Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D1.4.1
Simulation Demonstrator 1

Date of document: T12

Start date of the project: **1st October 2013** Duration: **36 months**

Organisation name of lead contractor for this deliverable: **UIO**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Simulation Demonstrator 1

This document summarises deliverable D1.4.1 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

This deliverable demonstrates the first prototype of the basic simulation engine for ABS, as part of the activities of Task 1.4. This task develops a simulation environment for ABS and addresses work package objective O1.4.

List of Authors

Rudolf Schlatte (UIO)

Contents

Chapter 1

Introduction

This deliverable reports on the first prototype of the basic simulation engine for ABS, as part of the activities of Task 1.4. This task develops a simulation environment for ABS and addresses work package objective O1.4. The simulation environment envisaged will combine system level descriptions with resource and deployment models to allow rapid prototyping. The simulation environment will closely reflect the formal semantics of the modeling language, yet allow the rapid prototyping of models in different deployment scenarios and with different load balancing strategies.

As mentioned in the Description of Work, the starting point for the development of the simulation environment in ENVISAGE is the ABS tool chain [3], which uses the rewriting engine Maude [1] to reflect the formal semantics directly in the simulation engine. This tool chain is written in a modular way, which makes it possible to add new features which are required for the Envisage project. In the first year of the project, the existing tool chain has been extended in various ways.

The rest of the deliverable is structured as follows. Chapter 2 gives an overview of the software and tool architecture of the demonstrator and the simulation backends. Chapter 3 shows how to use the demonstrator; Chapter 4 concludes and outlines future development.

Chapter 2

Basic Concepts

2.1 Starting Point

The starting point for the simulation demonstrator prototype is the ABS tool chain [3]. Figure 2.1 shows an overview of the architecture of the ABS tool chain. The architecture is modular; work has been ongoing on various parts in the first year of the project.

AST The abstract syntax tree (AST) definition represents an abstraction of all ABS language elements and their relations. The tool chain uses the JastAdd[1] toolkit to define the elements of the AST, thereby separating the AST definition of ABS from the concrete syntax of the language.

Parser The parser reads one or more ABS source files, detects and reports syntax errors, and creates an AST. The parser is currently being converted from a LALR grammar (using the beaver[2] parser generator and JFlex[3] scanner) to Antlr.[4] This removes many special cases from the grammar and separates code and grammar rules in different files. The benefit is a human-readable grammar that is easier to adapt for changes in the ABS language going forward.

Delta Resolver The delta resolver takes an AST in Full ABS and a list of features and creates an AST in Core ABS[5] by applying the deltas that implement the desired features. The delta resolver has seen

[1] `http://jastadd.org/`

[2] `http://beaver.sourceforge.net`

[3] `http://jflex.de`

[4] `http://www.antlr.org`

[5] Core ABS denotes the part of the language without variability constructs; a model in Full ABS can be transformed to one or more models in Core ABS according to its productline definition. The use of Full ABS is optional in the modeling process. Further details about the system variability concept of ABS and its implementation via products, features and deltas can be found in D1.1.

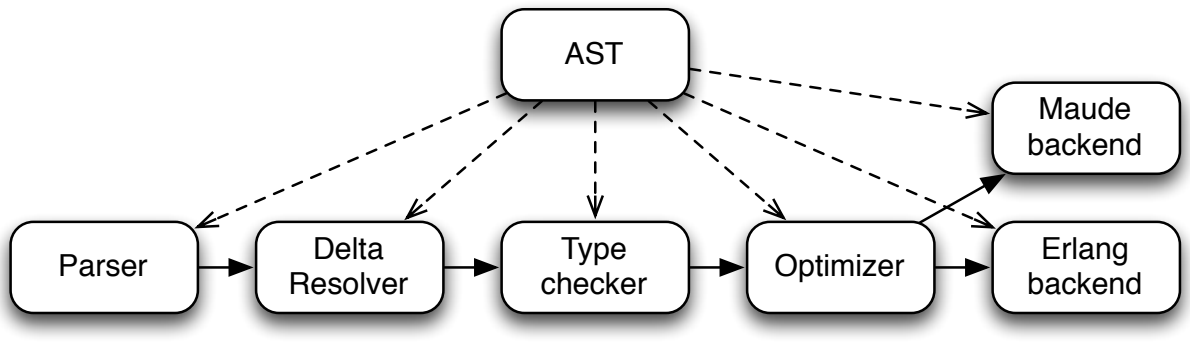

Figure 2.1: The ABS tool chain architecture

some bug fixes and minor extensions to variability.

