

# ASCENS

## Autonomic Service-Component Ensembles

### D6.2: Second Report on WP6

#### The SCE Workbench and Integrated Tools, Pre-Release 1

Grant agreement number: **257414**  
Funding Scheme: **FET Proactive**  
Project Type: **Integrated Project**  
Latest version of Annex I: **Version 2.2 (30.7.2011)**

Lead contractor for deliverable: **CUNI**  
Author(s): **Jacques Combaz (UJF-VERIMAG), Vojtěch Horký (CUNI), Jan Kofroň (CUNI), Jaroslav Keznikl (CUNI), Alberto Lluch Lafuente (IMT), Michele Loreti (UDF), Philip Mayer (LMU), Carlo Pincioli (ULB), Petr Tůma (CUNI), Andrea Vandin (IMT)**

Reporting Period: **2**  
Period covered: **October 1, 2011 to September 30, 2012**  
Submission date: **November 12, 2012**  
Revision: **Final**  
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**  
Tel: **+49 89 2180 9154**  
Fax: **+49 89 2180 9175**  
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**



## **Executive Summary**

This text and the tools listed within constitute the ASCENS project deliverable D6.2, the first preliminary release of the tools developed and integrated with the ASCENS project. At this stage of the ASCENS project, the tools are still under development – the text presents the emerging tool landscape, explains how the individual tools contribute to the ASCENS project vision, and provides status information on the tools themselves.

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Integration Environment . . . . .	5
1.2	Current Tool Landscape . . . . .	6
1.3	Collaboration With Other Workpackages . . . . .	6
1.4	Tool Presentation Overview . . . . .	7
<b>2</b>	<b>Modeling Tools</b>	<b>9</b>
2.1	jSAM: Java Stochastic Model-Checker . . . . .	9
2.1.1	Progress and Integration . . . . .	9
2.1.2	Installation and Usage . . . . .	10
2.2	Maude Daemon Wrapper . . . . .	10
2.2.1	Progress and Integration . . . . .	10
2.2.2	Installation and Usage . . . . .	10
<b>3</b>	<b>Implementation Tools</b>	<b>12</b>
3.1	BIP Compiler . . . . .	12
3.1.1	Progress and Integration . . . . .	12
3.1.2	Installation and Usage . . . . .	13
3.2	Gimple Model Checker . . . . .	14
3.2.1	Progress and Integration . . . . .	14
3.2.2	Installation and Usage . . . . .	15
<b>4</b>	<b>Runtime Tools</b>	<b>17</b>
4.1	ARGoS . . . . .	17
4.1.1	Progress and Integration . . . . .	17
4.1.2	Installation and Usage . . . . .	18
4.2	jDEECo: Java runtime environment for DEECo applications . . . . .	18
4.2.1	Progress and Integration . . . . .	18
4.2.2	Installation and Usage . . . . .	19
4.3	jRESP: Runtime Environment for SCEL Programs . . . . .	20
4.3.1	Progress and Integration . . . . .	20
4.3.2	Installation and Usage . . . . .	20
4.4	Science Cloud Platform . . . . .	20
4.4.1	Progress and Integration . . . . .	21
4.4.2	Installation and Usage . . . . .	21
<b>5</b>	<b>Introspection Tools</b>	<b>22</b>
5.1	SPL . . . . .	22
5.1.1	Progress and Integration . . . . .	22
5.1.2	Installation and Usage . . . . .	22
<b>6</b>	<b>Conclusion</b>	<b>24</b>



# 1 Introduction

The ASCENS project tackles the challenge of building systems that are open ended, highly parallel and massively distributed. Towards that goal, the ASCENS project considers designing systems as ensembles of adaptive components. Properly designed, such ensembles should operate reliably and predictably in open and changing environments. Among the outputs of the ASCENS project are methods and tools that address particular issues in designing the ensembles.

The structure of the ASCENS project reflects the multiplicity of issues in designing the ensembles. Separate workpackages aim at topics such as formal modeling of ensembles or the knowledge representation for awareness. It is, however, important that the tools developed by the individual workpackages permit integration into a comprehensive development process. Keeping track of the tool development and directing the integration is the goal of workpackage WP6.

