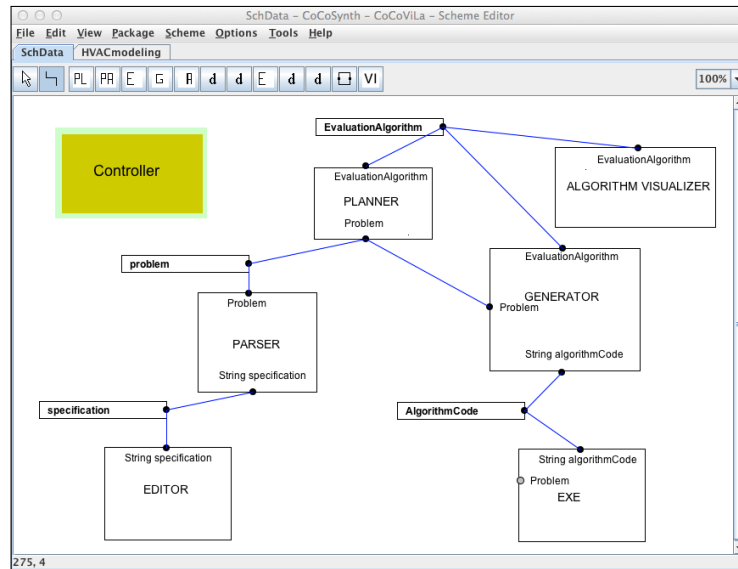
Fig. 6: Static model of CoCoSynth

interaction between a user and the system, a *dynamic model* of GUI is introduced. This model is specified as a statechart. This statechart may be explicitly given as a part of a requirements specification, or it may be implicitly described by other requirements. In the latter case, the development of the statechart is performed in a conventional way, e.g. as recommended by some UML-based technolgy.
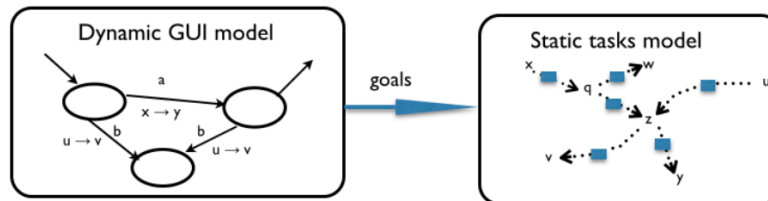

Fig. 7: Statcic and dynamic models are connected by goals specifying the tasks performed on the static model

Each transition t in the dynamic model is marked by an event $e(t)$ and a goal $g(t)$. The event is either a user action (e.g. pushing a button) or an event created by the system that triggers some transition. The goal describes a task to be performed on the static model when the transition t occurs. The tasks for all transitions must be solvable on the static model, i.e. for every task t must

be $g(t) \in G$. Fig. 7 shows abstractly a fragment of the dynamic model, and the connection between the static and dynamic models with the tasks $u \to v$, $x \to y$ and events denoted by a and b.

The dynamic model must be implemented as a component that becomes a superclass for the class of the static model. In our example, this is the component *controller*. Fig. 8 shows a part of the dynamic GUI model for our example with events created by user commands *Run*, *Compute goal*, *Compute all*, *Scheme*. One can find meaning of these commands from the documentation on CoCoViLa. The respective tasks for the commands are

$$c \to specification$$
$$c, specification \to algorithm$$
$$c, algorithm \to code$$
$$c, specification, goal \to algorithm$$
$$c, specification \to results$$
$$c, specification \to schemeMenuOpen,$$
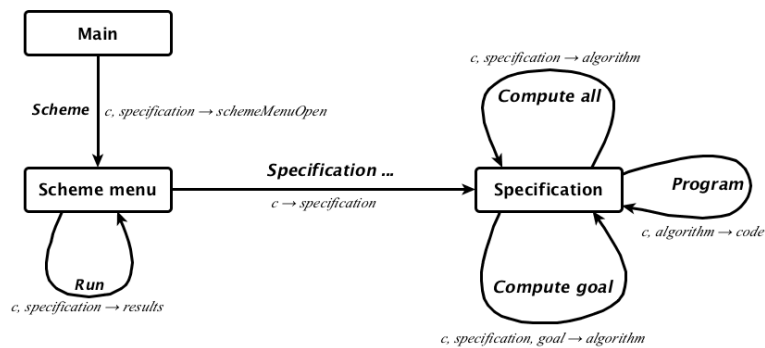
where c is a control and context variable.