Type checker The type checker analyzes an AST and detects type errors. No extensions to the type checker were necessary during Year 1.

Optimizer The optimizer takes an AST and returns a semantically equivalent but optimized AST. Some minor optimizations, like constant folding, have been implemented. The optimizer is not currently a focus of the Envisage project.

Backends Each backend uses the type-checked AST to generate code for its target platform. The modular design of the tool chain means that new backends can be (and have been) added easily, without having to change any other part of the system.

The individual backends of the simulator are described below in their own sections.

2.2 The Maude Backend

The Maude[6] backend is currently the main vehicle for designing prototypes for new semantics and behaviors for the ABS language. It takes the form of an interpreter which expresses the operational semantics of the language in rewriting logic and is used to simulate the behavior of ABS models on the Maude platform.

The result of a simulation run on the Maude backend is a full overview ("dump") in textual form of the internal state of all objects, cogs, deployment components and futures in the model.

The Maude backend supports the following features beyond Core ABS:

- An implementation of deployment components [2]. All cogs are associated with a deployment component, which serves as context for execution. This feature serves as input to T1.3, the results of that work task will influence the final implementation of this feature.
- An implementation of simulation time. Models have access to a clock (ranging in value from 0 to a specified value of type Rational) and can delay execution for given durations.
- A prototypical implementation of CPU and bandwidth resources, including their effect on simulation time and bookkeeping in the deployment components. This work can serve as input for experimentation both for T1.2 and WP4.

2.3 The Erlang Backend

The Erlang[7] backend consists of a code generator and runtime library for running ABS models in the Erlang VM. The core concepts of ABS (COGs, objects, futures) are mapped to Erlang constructs, and code in the model (classes, functions, the main block) is translated into Erlang and then compiled before execution.

The Erlang backend will be the main backend for the demonstrator to be delivered in D1.4.2; via this backend, the demonstrator will obtain higher speed of simulation, interactivity and the ability to influence the model at run time, and interactive visualization support.

The Erlang backend was created during the first project year and is currently in an early state of development. The Erlang backend already supports all features of Core ABS and, since delta flattening takes place before code generation, also supports modeling with features and deltas. Work is currently ongoing on supporting the timed and resource extensions to ABS which are the focus for Envisage.

[6] `http://maude.cs.illinois.edu`
[7] `http://www.erlang.org`

Chapter 3

End User Documentation

3.1 Installation Instructions

The simulation demonstrator is delivered on the same virtual machine image as the demonstrator for Deliverable D3.3.1;[1] see D3.3.1 for installation instructions.

3.2 Using the Simulator from the Command Line

3.2.1 Using the Maude Backend

Running a model on Maude involves compiling the code, then starting Maude with the resulting file as input, with the file `abs-interpreter.maude` accessible to Maude.

Compiling all files in the current directory into Maude is done with the following command:

```
$ absc -maude *.abs -o model.maude
```

The model is started with the following commands:

```
$ maude
Maude> in model.maude
Maude> frew start .
```

This sequence of commands starts Maude, then loads the compiled model and starts it. The resulting output is a dump of the complete system state after the execution of the model finishes.

It is also possible to display *intermediate states* of the simulation by restricting the number of rewriting steps that Maude should perform. The number of steps is given as an argument to the `frew` command:

```
Maude> frew [20] start .
```

The above command performs 20 rewrite steps and then displays the system state.

In case of problems when simulating on the Maude backend, check the following:

- `absc` should be in the path; check the `PATH` environment variable.
- `absfrontend.jar` should be in the environment variable `CLASSPATH`.
- `abs-interpreter.maude` should be in the same directory as the compiled model, or in a directory listed in the environment variable `MAUDE_LIB`.

[1] `http://www.envisage-project.eu/SACOVirtualMachine.ova`

3.2.2 Using the Erlang Backend

Running a model in Erlang involves compiling the ABS code, then compiling and running the resulting Erlang code.