Following the deliverable D6.1, which collected the tool integration requirements, workpackage WP6 plans three deliverables that take the form of tool releases. These are the preliminary tool releases D6.2 and D6.3 and the final tool release D6.4. Although the eventual goal is to have the tools as much self describing as possible, with the accompanying documentation in the usually preferable form of online help, examples or tutorials, we also provide a textual deliverable that outlines the tool development and integration progress (this is important especially for tools that are still under development). Thus, the purpose of this text is to inform about progress, not to supplant the tool documentation.

This text concerns the very first release within the ASCENS project, when the tools are planned to be in an alpha stage. Both the user experience and the integration directions are just emerging and the tools are still undergoing possibly major changes. Unfortunately but unavoidably, this impacts the user experience. Most of the tools are only available from source code repositories and have to be built before being used. The usual packaging and distribution support is not yet in place (the heterogeneous character of the tools prevents straightforward use of the update sites – even if parts of the tool modules can be installed automatically, other parts are platform specific and require manual installation). Overall, we have still decided to provide detailed information on where the tool development and integration process is heading, even if that implies providing early access to many of the tools with the user experience not yet up to standards.

## 1.1 Integration Environment

In the last reporting period, the requirements on tool integration have been collected in the deliverable D6.1, which has also justified the choice of the Service Development Environment (SDE) as the tool integration platform. SDE has originated in the FP6 SENSORIA project and is currently used and developed in the FP7 ASCENS and FP7 NESSOS projects.

SDE runs on the Eclipse platform, where it facilitates orchestration of the individual tools – a particular orchestration configuration connects the inputs and outputs of the individual tools as directed to achieve the desired integration. The choice of the Eclipse platform makes the integration particularly efficient for tools compatible with OSGi. As much as it is practical, we therefore develop tools that can be packed as OSGi bundles. For tools that do not fit the OSGi bundle format, we develop wrappers as appropriate.

With the project activities focused on the individual tools, the recent SDE development has been limited to fixing minor issues and tracking the development of the Eclipse platform, which has moved from the 3.7 Indigo to the 4.2 Juno version and prompted some SDE changes.

## 1.2 Current Tool Landscape

The ASCENS project plan calls for the integration of both the development tools and the runtime tools within the SDE umbrella. This practice follows the general trend of tool integration apparent in standard integrated development environments – there, the modeling and editing tools are integrated with profiles and debuggers, making it possible to reflect the runtime observations back into the development.

On the development side, our tool support starts with the early stage formal modeling tools. At the current project stage, these tools include the jSAM stochastic model checker (Section 2.1) for the modeling approaches that rely on process algebras and the Maude Daemon Wrapper (Section 2.2) for the modeling approaches that rely on rewriting logic.

Where applicable, we continue with tools for transition from models to implementations. At the current project stage, these tools include the BIP compiler (Section 3.1) for the approaches that rely on correctness by construction. For manual implementation, we provide frameworks that reify the formal modeling concepts, at the current project stage these are jRESP (Section 4.3) and jDEECo (Section 4.2) – as explained in other deliverables, the two frameworks follow different strategies in mapping the SCEL language entities into implementation constructs.

Because the manual implementation approaches do not guarantee preserving the correspondence between the model and the code, we also develop methods and tools to verify whether code complies to models. At the current project stage, these tools are represented by the GMC model checker (Section 3.2).

On the runtime side, our tool support has to consider the differences between ensembles and more ordinary applications. The fact that ordinary applications can be launched as child processes of the integrated development environments greatly simplifies the runtime support implementation. In contrast, ensembles are not easily executed on demand – they may just be too large, or they may even consist of components that are not purely software. To cope with this particular issue, we are working on two complementary alternatives for runtime support. Where possible, such as in the scientific cloud, we plan to use live ensemble introspection. Where not possible, such as in the robotic swarms, we plan to introspect ensemble simulations.

At the current project stage, the simulation environment for the robotic swarms is ARGoS (Section 4.1). This simulation environment provides built in observation and introspection capabilities. Prototypes that act as ensemble simulators are also being built in jRESP and jDEECo. The current tool for introspection in these environments is based on the DiSL instrumentation framework [MVZ<sup>+</sup>12]. On top of DiSL, the SPL evaluation tool (Section 5.1) is used to reason about performance. Additionally, our work on these introspection tools is aligned with the development of the Science Cloud Platform (Section 4.4) to eventually allow live ensemble introspection.