Fig. 8: A part of the dynamic model of CoCoSynth

7 Implementing user interface

The user interface is implemented as a component as described above. Its functionality is described by the dynamic model that is in the form of a statechart. Java technology can be used in full in this implementation. However, implementation of a connection between of the dynamic and the static model by means of events and tasks deserves a special attention, and here are some hints for this.

1. The user interface component (*controller* in our case) can be implemented as a superclass for the static model. This enables one to use the names of variables of the static model for representing the tasks to be solved on it.
2. A task is always invoked by a respective event (see the dynamic model). The event is handled by an event handler in Java. Hence a call of program for a task must be included in the event handler.
3. As a program for a task is synthesised automatically using SSP, each task must be described by an implication in an axiom whose implementation creates event handlers. In our example, the event handlers are created by the method `initGUI`. Its axiom, included in the metainterface of *controller*, is as follows:

```
[c -> specificatin], [c,specification -> algorithm],
[c,algorithm -> code],[c,specification,goal -> algorithm],
[c,specification -> results],
[c,specification -> schemeMenuOpen],
c -> doneinitGUI{initGUI};
```

8 Synthesising the software

When the static model is implemented including the dynamic model as a superclass, a new program can be synthesised automatically by giving command *Run* from *Scheme* menu of the CoCoViLas main window. This means bootstrapping CoCoViLa in our CoCoSynt example.

The bootstrapping process and its results can be explained in Fig. 9. It shows the windows that open during the bootstrapping and running the bootstrapped tool. The two upper windows belong to CoCoViLa, they are a diagram of the static model of the new CoCoViLa (CoCoSynth) and the Java code of CoCoSynth synthesized in CoCoViLa. This completes the development of the new tool CoCoSynth.

The static model differs from the model in Fig. 6 by more compact presentation, where ports of the components are directly connected with each other without intermediate data components. Also an extra component *GUIactions* has been added to the model. It includes additional action listeners for the *controller*. When command *Run* is given from *Scheme* menu in this window, CoCoSynth is synthesized and started as well. The lower windows belong to the synthesized CoCoSynth. The leftmost window is the main window of CoCoSynth, it opens automatically after Run command given from CoCoViLa. It can be used for loading packages, drawing diagrams and performing computations. We see a package Gearbox loaded, and a diagram with several wheels, a motor and two visualizer components in it. After invoking *Specification...* command from *Scheme* menu of CoCoSynth, a new specification window opens according to the dynamic model.

This window can be used for textual editing of specification, for synthesizing application programs, for running these programs and for some other actions. After the command *ComputeAll* from the specification window, this window will show the synthesized Java program (partially visible in the second window
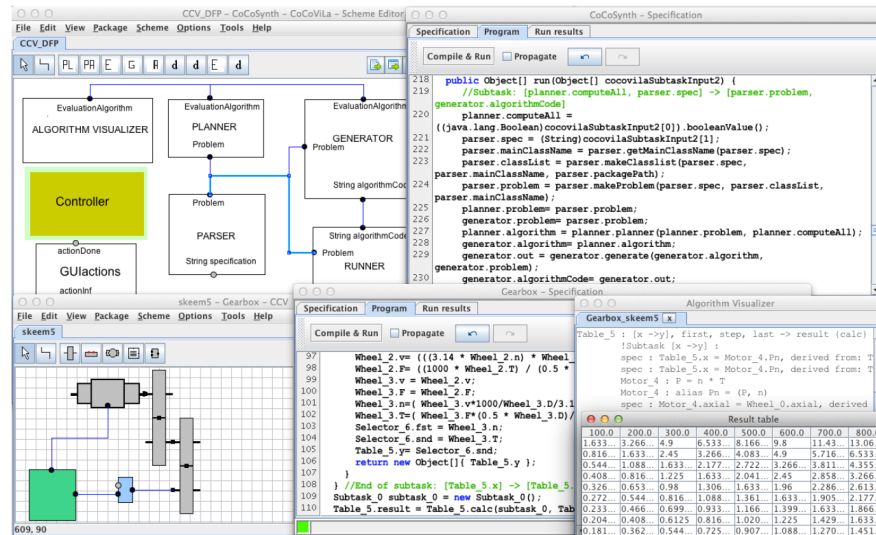
Fig. 9: Bootstrapping CoCoViLa

from left). After the command *Compile & Run* from this window, the synthesized program is compiled and executed and, in our example, a table of results is calculated and visualized (visible in the lower right window). Also a small part of algorithm for calculations on gearbox is visible from behind the result window. As the structural synthesis of programs used in CoCoViLa is fast, the whole process of bootstrapping and creating these windows takes less than 30 seconds (including interaction with a user, e.g. opening windows from GUI, etc) in the case when the diagrams are ready and loaded from some repository.

