CoCoViLa as a Multifunctional Simulation Platform

Vahur Kotkas Andres Ojamaa Pavel Grigorenko
Riina Maigre Mait Harf Enn Tyugu

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
{vahur, andres.ojamaa, pavelg, riina, mait, tyugu}@cs.ioc.ee

ABSTRACT

A flexible Java-based simulation platform that includes both continuous-time and discrete event simulation engines and is intended for applications in a variety of domains is presented. The platform supports visual and model-based software development and uses structural synthesis of programs for translating declarative specifications of simulation problems into executable code. Rich components are an important concept of the work. They are implemented as Java classes with additional specifications for program synthesis, and include visual representations as well as daemons supporting continuous interaction with the user during the simulation. The platform is developed as an open-source software, and its extensions can be written in Java and included into simulation packages.

1. INTRODUCTION

We expect that simulation problems are becoming not only computationally heavier, but also considerably more complex in the sense that a single problem may require orchestrated usage of numerous simulation engines, optimization programs as well as visualization and statistics software. A simulation engine and other useful tools are usually built into the simulation platform. It is a common practice that a simulation problem itself is described using a model-based technology, mainly by specifying a model of the simulated system. In the present work we propose to use the model-based approach for constructing and compiling a completely new program for every simulation, using simulation engines, optimizers, visualizers etc. as preprogrammed components. In this case a simulation platform is a model-based software development environment plus assets (components, language, models) for specifying simulation problems. This requires good automation of program construction from a given model.

We have implemented CoCoViLa [4] as a suitable software development environment. It has visual tools, but most importantly, it supports full automatic program construction from specifications that are given visually. The CoCoViLa itself has been developed bearing in mind programming of simulation problems. However, it has been used also in a more general model-based software development, e.g., for composing web services on large service models [11]. In this sense it is similar to visual software development tools like MetaEdit [17]. In our design and implementation, we have paid special attention at flexibility. This allows us to call the CoCoViLa platform multifunctional. In particular, it is applicable for very different specialized problems as almost every aspect (optimization algorithms, simulation engines, etc.) of the simulation can be customized by replacing a component with another from the toolbox or implementing a new component.

The present paper is structured as follows. Section 2 describes the conceptual design of the simulation platform. The specification language is described in Section 3. Section 4 presents implementation aspects and Section 5 contains application examples.

SIMUTOOLS 2011, March 21-25, Barcelona, Spain
Copyright © 2011 ICST 978-1-936968-00-8
DOI 10.4108/icst.simutools.2011.245553

2. DESIGN PRINCIPLES

Our main design decision is that we rely on a completely automatic program construction from a specification, and use a fast synthesis method that gives out a source code ready for compilation and execution. This program synthesis—structural synthesis of programs (SSP) [18] is not new, it has been implemented and used in several earlier software tools [12, 19]. It uses essentially dataflow for composing a program, like, for instance, Simulink [2] does. However, the usage of components in the form of higher-order functions that take synthesized parts of a program as inputs makes a significant difference over conventional dataflow techniques. The program synthesis process uses dataflow recursively for solving so called subtasks—generating inputs for the higher-order functions, i.e., for achieving the goals like “synthesize a body for the loop in this particular component”. This makes the method universally applicable – theoretically any algorithm can be synthesized in this way from a suitable specification and a fixed set of preprogrammed components [12]. A program is synthesized piecewise and the pieces are bound together by preprogrammed higher-order functions that realize required control structures. The planner is CoCoViLa’s core part responsible for synthesis, it is described in more detail in Section 4.2.

The second design decision is using full capabilities of model-based software development—developing a completes model for each simulation problem that includes not only description of a simulated system, but describes also the usage of simulation engines and other tools—visualizers, optimizers etc. needed for the particular problem. A model-based software development consists of two stages: domain engineering that provides assets for developing applications, and application engineering that uses the assets. Assets of our model-based approach are rich components [13]—Java classes extended with specifications for program synthesis and supplied also with visual representation like Java beans. A rich component may have another class associated with it—that specifies a daemon—a thread that runs permanently and supports user interaction during the problem description and simulation phases. A component with a daemon can behave as an agent and can be compared with agents in simulation environment, for instance, provided by AgentSheets [16].

A collection of rich components for a problem domain constitute a package that is an implementation of a domain specific language (DSL) for this particular domain. Naturally, packages can be restructured using export/import commands, and components of different packages can be used for specifying a simulation problem. Generally applicable components like simulation engines, optimizers, input-output components are collected in a package called toolbox.

We have chosen Java as the implementation environment of our simulation platform. This is justified by useful properties of Java: good portability and interoperability, support for distributed computing, open source ideology, dynamic compilation and loading of classes. Our platform has been developed in a way that does not restrict the usage of Java for programming of classes. And from the other side—all Java types and classes can be used as types of objects in specifications.