## 1.3 Collaboration With Other Workpackages

Positioned as a tool integration workpackage, WP6 not only requires, but encourages and coordinates collaboration with other workpackages of the ASCENS project where tool development is concerned. Organizationally, this collaboration uses multiple venues available to the ASCENS project participants, especially personal meetings and distributed development support.

The personal meetings include the regular project meetings, where a dedicated slot is reserved for planning and coordinating the tool integration activities. In this reporting period, these were specifically the March 2012 meeting in Firenze, the May 2012 meeting in Berlin, and the July 2012 meeting in Limerick. At each of the meetings, a summary of current issues and future directions was created

and distributed among the project partners. The regular meetings are complemented with bilateral partner meetings where more detailed issues are discussed.

The distributed development support relies on tools such as source control repositories, issue trackers, blogs and wikis to maintain connection between the software development activities of the individual partners. Most tools have one partner as the primary developer, and the workpackage activities include making the development activities of this partner available to the other partners as soon as the development reaches an appropriate stage. Specific technical details on access to the individual tools are distributed through the project wiki and are also available in this deliverable.

On the thematic side, we list the collaboration areas per workpackage. Given the focus of WP6 on tool integration, the collaboration naturally revolves around the tool development activities and the tool use feedback:

- WP1 focuses on the languages for coordinating ensemble components. The collaboration between WP1 and WP6 includes providing feedback from the implementation activities into the language design effort, reflected in the SCCEL language refinements. The runtime environments for ensembles based on SCCEL models also originate in WP1. This includes the jDEECo and jRESP frameworks, described later in this deliverable.
- WP2 focuses on the models for collaborative and competitive ensembles. The collaboration between WP2 and WP6 focuses on integrating the modeling tools, which are gradually being developed. This includes especially the BIP compiler, which represents a foundational block for multiple modeling and verification tools.
- WP3 deals with knowledge modeling for ensembles. The collaboration between WP3 and WP6 follows the knowledge tool development plan. The plan focuses on the KnowLang toolset that will include editing tools, parsers and checkers, and a knowledge reasoner. The development of the KnowLang toolset has commenced and the integration requirements are distributed among the partners, especially as far as the integration of the knowledge reasoner with the runtime ensemble frameworks is concerned.
- WP4 activities concern the ensemble self expression, with modeling and simulation being prominent. The collaboration between WP4 and WP6 involves integration of the simulation environments. Here, ARGOS is a major simulation tool, whose integration is driven within WP6.
- WP5 deals with the verification techniques for components and ensembles. The collaboration between WP5 and WP6 focuses on integrating the verification tools. These are both general verification tools that are used but were not developed within the project, such as Maude, and project specific verification tools developed directly within the project, such as GMC.

Together with WP7 and WP8, the WP6 workpackage forms an integrated block of activities focused on applying the project results. Where WP6 provides tool integration, WP7 drives the case studies that use the tools, and WP8 complements the tools with other ensemble software components. The application of the tools within WP7 is described in the deliverable D7.2. The ensemble software components of WP8 are described in the deliverable D8.2.

## 1.4 Tool Presentation Overview

The next sections contain a brief description of each of the tools following a unified outline. First, the purpose of the tool is briefly outlined, together with the information on what inputs and outputs

the tool has. Next, the text describes what progress has been made since the last reporting period and what is the integration perspective. Finally, for tools whose development has progressed sufficiently, the text provides compact installation and usage directions.

To reflect the tool landscape structure, we arrange the tool descriptions into groups. In Section 2, we place tools that deal mostly with formal modeling of ensembles. In Section 3, we describe tools that help implement ensembles or simulations. Section 4 contains tools that provide runtime frameworks for executing either ensembles or simulations. Finally, Section 5 deals with tools for introspection at execution time. Of necessity, the classification categories are not entirely distinct – some tools would fall into multiple categories. Such tools are still listed under one category only, but the tool description reflects the complete purpose of the tool.