9 Related work and discussion

Model-based software development (MBSD) has been increasingly popular during several decades and a number of tools supporting implementation of model-driven principles have developed. The comprehensive study of achievements in the field can be found, for example, in the book [1]. Its most successful applications are in simulation software for automotive engineering, space technology and others, there are well known specialized products, mainly in simulation domain like Simulink [6] and more recently several systems like MetaEdit+[8], Modelica [3] etc.

UML[2] is de facto standard not to be ignored in model-based software development. Our technology uses two kinds of UML diagrams: statecharts and component diagrams. However, we have implemented the semantics of these diagrams in a way that guarantees automatic code generation from them for large

---

[2] www.uml.org

Java software including more than two hundred classes. This is a difference from the existing examples of code generation from UML models, e.g. [1], where only relatively small examples have been implemented.

Considerable amount of work is being done in improving the existing UML-based approaches with the aim of providing automated support to the software development[2] and language development [13]. One of the most successful approaches in this direction has been made by the Eclipse community. Eclipse Modelling Project (EMP) [5] that includes Eclipse Modelling Framework (EMF), Graphical Modelling Framework (GMF) and the Generative Modelling Tools (GMT) is a relatively new collection of technologies for building domain specific languages (DSLs). Generally speaking, EMP is a powerful set of tools, but it requires a lot of effort to develop a working DSL from scratch.

The system CoCoVILa used in this papaer belongs to the another research direction in the MBSD – Model-Integrated Computing (MIC) that addresses the problems of designing, creating, and evolving information systems by providing rich, domain-specific modelling environments including model analysis and model-based program synthesis tools [7]. CoCoViLa, integrates tools for specification and implementation of visual DSMLs (both for abstract and concrete syntax together with some well-formedness constraints) and translators that implement mappings from syntactic form of modes into formal theories in the semantic domain. The latter is a domain-independent framework for representing semantics (components and relations) of domain-specific models as sets of axioms in intuitionistic logic calculus equipped with specific inference rules (rules for structural synthesis of programs (SSP)) for generating of algorithms in a formal theory corresponding to the domain-specific model [12].

CoCoViLa has been applied mainly as a model-based simulation tool [10][3]. The novelty of this paper is to apply this approach for design and development of software product, including its static and dynamic aspects as well as user interface.

A technology with automatic code generation from models has been developed by Steven Kelly and Juha-Pekka Tolvanen. Their technology and tool MetaEdit+ have been well described in literature [9]. Our software technology is similar to that. However, we use deductive program synthesis for code generation. This makes the implementation of components much faster, because no generator development is needed.

Bootstrapping of software tools has been popular since early days of computing. It was helpful in developing compilers for new hardware, when there was no tool support on hardware. It is a non-trivial test of the language compiled. A list of languages having self-hosting compilres today includes 41 names [4]. From this perspective, bootstrapping of CoCoViLa can also be considered a good test of our technology.

---

[3] see also http://cocovila.github.io/

[4] https://en.wikipedia.org/wiki/Bootstrapping_(compilers)