3. LANGUAGE

We have implemented one specification language for describing both components and simulation problems. This gives immediately a possibility of developing hierarchical descriptions—a part of a model of simulated system can easily be declared to be a component. This language is built on top of Java and merged with Java in a simple way: a specification is always a comment of a special form included in a Java class. It may happen that a class has nothing else than a specification in it. An important decision has been to keep separated the namespaces of a procedural part written in Java and of a specification inside a Java class. Here is an example of a specification that shows also the form of a specification, and the possibility of usage of equations in a specification:

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

The only connection between the actual Java code and the specification is through the method names that refer to implementations of functions in a specification. The following is an example of usage of a Java method (`getMaxVal`) as an implementation of a higher-order function in a specification:

```
class Max {
    /*@ specification Max {
        int arg, val, maxval;
        [arg->val]->maxval{getMaxVal};
    } @*/

    public int getMaxVal(Subtask sbt) {
        ...
        return maxval;
    }
}
```

Argument `sbt` of the method `getMaxVal` should get a synthesized method that implements the interface `Subtask`.

Selection of program synthesis puts restrictions on the specification language: each function usable in synthesis must have a specification describing its input and output conditions. We call these specifications *axioms*, as it has been the convention for structural synthesis of programs. These axioms have a precise logical meaning, but we are able to explain them in terms of dataflow only. The example above includes the axiom

```
    [arg->val]->maxval{getMaxVal};
```

This axiom has one input `[arg->val]` that is a subtask describing a function for computing `val` from given `arg` (this function has to be synthesized by the planner). The axiom has an output `maxval`.

The platform supports a textual and a visual representation of the specification language. The specification language has a simple and a rather conventional syntax of declarative compositional languages. It enables one to specify typed objects and bind them with each other by connecting their attributes by equalities. Numeric variables can be bound also by algebraic equations. Constant values can be assigned to variables of any type, as soon as the value has a textual representation. The textual specification enables one also to specify the usage of methods of the class where the specification is included by writing axioms about their applicability, i.e., by giving their pre- and postconditions. Extensibility of the language is achieved by introduction of new types. The following is core of the language:

1. declaration of a component:

   type id;

   This declaration specifies a component of a model with given type and name.

2. binding:

   $var1.portA \ = \ var2.portB$;

   This statement specifies an equality between variables (ports) of components.

3. valuation:

   $var1.portC \ = \ value$;

   This statement defines a functional dependency with no inputs and with one output that receives a constant value.

4. axiom

   $precondition \rightarrow postcondition\{implementation\}$;

   The precondition of axioms is a list of component names and subtasks. The postcondition is a component name.

The names in precondition show components that are inputs and the name in postcondition shows a component in output of the computation by the function given by the method with the name implementation. A subtask has the form $[x_1, \ldots, x_n \rightarrow y_1, \ldots, y_m]$ and defines a function with inputs and outputs given on the left and right side of the arrow. The function defined by a subtask has to be synthesized and given as an input to the function described by the axiom.

5. equation

   $AExpression \ = \ AExpression;$

   Equation defines one or more functional dependencies that are solving functions for variables bound by the equations. Arithmetic expressions are the Java ones, and can be solved only for the variables that have one occurrence in an expression.

6. tuple

   $alias \ id \ = \ (ListOfNames);$

   where $ListOfNames$ can include names with wildcards of the form $*.id$. In this case all ports of components of a specification that have the name id are included in $ListOfNames$. For example, `alias state = (*.state)` describes a state vector consisting of states of components.

The visual language describes schemes and, strictly speaking, uses only the first two kinds of statements. However, through pop-up windows one can add also valuations, aliases and equations to a scheme.

4. IMPLEMENTATION

4.1 The platform

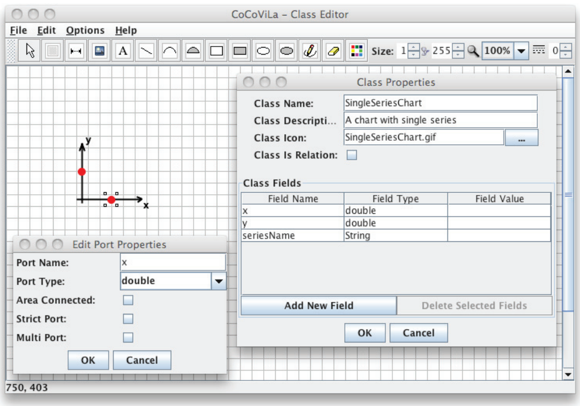
From a user's perspective, CoCoViLa consists of two visual editors: the Class Editor for creating simulation packages by developing domain-specific concepts and the Scheme Editor for the visual composition of simulation problems and their execution.

4.1.1 *Class Editor*

In the Class Editor users can define visual aspects of rich components using drawing capabilities or by importing corresponding bitmaps. Figure 1 shows the development of a component responsible for plotting charts in the window of the Class Editor. The image of the chart contains two ports (dots) for providing data to the axis. A smaller pop-up window is for defining properties of a highlighted port. Another window is for specifying attributes of the given component, e.g., class name, toolbar icon, description and a set of fields of visual interface with types and default values. Functional properties of this component are implemented in a Java class.