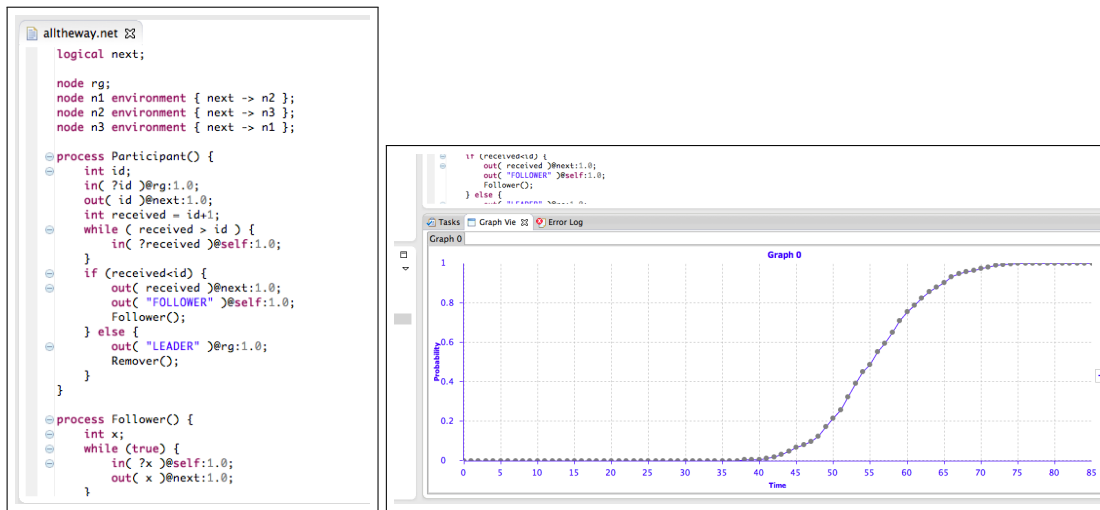


Figure 1: A jSAM specification (left) and the result of model-checking (right).

## 2 Modeling Tools

### 2.1 jSAM: Java Stochastic Model-Checker

jSAM is an Eclipse plugin integrating a set of tools for stochastic analysis of concurrent and distributed systems specified using process algebras. More specifically, jSAM provides tools that can be used for interactively executing specifications and for simulating their stochastic behaviors. Moreover, jSAM integrates a *statistical* model-checking algorithm [CL10, HYP06, QS10] that permits verifying if a given system satisfies a CSL-like [ASSB00, BKH] formula.

jSAM does not rely on a single specification language, but provides a set of basic classes that can be extended in order to integrate any process algebra. One of the process algebras that are currently integrated in jSAM is STOKLAIM [DKL<sup>+</sup>06]. This is the stochastic extension of KLAIM, an experimental language aimed at modeling and programming mobile code applications. Properties of STOKLAIM systems can be specified by means of MOSL [DKL<sup>+</sup>07] (*Mobile Stochastic Logic*). This is a stochastic logic (inspired by CSL [ASSB00, BKH]) that, together with qualitative properties, permits specifying time-bounded probabilistic reachability properties, such as *the likelihood to reach a goal state within  $t$  time units while visiting only legal states is at least  $p$* . MOSL is also equipped with operators that permit describing properties resulting from resource production and consumption. In particular, state properties incorporate features for resource management and context verification. Context verification allows the verification of assumptions on resources and processes in a system at the logical level, i.e. without having to change the model to investigate the effect of each assumption on the system behavior.

As its input, jSAM accepts a text file containing a system specification. For instance, Figure 1 (left) contains a portion of a STOKLAIM system. The results of stochastic analyses (both simulation and model-checking) are plotted in graphs, see Figure 1 (right).

#### 2.1.1 Progress and Integration

During the second year of the project, an earlier version of the tool called SAM has been ported to Java. SAM provides similar features to jSAM but is implemented in OCaML. This porting is the necessary first step towards the integration of the tool into SDE.

### 2.1.2 Installation and Usage

jSAM can be downloaded from <http://code.google.com/p/jsam>, where both the binaries and source code are available. jSAM can also be installed by relying on the Eclipse installation tools. Detailed instructions and examples are available on the same site.

## 2.2 Maude Daemon Wrapper

Maude [CDE<sup>+</sup>07] is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications. It is a flexible and general framework for giving executable semantics to a wide range of languages and models of concurrency, and has been also used to develop several tools comprising theorem provers and model checkers. As a matter of fact Maude is being used within ASCENS as a convenient formalism and tool for the modeling and analysis of self-adaptive systems (see for example [BCG<sup>+</sup>12]). The Maude Daemon Wrapper is a plugin integrating the Maude framework in the SDE environment.