Compiling all files in the current directory into Erlang is done with the following shell command:

```
$ absc -erlang *.abs
```

The generated files can be found in the subdirectory `gen/erl/`. The model is started with the following commands, where `Modulename` should be the name of the module containing the main block:

```
$ cd gen/erl
$ erl
1> make:all([load]).
2> runtime:start("/Modulename/").
```

This sequence of commands starts Erlang, then compiles the generated Erlang code and starts it.

The Erlang backend currently displays any `println` output during the simulation and returns the value of the last statement in the main block of the model. Future work includes introspective capabilities, both for monitoring and controlling the simulation.

3.3 Using the Simulator from the Emacs Text Editor

The Emacs editor can serve as a lightweight IDE for writing and running ABS models. It supports code highlighting and automatic indentation, code navigation and compilation and execution of ABS models inside the editor. Figure 3.1 shows a user in the process of selecting a backend (currently supported are Maude and Erlang). The model was changed after the previous compilation whose results are also shown, which is why the "Run" menu item is grayed out. Figure 3.2 shows the result of running the model on the Erlang backend inside Emacs.

To configure the ABS support inside Emacs, the easiest way is using Emacs' built-in support, via the command `M-x customize RET abs-mode RET`. In particular, if the `absc` command is not reachable from directories listed in the user's `PATH` environment variable, its location can be set in that way.