4.1.2 *Scheme Editor*

The Scheme Editor is a multi-purpose tool. It allows to load a simulation package created in the Class Editor (user interface with toolbars and menus is automatically generated from the package description) and to compose visually a simulation problem in a scheme. Schemes can be saved and used as components in other schemes higher in the hierarchy. This is due to the fact that each scheme is a Java



Class Name: SingleSeriesChart
Class Descripti...: A chart with single series
Class Icon: SingleSeriesChart.gif
Class Is Relation: ☐

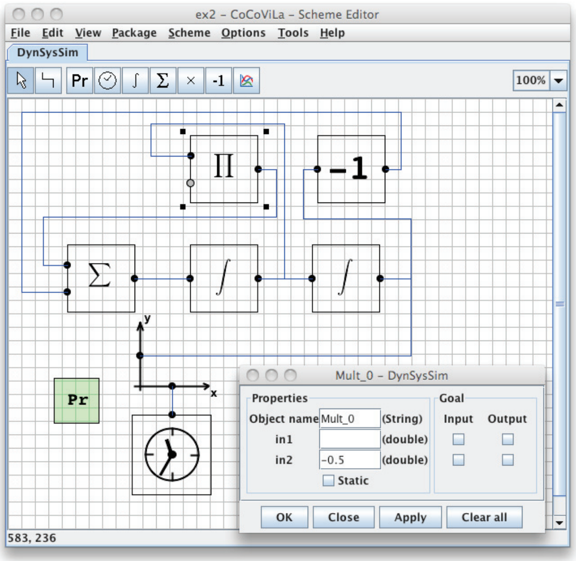
Class Fields

Field Name	Field Type	Field Value
x	double	
y	double	
seriesName	String	

Port Name: x
Port Type: double
Area Connected: ☐
Strict Port: ☐
Multi Port: ☐

Figure 1: The Class Editor window

class. From a visual description of a problem, a simulation program is synthesized automatically. There are also debugging capabilities (algorithm visualizer, viewer for synthesized code, etc.). An executed simulation problem can show results both in a separate window or display the feedback directly on a scheme.



Object name	Mult_0	(String)		
in1		(double)	☐	☐
in2	-0.5	(double)	☐	☐
	☐ Static			

Figure 2: The Scheme Editor window

Figure 2 shows the Scheme Editor with a loaded package for simulating simple dynamic systems. Such systems are described by ordinary differential equations. The scheme shown in the figure has been composed by connecting ports of components of the following types: `Integrator`, `Adder`, `Multiplier`, `Inverter`, `Clock` and `Chart`. A continuous-time simulation engine (`Pr`) is added as a superclass of the class `ex2` of the scheme. The toolbar at the top of the scheme is for adding objects and connections to the scheme. A pop-up window is for specifying attributes of a selected `Multiplier` object. The Scheme Editor is syntax directed and the correctness of the scheme is forced during editing.

4.2 Planner

The *planner* is a core part of our platform. Its purpose is to transform declarative specifications of simulation problems into executable programs. The planner determines computational paths from initial variables to required goal variables (i.e., tries to solve a given *computational problem* "find values of V from given values of U", where U and V are sets of input and output variables). The planner's task is not only to construct a linear dataflow, but also to solve subtasks (higher-order dataflow) and to perform optimization of an algorithm. Generation of a resulting program's code from an algorithm is then a straightforward process.

Let us have a small example of dataflow planning where a goal is to compute a next state of a simulation from a current state. That is, a computational problem can be expressed with a following statement:

```
state -> nextstate;           (1)
```

The problem can be solved only if a relation between `state` and `nextstate` is specified. Assume the state is just a numerical value and the next state is an increment of a current state by a step which is obtained using a method `getStep`:

```
nextstate = state + step;
-> step{getStep};
out = nextstate;
```

In this case the planner produces the following dataflow if a value of the state is given as input:

```
state = getStep();
nextstate = state + step;
```

Note that the calculation of a variable `out` is not included in the dataflow because it is not required for solving a given problem (1). We can add another statement into our specification that prints the value of the nextstate

```
nextstate -> printed{print};  (2)
```

The dataflow for (1) will not include (2) because (1) does not include a control variable `printed` in a set of outputs. Another computational problem has to be stated:

```
state -> printed;
```

The example above shows how to construct a linear dataflow, i.e., to compute the value of the next state once. Simulation tasks require to compute states in a loop until some satisfying final state is reached. To specify such task in CoCoViLa, subtasks have to be used. The following statement specifies that a final state can be computed from a given initial state if there exists a function that calculates the next state from a given state.

```
[state -> nextstate], istate -> fstate{proc};
```

To solve a topmost computational problem `istate->fstate`, the subtask `state -> nextstate` must be solved. Having (2), subtask is solvable and higher-order dataflow can be constructed by the planner. The synthesized function `state -> nextstate` is passed as an argument to the method `proc` and this method can iteratively call the function to increment the state as long as it is needed.

Following the described technique, simulation engines that require loops and other control structures are implemented using subtasks and the planner takes care of synthesizing bodies of subtasks. In addition, the planner can be invoked at runtime for generating new programs to solve tasks on models that might have been changed dynamically during the simulation.

In general, the synthesis of an algorithm with subtasks has exponential time complexity with respect to the number of subtasks in a specification, as the solvability of one subtask may depend on the solvability on another subtask or it can be the case that one and the same subtask has to be solved repeatedly in one and the same branch. The solvability search is done on an *and-or* tree where *and*-nodes are axioms and *or*-nodes are subtasks. An implemented algorithm is incremental depth-first search with backtracking and additional heuristics.