10   Summary

The presented technology is summarised in Fig. 10. It shows the roles of different experts: domain expert, system engineer, graphic expert and code developer in a project developed in accordance with this technology. We see that after the requirements specification, developing the knowledge architecture and components specification, one can proceed by developing in parallel components metainterfaces and graphics as well as a dynamic model. Components code could be developed in parallel with graphics and metainterfaces as well, but one will need the dynamic model for writing the user interface component. The most important role has a system engineer, who performs six steps of the project. Also coordination of the project as a whole belongs to his role.
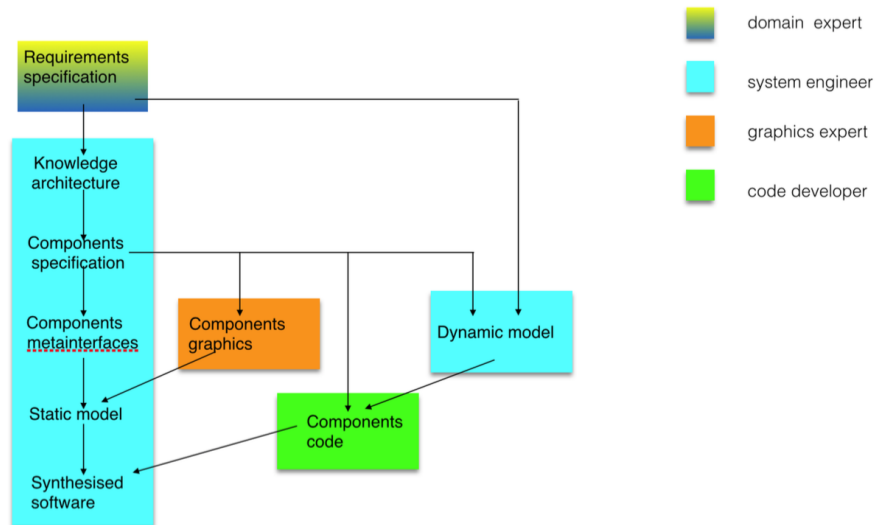

Fig. 10: Software technology step by step

Debugging and testing are not shown in Fig. 10. After developing the static model, its completeness can be tested on tasks prescribed by the dynamic model even before the code of components has been developed. After developing the code of user interface, one can test the interaction of dynamic and static model even without of other components codes. As usual, repetitions of some steps of the technology will be needed when errors are detected.

Acknowledgements

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers (2012), http://dx.doi.org/10.2200/S00441ED1V01Y201208SWE001

2. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML activities using dynamic meta modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4468, pp. 76–90. Springer (2007), http://dx.doi.org/10.1007/978-3-540-72952-5_5

3. Fritzson, P.: Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica. Wiley (2011), https://books.google.ee/books?id=413e_S4DI-IC

4. Grigorenko, P., Saabas, A., Tyugu, E.: Cocovila – compiler-compiler for visual languages. Electron. Notes Theor. Comput. Sci. 141(4), 137–142 (Dec 2005), http://dx.doi.org/10.1016/j.entcs.2005.05.009

5. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional, 1 edn. (2009)

6. Jain, S.: Modeling & Simulation Using MATLAB Simulink (With CD). Wiley India Pvt. Limited (2011), https://books.google.ee/books?id=qpv9ygAACAAJ

7. Karsai, G.: Lessons learned from building a graph transformation system. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday. Lecture Notes in Computer Science, vol. 5765, pp. 202–223. Springer (2010), http://dx.doi.org/10.1007/978-3-642-17322-6_10

8. Kelly, S., Lyytinen, K., Rossi, M., Tolvanen, J.: Metaedit+ at the age of 20. In: Jr., J.A.B., Krogstie, J., Pastor, O., Pernici, B., Rolland, C., Sølvberg, A. (eds.) Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE, pp. 131–137. Springer (2013), http://dx.doi.org/10.1007/978-3-642-36926-1_10

9. Kelly, S., Tolvanen, J.: Domain-Specific Modeling - Enabling Full Code Generation. Wiley (2008), http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html

10. Kotkas, V., Ojamaa, A., Grigorenko, P., Maigre, R., Harf, M., Tyugu, E.: Cocovila as a multifunctional simulation platform. In: Liu, J., Quaglia, F., Eidenbenz, S., Gilmore, S. (eds.) 4th International ICST Conference on Simulation Tools and Techniques, SIMUTools '11, Barcelona, Spain, March 22 - 24, 2011. pp. 198–205. ICST/ACM (2011), http://dx.doi.org/10.4108/icst.simutools.2011.245553

11. Maslov, S.Y.: Theory of Deductive Systems and Its Applications (Foundations of Computing). MIT Press (1987)

12. Mints, G., Tyugu, E.: Propositional logic programming and priz system. J. Log. Program. 9(2&3), 179–193 (1990), http://dx.doi.org/10.1016/0743-1066(90)90039-8

13. Selic, B.: A systematic approach to domain-specific language design using UML. In: Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece. pp. 2–9. IEEE Computer Society (2007), http://dx.doi.org/10.1109/ISORC.2007.10

14. Tyugu, E.: Understanding knowledge architectures. Knowledge-Based Systems 19(1), 50 – 56 (2006), http://www.sciencedirect.com/science/article/pii/S0950705105000936