Our tool is a minimal wrapper for the Maude Daemon plugin, an existing Eclipse plugin which embeds the Maude framework into the Eclipse environment by encapsulating a Maude process into a set of Java classes. The Maude Daemon plugin provides an API to use and control a Maude process from a Java program, allowing to programmatically configure the Maude process, to execute it, send commands to it, and get the results from it.

### 2.2.1 Progress and Integration

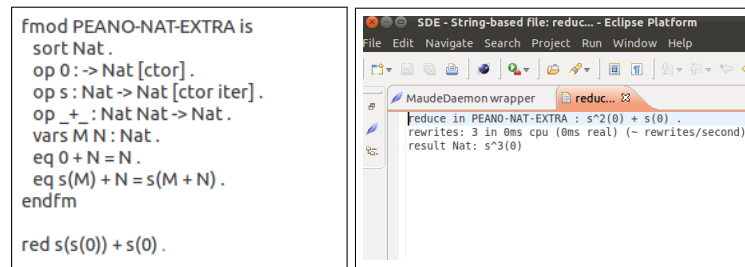
The Maude Daemon Wrapper has been developed during the second year of the project.

The Maude Daemon Wrapper facilitates the interaction of Maude with other tools registered to the SDE by exposing those features via the function `executeMaudeCommand(command, commandType, resultType)`, which takes care of initialization tasks, executes the Maude command `command`, and returns part of the Maude output as specified by `resultType`. A detailed description of Maude and its commands is available in the Maude manual at <http://maude.cs.uiuc.edu/maude2-manual>.

### 2.2.2 Installation and Usage

The Maude Daemon Wrapper plugin can be installed in Eclipse using the <http://www.albertolluch.com/updateSiteMaudeDaemonWrapper> update site (Help → Install New Software → Add). Eclipse will install all the required plugins, including the Maude Development Tools. Before actually using the plugin, it is necessary to configure the Maude Development Tools by setting the path of the Maude binaries in the preferences dialog (Window → Preferences → Maude Preferences). Once installed and configured, the plugin can be tested by opening the SDE perspective (Window → Open Perspective → Other → SDE).

The input of the Maude Daemon Wrapper consists of three Java strings: a Maude command and the command and result types. A Maude command typically contains a sequence of Maude modules (a Maude specification) and the actual command to be executed (for example `reduce t`, `rewrite t`, `search t`, with `t` being a Maude term). Figure 2 (left) exemplifies a Maude command defining the algebra of Natural numbers, followed by a command to compute the sum  $2 + 1$ . The command type is either `core` or `full`, specifying, respectively, if we are executing a core Maude or a full Maude command. Finally, the result type parameter is used to filter the Maude output, discarding eventual unnecessary information (such as the number of rewrites or the time spent to execute the command).



```
fmod PEANO-NAT-EXTRA is
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor iter] .
op _+ : Nat Nat -> Nat .
vars M N : Nat .
eq 0 + N = N .
eq s(M) + N = s(M + N) .
endfm

red s(s(0)) + s(0) .
```

```
SDE - String-based file: reduc... - Eclipse Platform
File Edit Navigate Search Project Run Window Help
MaudeDaemon wrapper
reduc...
Reduce in PEANO-NAT-EXTRA : s^2(0) + s(0) .
rewrites: 3 in 0ms cpu (0ms real) (- rewrites/second)
result Nat: s^3(0)
```

Figure 2: A Maude command (left) and its evaluation (right).

As output, the tool offers a Java string containing the output generated by Maude, filtered according to the result type given as the invocation parameter. Figure 2 (right) shows the whole Maude output obtained executing the command in Figure 2 (left).

### 3 Implementation Tools

#### 3.1 BIP Compiler

We have developed the behaviour, interaction, priority (BIP) component framework to support a rigorous system design flow. The BIP framework is:

- model-based, describing all software and systems according to a single semantic model. This maintains the overall coherency of the flow by guaranteeing that a description at step  $n + 1$  meets essential properties of a description at step  $n$ .
- component-based, providing a family of operators for building composite components from simpler components. This overcomes the poor expressiveness of theoretical frameworks based on a single operator, such as the product of automata or a function call.
- tractable, guaranteeing correctness by construction and thereby avoiding monolithic a posteriori verification as much as possible.

BIP supports the construction of composite, hierarchically structured components from atomic components characterised by their behaviour and interfaces. It lets developers compose components by layered application of interactions and priorities. This enables an expressiveness unmatched by any other existing formalism. Architecture is a first-class concept in BIP, with well-defined semantics that system designers can analyse and transform.