The planning algorithm can be explained also in terms of logic, and vice versa—theorems of intuitionistic propositional calculus can be encoded as sets of axioms (axioms can be considered as propositional formulas where arrows denote implications and commas denote conjunctions). To test planner's performance, CoCoViLa has been compared to several well-known theorem provers [6]. Full description and results of the benchmark can be found in [3].

To give some details, a formula $((((A \rightarrow B) \rightarrow A) \rightarrow A) \rightarrow B) \rightarrow B$ (an intuitionistic analog of classical Peirce's law) was elaborated to a special form where it was possible to include copies of parts of initial formula to create more levels of nestedness. The results are summarized in the Table 1.

	n-th level formula					
Tool	2	4	6	7	10	11
CoCoViLa	<0.01	0.05	36.24	1579.6	–	–
STRIP (ch)	<0.01	0.34	3781.3	–	–	–
STRIP (pr)	34.87	–	–	–	–	–
iLeanCoP	–	–	–	–	–	–
iLeanSeP	–	–	–	–	–	–
LJT	<0.01	0.05	35.15	1572.3	–	–
PITP	0.01	0.05	15.73	343.5	–	–
Gandalf	0.02	0.19	0.53	0.9	7.6	–

Table 1: Proof search time (in seconds) of various theorem provers

4.3 Rich components

Rich components are descriptions of domain-specific concepts that are used in simulation. Rich components collected into packages are the building blocks for computational problems and simulation tasks. Such components are designed to consist of four different parts (views).

The first part is the graphical representation. Having such representation we can build schemes of simulation tasks via visual composition of the components.

The graphical representation can be translated into a textual specification—the second part. For textual specification of components we use the composition language described in Section 3. The automatic composition of simulation programs is supported by the transformation of textual specifications into the logical representation used by attribute evaluation algorithms implemented in the simulation platform.

The third part of a rich component is a *metaclass* [4]—the Java class in our case that may contain methods that are the realizations of dependencies described in textual specification of a rich component. The textual specification included into the metaclass is called a *metainterface*. Metainterfaces appear in metaclasses as comments and are used only by the simulation platform to facilitate automatic composition of simulation programs.

The optional fourth part of a rich component is called a *daemon*. The daemon is a Java class that describes a thread that can be started by a user, if it is needed. Daemons enable one to develop flexible interfaces to simulation programs. In addition, daemons have the reference to the scheme being composed by the user. This gives them full control over the scheme enabling one to develop components that can assist the user in the task of composing a simulation model by displaying dialog windows and arbitrarily complex visual feedback or directly modifying the model. For example, a daemon can be used to enforce some complex syntax rules on the structure of the visual scheme.

Simulation packages may contain simulation engines implemented as rich components, in this case such components can be of considerable size. From the other side, rich components may be very small entities that are used intensively in an internal loop of a simulation program. In other words, depending on the implementation and the purpose of the application, rich components can be very lightweight containing only visual and metainterface part or heavyweight including all four functional parts.

4.4 Toolbox

Toolbox is a package in CoCoViLa that contains a number of components useful for building simulation programs. It includes several simulation engines and also visualization components. Another feature implemented as a standalone tool is an expert system environment (some simulation problems require handling of the expert knowledge).

4.4.1 Simulation engines

The toolbox includes basic simulation engines as components for driving discrete-event, continuous time and hybrid simulations. Simulation engines are used as superclasses of schemes, but they are not scheme specific. Being a superclass, a simulation engine is able to collect parts of a state from the underlying components automatically using aliases and wildcards.

An important feature of the CoCoViLa platform is the ability to generate parts of the simulation program automatically, even at runtime, if needed. This adds flexibility in developing and reusing simulation models, as, for example, components can be synthesized and/or initialized lazily including fragments of event handlers.

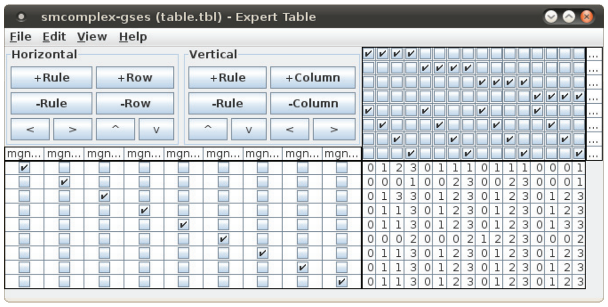
4.4.2 Visualization

A visualization package in CoCoViLa consists of several components for plotting diagrams, two- and three- dimensional charts and also components for data input (sliders, etc.). Users can take individual components, import them into their own simulation packages and customize according to their specific needs.

4.4.3 Expert system

The expert system in CoCoViLa is developed as a standalone tool which may contain a number of decision tables. A decision table is implemented as a set of production rules. Several decision tables can be used simultaneously in one scheme. The expert system has the following features:

- XML format for storing tables in files;
- forward chaining inference engine for acquiring data from tables;
- graphical user interface for creating and managing tables, see Fig. 3;
- API to support querying tables from specifications of components.