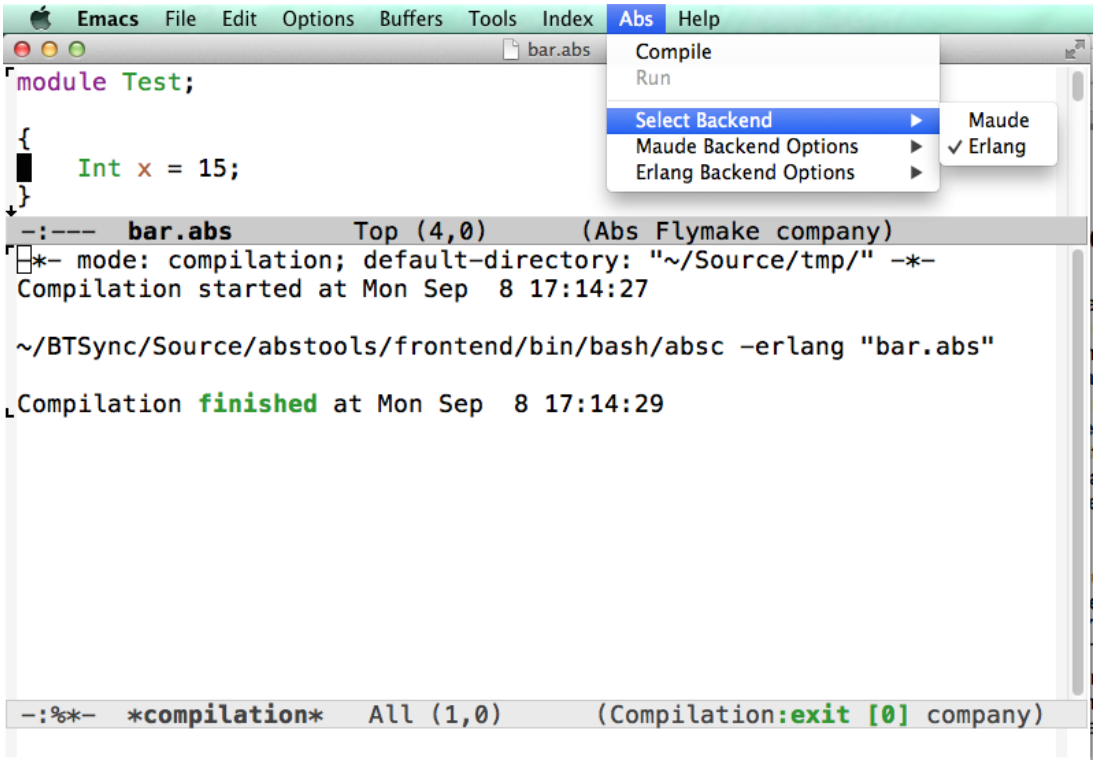

Figure 3.1: Compiling a model for the Erlang backend

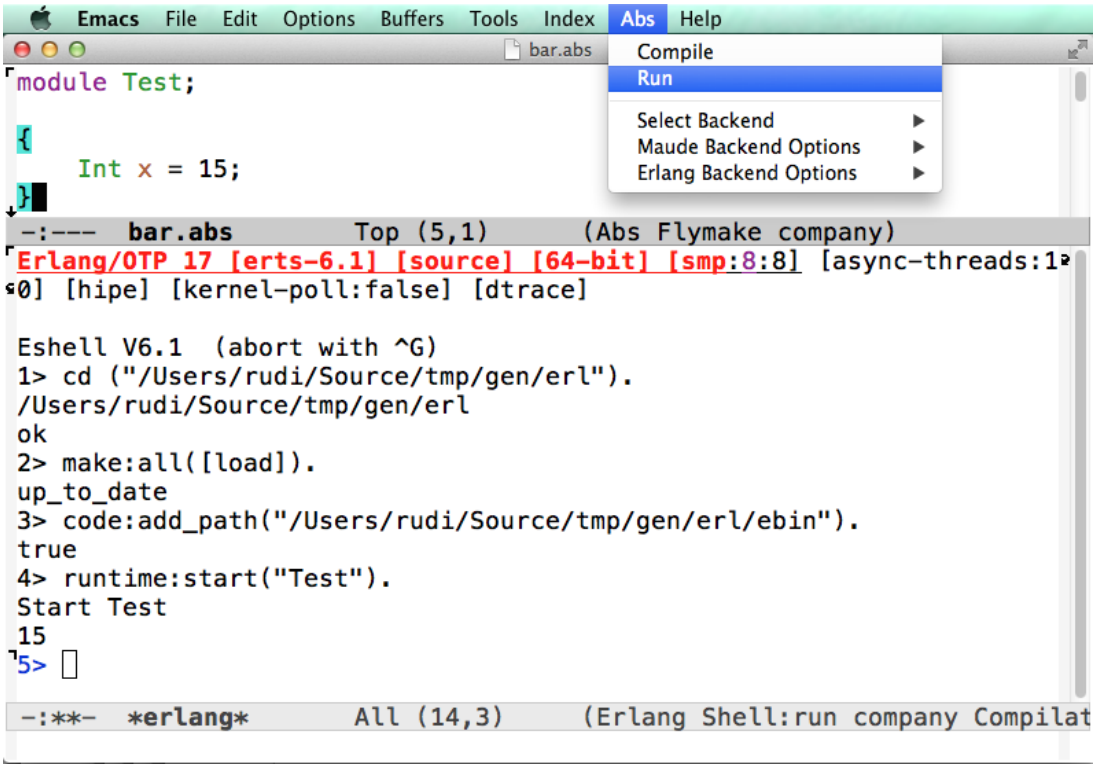

Figure 3.2: Running a model inside Emacs. The Erlang shell is started and the model is loaded and run automatically.

Chapter 4

Conclusions and Future Work

During the first year, the existing ABS tool chain has been extended in multiple ways. The main points were mentioned throughout Chapter 2. Specifically, the performance problems inherent in directly executing a formal semantics of ABS on the Maude rewriting logic engine were addressed via an additional Erlang backend.

Maintenance of the Maude backend will continue throughout Envisage. A formal semantics of ABS is advantageous for the project, and the fact that the semantics is executable makes it easy to add and evaluate new language features with minimum implementation work. Also, since Maude offers bounded execution the simulation environment already provides full access to the system state at different points during simulation.

The newly-added Erlang backend will be enhanced with the semantics of timed, resource-aware ABS, as needed to fulfill the goals of Envisage. The faster execution speed and possibility of interaction with a model running on the Erlang backend make it possible to add the visualization, querying and control capabilities to the simulation engine that are outlined in the Description of Work.

Bibliography

[1] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[2] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 2014. Available online.

[3] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.

Glossary

ANTLR A parser generator for reading, processing, executing, or translating structured text or binary files – `http://www.antlr.org`

AST Abstract syntax tree

beaver A Java-based LALR(1) parser generator – `http://beaver.sourceforge.net`

Erlang A functional programming language used to build massively scalable soft real-time systems with requirements on high availability – `http://www.erlang.org`

JastAdd A meta-compilation system that supports Reference Attribute Grammars (RAGs) – `http://jastadd.org/`

JFlex A Java-based scanner generator – `http://jflex.de`

Maude A reflective language and system supporting both equational and rewriting logic specification and programming – `http://maude.cs.illinois.edu`