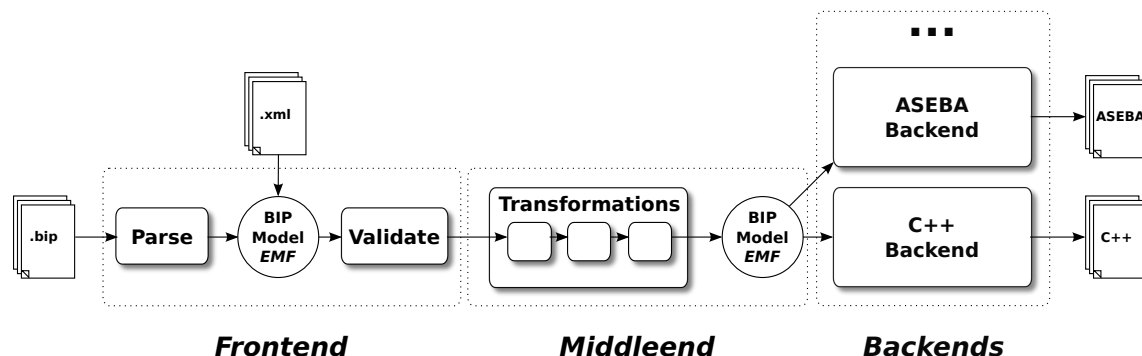


Figure 3: The BIP Compiler tool-chain.

The BIP framework is supported by a tool-chain including model-to-model transformations and code generators (see Figure 3).

##### 3.1.1 Progress and Integration

The BIP compiler and the core BIP tools have been recently rewritten. The BIP compiler is organized in Java packages in a modular way, allowing the dynamic invocation of model-to-model transformers and backends. We currently support C++ code generation, which can be executed, simulated or explored using the centralized single-thread engine. We plan to integrate the latest version of the core BIP tools, that is, the compiler and its dedicated execution engine.

The rewrite of the BIP compiler and the core BIP tools naturally impacts the additional analysis tools that rely on the BIP compiler, such as the D-Finder compositional verification tool. Updating these tools is work in progress, carried out to reflect the project tool integration requirements.

### 3.1.2 Installation and Usage

Installation instructions can be found at <http://www-verimag.imag.fr/New-BIP-tools.html>. The BIP compiler and engines are provided as an archive containing only the binaries needed for executing the tool (the sources are not freely distributed). The target platforms are GNU/Linux x86 based machines, however, the tool are known to work correctly on Mac OSX, and probably other Unix-based systems. The tool requires a Java VM (version 6 or above), a C++ compiler (preferably GCC) with the STL library, and the CMake build tool.

The installation itself consists of extracting the archive, adding the path to the `bipc.sh` compiler to `PATH`, and configuring the environment variables of the engine, namely `BIP2_ENGINE_LIB_DIR`, `BIP2_ENGINE_LIB_GENERIC_DIR`, and `BIP2_ENGINE_SPECIFIC_DIR`. The archive contains the `setup.sh` script that does this automatically.

In BIP, programs are organized in packages that are stored in separate files. Packages are collections of BIP types, that is, of atom types, connectors types and compound (hierarchical) types. Consider the BIP package containing an atom type `MyAtom` and a compound type `MyCompound`, depicted on Figure 4. The package should be saved in a file name `MyPackage.bip` (package names and their corresponding file names must match).

```
package MyPackage
  port type MyPort()

  atom type MyAtom()
    port MyPort p()
    place START,END
    initial to START
    on p from START to END
  end

  compound type MyCompound()
    component MyAtom c1()
  end
end
```

Figure 4: Example BIP package

To compile the package, first create the `out` directory to hold the generated C++ files. Next, invoke the BIP compiler (`bipc.sh`) using `bipc.sh -p MyPackage -d "MyCompound()" --gencpp-output-dir out`. The `-p` option is used to provide the name of the package to compile, the `-d` option specifies an instance of a compound type corresponding to the system to deploy, `MyCompound` in this case. The `--gencpp-output-dir` option sets the location for the generation of the C++ files. Using this option automatically invokes the C++ backend, which generates the C++ files corresponding to an instance of `MyCompound`.

The compilation of the generated C++ code is managed by `cmake`. Create a directory `build` in the directory `out`, invoke `cmake` from `build` and then `make`.