									✓	✓	✓	✓	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	...
									☐	☐	☐	☐	✓	✓	✓	✓	☐	☐	☐	☐	☐	☐	☐	☐	...
									☐	☐	☐	☐	☐	☐	☐	☐	✓	✓	✓	✓	☐	☐	☐	☐	...
									☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	✓	✓	✓	✓	...
									✓	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	...
									☐	✓	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	...
									☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	✓	☐	...
mgn...	mgn...	mgn...	mgn...	mgn...	mgn...	mgn...	mgn...	mgn...	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	✓	...
✓	☐	☐	☐	☐	☐	☐	☐	☐	0	1	2	3	0	1	1	1	0	1	1	1	0	0	0	1	
☐	✓	☐	☐	☐	☐	☐	☐	☐	0	0	0	1	0	0	2	3	0	0	2	3	0	0	0	1	
☐	☐	✓	☐	☐	☐	☐	☐	☐	0	1	3	3	0	1	2	3	0	1	2	3	0	1	2	3	
☐	☐	☐	✓	☐	☐	☐	☐	☐	0	1	1	3	0	1	2	3	0	1	2	3	0	1	2	3	
☐	☐	☐	☐	✓	☐	☐	☐	☐	0	1	1	3	0	1	2	3	0	1	2	3	0	1	3	3	
☐	☐	☐	☐	☐	✓	☐	☐	☐	0	0	0	2	0	0	0	2	1	2	2	3	0	0	0	2	
☐	☐	☐	☐	☐	☐	✓	☐	☐	0	1	1	3	0	1	2	3	0	1	2	3	0	1	2	3	
☐	☐	☐	☐	☐	☐	☐	✓	☐	0	1	1	3	0	1	2	3	0	1	2	3	0	1	2	3	
☐	☐	☐	☐	☐	☐	☐	☐	✓	0	1	1	3	0	1	2	3	0	1	2	3	0	1	2	3	

Figure 3: An example decision table opened in the expert system management GUI

5. APPLICATIONS

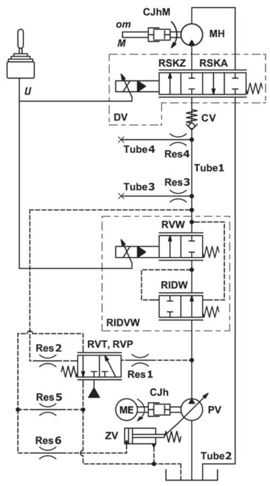
5.1 Simulation of hydraulic systems

The most sizable application of the CoCoViLa simulation tool is a modeling and simulation system for fluid power devices [5]. Fluid power systems assume a lot of drive and control tasks in machinery because of their high power density, flexible system character and required reliability. Computer modeling and simulation is an important phase in the design of such systems.

Multi-pole mathematical models and signal-flow graphs of hydraulic elements are used. This enables methodical, graphical representation of large and complicated chain systems. Simulated systems are decomposed into subsystems and functional elements. Multi-pole models of them may use programs, depending on the observed process (steady-state condition, frequency characteristics, transient responses).

Calculations are performed using multi-level method. In this way we can decompose large differential equation systems into smaller ones. First, calculations on the level of elements or subsystems are performed, thereafter variables between elements and subsystems are made congruent by iteration methods.

A library has been composed containing over hundred multi-pole models of fluid power elements that can be used for composing different schemes of fluid power devices and performing simulations. In particular modeling and simulation systems for hydraulic elements, electro-hydraulic servo systems and load sensing hydraulic systems have been developed. This is a joint work of Institute of Machinery and Institute of Cybernetics at Tallinn University of Technology.

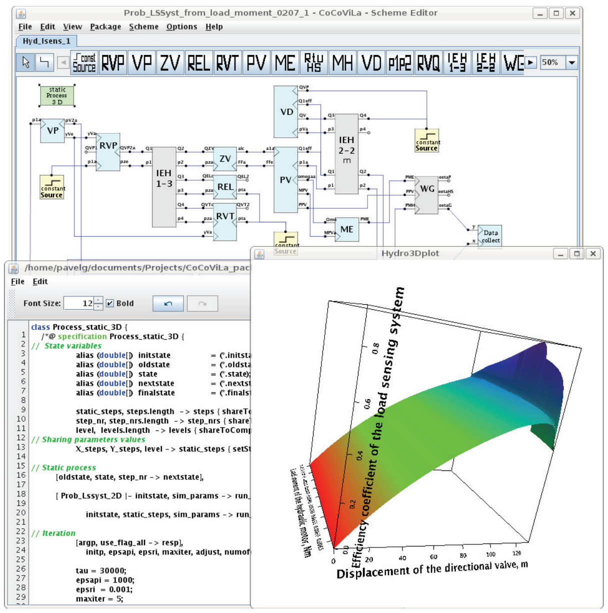


PV – Pump; CJh – Clutch; M – Electric motor; ZV – Positioning cylinder; VP – Displacement of the pump control valve; RVP – Meter-in throttle edge of the control valve; RVT – Meter-out throttle edge of the control valve; Res1,. . . ,Res6 - Hydraulic resistors; RIDVW – Measuring valve with pressure compensator; RSKZ – Check valve with meter-in throttle edge of hydraulic motor; MH – Hydraulic motor; RSKA – Meter-out throttle edge of hydraulic motor; CJhM – Clutch with drive mechanism; Tube1,. . . ,Tube4 – Tubes.

Figure 4: Functional scheme of the hydraulic-mechanical load-sensing system

The example below considers simulation of a hydraulic load sensing system shown in Figure 4. Hierarchically built model of the device includes over 4500 dependencies represented by equations and Java methods. Typically, two kinds of simulations are performed: calculating steady state conditions and dynamic responses.

To give an impression of the visual features of CoCoViLa we present in Figure 5 a screenshot of a top level description of a hydraulic simulation problem of steady state conditions together with visualization of results of simulation shown in the lower right corner of the figure.


Prob_LSSyst_from_load_moment_0207_1 - CoCoViLa - Scheme Editor
File Edit View Package Scheme Options Help
Hyd_lsens_1
const Source
RVP
VP
ZV
REL
RVT
PV
ME
MH
VD
RVQ
WG
50%
static Process 3D
IEH 1-3
IEH 2-2 m
constant Source
Data collect
/home/pavelg/documents/Projects/CoCoViLa_pac
File Edit
Font Size: 12
Bold
Hydro3Dplot
Efficiency coefficient of the load sensing system
Displacement of the directional valve, m
0
20
40
60
80
100
120
0.2
0.4
0.6
0.8

```
   class Process_static_3D {
 1    /*@ specification Process_static_3D {
 2 //  State variables
 3             alias (double[])  initstate          = (*.initsta
 4             alias (double[])  oldstate           = (*.oldsta
 5             alias (double[])  state              = (*.state);
 6             alias (double[])  nextstate          = (*.nextst
 7             alias (double[])  finalstate         = (*.finalst
 8
 9             static_steps, steps.length  -> steps { shareTo
10             step_nr, step_nrs.length  -> step_nrs { share
11             level,  levels.length  -> levels { shareToComp
12 // Sharing parameters values
13             X_steps, Y_steps, level -> static_steps { setSt
14
15 // Static process
16        [oldstate, state, step_nr -> nextstate],
17
18        [ Prob_Lssyst_2D |- initstate, sim_params -> run_
19
20                initstate, static_steps, sim_params -> run_
21
22 // Iteration
23             [argp, use_flag_all -> resp],
24                initp, epsapi, epsri, maxiter, adjust, numof
25
26             tau = 30000;
27             epsapi = 1000;
28             epsri  = 0.001;
29             maxiter = 5;
```

Figure 5: Simulation of a hydraulic-mechanical load-sensing system

The lower left window shows a piece of a text in the knowledge representation language. The text is obtained automatically from the visual description. It includes declarations of essential objects as well as logical rules for computing new values from given ones (lines from 10 to 24) and some explicitly given values (lines from 26 to 29) that have been entered through pop-up windows of the visual components. A toolbar with buttons RVP, VP etc. is scrollable and represents the concepts of the domain specific language for simulation of hydraulic systems.

Different simulation engines are used for calculating steady state conditions, 3D simulations and dynamic transient responses.

A special technique is used for calculating variables in loop dependences that can appear when a scheme of hydraulic device is composed from visual components. Splitting, using initial approximate values and iterative re-computing of the variables is used. Re-computing algorithms are constructed by the CoCoViLa program synthesizer as a result of solving corresponding subtasks. This avoids solving large equation systems during simulations.

The typical simulating task for calculating transient responses of the load-sensing system considered above contains:

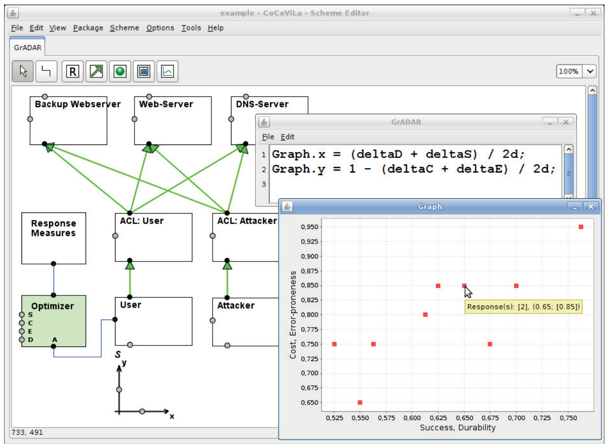
- 37 classes, including 26 functional element classes;
- 16 variables that have to be iterated during the computations.

The automatically synthesized Java code for solving the simulation task for calculating dynamic transient responses that mainly consists of calls of methods has 4449 lines and includes 4 algorithms for solving different subtasks. Its synthesis takes less than a second on a typical 2 GHz laptop. Application described above is aimed at giving to the end user a convenient tool for experimenting with different model configurations and varying parameters of the models.

5.2 Simulations in cyber security

CoCoViLa has been applied in the cyber security domain for modeling and simulation of cyber attacks and selection of optimal countermeasures. Graph-based Automated Denial-of-Service Attack Response (GrADAR) [7] is an approach where the selection of attack responses is made according to an estimation of an impact of simulated counter-attack measures. CoCoViLa was used to create a GrADAR package for visual modeling and simulation of computer networks on the basis of information such as dependencies between system resources and their availability and workload values. The optimizer component of the package implements algorithms for automatic selection and application of response measures. This is a joint work of Fraunhofer Institute for Communication, Information Processing and Ergonomics FKIE (Germany), Cooperative Cyber Defence Centre of Excellence and Institute of Cybernetics (Tallinn, Estonia) [9].

Figure 6 shows a scheme in CoCoViLa representing a dependency graph of network resources with connections for propagating workload and availability values. The goal is to simulate and analyze the effect of response measures. CoCoViLa allows not only to enter the parameter values of components using the graphical user interface, but thanks to absence of restrictions on the usage of Java language and libraries, it also allows integration with other software, e.g., back-end management systems to receive live values into the running program synthesized from the scheme.



```
Graph.x = (deltaD + deltaS) / 2d;
Graph.y = 1 - (deltaC + deltaE) / 2d;
```

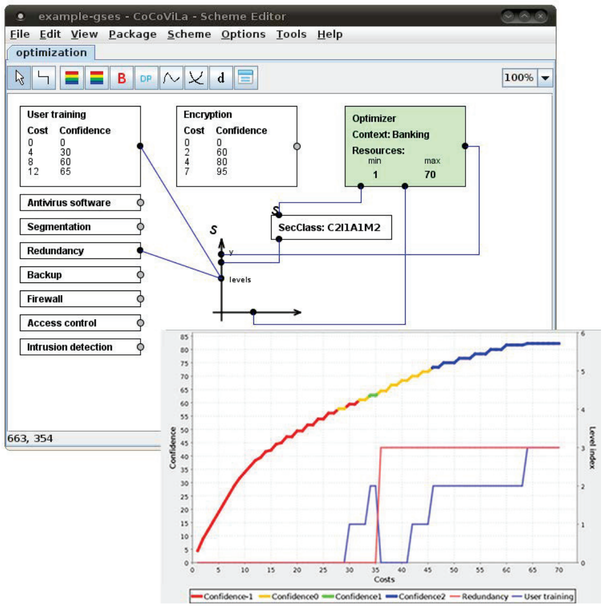
Figure 6: Visual specification of a response analysis problem in GrADAR package

5.3 Simulating graded security

This section concerns a work [8] in the field of modeling and simulation of graded security measures used in a banking security design. The graded security model is intended to help to determine reasonable or optimal sets of security measures according to the given security requirements. Currently, there are four main security goals in the model: confidentiality (C), integrity (I), availability (A) and satisfying mission criticality (M). For each goal, a certain level (0-3) is attached to specify the security requirements defined by a particular security class. As a simplified example, to achieve the security goals, the following groups of security measures might be selected for consideration: user training, antivirus software, segmentation, redundancy, backup, firewall, access control, intrusion detection, and encryption. The relative importance of a measures group can be specified by giving a weight value. In realistic scenarios studied so far the number of measures groups has been between 30 and 40.

Figure 7 shows a screenshot of the graded security expert system which consists of a visual specification language, implemented optimization algorithms and knowledge modules in the form of decision tables. In this package, security situations are described using the visual language. Suitable optimizers are applied to simulate all possible outcomes to find the Pareto-optimal set. During the simulation the optimizer queries decision tables for specific values. In the lower right corner of the figure the result of simulation, in the form of a Pareto curve and resource distribution curves, is shown.

As optimization procedures are contained in regular rich components, the algorithm used for a computation is easy to change by just replacing a component on the scheme. It can be useful to have a package contain several optimizers. Then the one making the most appropriate trade-offs for a particular experiment can be chosen by the user. In case of this package we have implemented two different optimizers, one employing brute-force and the other discrete dynamic programming algorithms. Adding a component implementing a genetic optimizer for the cases that cannot be handled by existing optimizers is anticipated in the future.



User training

Cost	Confidence
0	0
4	30
8	60
12	65

Encryption

Cost	Confidence
0	0
2	60
4	80
7	95

Optimizer
Context: Banking
Resources: min 1 max 70

Figure 7: Visual specification and the result of simulation

6. RELATED WORK

We are aware of a number of software products applicable for modeling and simulation. The ones we are mostly interested can be roughly divided into three categories: general-purpose modeling and simulation software, model-based application development software and large-scale discrete event simulation software. The common feature of most of these tools is visual specification capability by drawing schemes from components and connecting them to each other.

General purpose simulation products (Simulink [2], Scicos [1], Ptolemy II [10], etc.) typically possess built-in hybrid simulation engines and the simulation is flow-based — i.e., all the connecting arcs are directed and all the ports are either inputs or outputs. The models can be composed hierarchically and the components can be easily reused. Simulations are typically carried out on a virtual machine but also model translators exist to get code in some programming language (e.g., C) and to improve efficiency running the compiled code. Standard components (blocks) are grouped into packages (palettes). There exist large variety of blocks but also new blocks can be developed.

Model-based application develpment tools (e.g., MetaEdit [17]) have commonly a rich set of tools for program specification, analysis and verification. Taking the model-based approach to simulation development, it is easy to compose modeling and simulation applications with these tools. However, it typically needs more effort than using a dedicated tool.

Large-scale discrete event simulation tools (OPNET Modeler [14], OMNet++ [20], NS3 [15], etc.) are scalable simulators capable of handling large number of nodes and events. Their main concern is simulation performance and they are mostly used for in-detail analysis of network behavior and embedded systems, simulating all actions down to hardware level.

The CoCoViLa's planner provides flow based analysis similar to Simulink or Scicos, but its capability of constructing higher-order data flows allows us the development of model- and simulation-specific simulation engines in contrast to the mentioned tools. As the resulting simulation is always compiled into a single program it allows to achieve good simulation performance.

In comparison to other tools, CoCoViLa's support for handling of equations is very useful as this allows better reuse of rich components. For example, the ports are not defined strictly as inputs or outputs like it is the case with the tools belonging to the first two categories. It is up to the planner to decide which way the data flows in order to match the needs of the given problem—like it is the case also in real life—when defining a pipe it is not known beforehand in which direction the liquid flows in it.

Thanks to the hybrid simulation engine included in the toolbox, CoCoViLa's functionality is comparable to the general purpose simulation tools belonging to our first category. The hybrid simulation engine also covers the basic functionality of the tools belonging to the third category. However, it probably cannot always compete with specialized tools performance-wise.

Thus, although by its nature CoCoViLa is closest to the tools belonging to the second category, it is multifunctional and is more broadly applicable.

7. CONCLUSION

In this paper we described a flexible Java-based simulation platform CoCoViLa. It allows implementing simulation engines and other domain-specific simulation concepts as reusable components. The platform supports visual and model-based software development and uses structural synthesis of programs for translating declarative specifications of simulation problems into efficient executable code. To demonstrate the feasibility of our approach, several real-world applications were presented. Based on our experience, we suggest that CoCoViLa is suitable for creating and performing simulations in various engineering domains.

8. ACKNOWLEDGMENTS

This work was partially supported by the Estonian Ministry of Defence (grant No. 372/0807) and the target-financed theme No. 0322709s06 of the Estonian Ministry of Education and Research. The second and the fourth authors would like to thank the Estonian Information Technology Foundation's Tiger University+ program and the Estonian Doctoral School in ICT for supporting this work.

9. REFERENCES

[1] S. L. Campbell, J.-P. Chancelier, and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer, 2010.

[2] J. B. Dabney and T. L. Harman. *Mastering SIMULINK*. Prentice Hall, 2001.

[3] P. Grigorenko. *Higher-Order Attribute Semantics of Flat Languages*. PhD thesis, Tallinn University of Technology, 2010.

[4] P. Grigorenko, A. Saabas, and E. Tyugu. Cocovila - compiler-compiler for visual languages. *Electr. Notes Theor. Comput. Sci.*, 141(4):137–142, 2005.

[5] G. Grossschmidt and M. Harf. 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, 2009.

[6] ILTP Library homepage. http://www.cs.uni-potsdam.de/ti/iltp.

[7] M. Jahnke, G. Klein, J. Tölle, and P. Martini. Protecting military networks with gradar - an approach for graph-based automated denial-of-service attack response. In *Proceedings of the International Military Communication Conference (MCC 2009)*, Prague, Czech Republic, Sept. 2009.

[8] J. Kivimaa, A. Ojamaa, and E. Tyugu. Graded security expert system. In *CRITIS*, pages 279–286, 2008.

[9] G. Klein, A. Ojamaa, P. Grigorenko, M. Jahnke, and E. Tyugu. Enhancing response selection in impact estimation approaches. In M. Amanowicz, editor, *Concepts and Implementations for Innovative Military Communications and Information Technologies.* Military University of Technology, Warsaw, 2010.

[10] X. Liu, Y. Xiong, and E. A. Lee. The ptolemy ii framework for visual languages. In *HCC*, 2001.

[11] R. Maigre, P. Küngas, M. Matskin, and E. Tyugu. Handling large web services models in a federated governmental information system. In *ICIW*, pages 626–631, 2008.

[12] G. Mints and E. Tyugu. The programming system PRIZ. In *Baltic Computer Science*, pages 1–17, 1991.

[13] A. Ojamaa and E. Tyugu. Rich components of extendable simulation platform. In H. R. Arabnia, editor, *MSV*, pages 121–127. CSREA Press, 2007.

[14] OPNET Technologies, Inc. OPNET Solutions. http://www.opnet.com/.

[15] S. Papanastasiou, J. Mittag, E. G. Ström, and H. Hartenstein. Bridging the gap between physical layer emulation and network simulation. In *Proceedings of IEEE WCNC Conference, 2010*, April 2010.

[16] A. Repenning, A. Ioannidou, and J. Zola. Agentsheets: End-user programmable simulations. *J. Artificial Societies and Social Simulation*, 3(3), 2000.

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

[18] E. Tyugu. The structural synthesis of programs. *Algorithms in Modern Mathematics and Computer Science*, pages 290–303, 1979.

[19] E. Tyugu, M. Matskin, and J. Penjam. Applications of structural synthesis of programs. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, pages 551–569, London, UK, 1999. Springer-Verlag.

[20] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *SimuTools*, page 60, 2008.