Note that the BIP execution engine is currently provided as a static library which is automatically linked when building the resulting executable system. To execute the instance of `MyCompound` by the engine, simply run `system`, as shown on Figure 5.

Default options produce a single execution sequence, in which non-deterministic choices are re-

```
$ ./system
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]:   [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: -> choose [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: state #1: deadlock!
```

Figure 5: Example BIP engine output

solved in a random fashion. The interactive mode (option `--interactive`) is useful whenever the non-deterministic choices have to be resolved by the user. Computing exhaustively all the execution sequences is achieved by using option `--explore`.

Detailed BIP documentation is available at <http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/index.html>.

## 3.2 Gimple Model Checker

GMC is an explicit-state code model checker for C programs. GMC supports threads and executes all possible interleavings to discover errors manifested only in certain thread schedules. From the ASCENS project perspective, GMC is unique in that it can check some ensemble related properties, such as particular sequences of accesses to the ensemble knowledge (using custom made listeners).

On the technical side, GMC detects low-level programming errors such as invalid memory usage (buffer overflows, memory leaks, use-after-free defects, uninitialized memory reads), null-pointer dereferences, and assertion violations. GMC understands not only the pthread library, but also offers means to add support for other thread libraries.

Same as other explicit state model checkers, GMC requires that the actions (steps) of the verified program are revertible, which is not always the case (for example when accessing hardware or external services). For such cases, the user has to create models which describe how a given action modifies the program state and how to revert the action. GMC already contains models for the basic functions from the standard C library.

The input of GMC is the source code of a complete program. The source code is processed via an extended GCC compiler [SD09], which dumps a GIMPLE file – the intermediate representation of the program used in GCC. The serialized GIMPLE representation is passed to the model checker, which interprets it and exhaustively searches for errors. If an error is found, GMC dumps a brief error description and an error trace which leads to the error.

### 3.2.1 Progress and Integration

Work on GMC in the second project year included extensions for the C++ language features, and support for custom listeners. Registered custom listeners get notified during the state space exploration as soon as a potentially interesting action, such as a method call, an instruction executed, or backtracking occurs. This extension distinguishes GMC for the purpose of the ASCENS project, where it can check ensemble related properties. Multiple bugfixes and code optimizations have also been implemented.

As the development of GMC progresses, the integrated development platform will allow using GMC on ARGoS controllers, verifying properties either encoded as assertions in the code, or specified externally.

### 3.2.2 Installation and Usage

The source code of GMC is available from <http://d3s.mff.cuni.cz/~sery/gmc/index.html>. During the installation, it is necessary to compile the extended GCC and GMC itself. A detailed step-by-step description of the installation and prerequisites can be found in the `INSTALL` file, which is provided in the source code distribution.

To run the model checker, use the prepared script `dist/GMC`. As the first parameter, the script takes either:

- i for Interpret – checks one random thread interleaving, or
- m for Model-check – explores all thread interleavings.

The remaining parameters are the files with the source code of the program to verify. GMC can also be integrated directly into a build system – in this case, the modified GCC must be used during the build, with an additional flag which prompts GCC to dump the GIMPLE file, which is later passed to the actual model checker executable `dist/ModelChecker`. In this case, the arguments are still similar to the GMC script, but instead of sources, the script expects the GIMPLE file.

```
~/GMC/dist$ ../setEnv.sh
~/GMC/dist$ ./GMC -m example.c
SOURCES=example.c
1
GIMPLEXX_FILE=gimplexx.ser
~/GMC/dist$ cat ModelChecker_stderr
Everything is OK!
Deserializing...OK.
*****
No errors detected.
```

Figure 6: GMC usage example

When running GMC, the result of the verification is displayed on the standard error output. It can also be found in the `dist/ModelChecker_stderr` file. Check Figure 6 for an output example when no error is found, and Figure 7 for an output example with an error. If an error is found, the output contains a description of the error and a sequence of the GIMPLE instructions that lead to the error.

```
~/GMC/dist$ cat ModelChecker_stderr
Everything is OK!
Deserializing...OK.
*****
Uncaught exception encountered:
Program divides by zero
*****
Logger:  Instructions executed:
Thread:  0, Instruction:  &_builtin_puts ("Rand example:"[0])
Thread:  0, Instruction:  i = &gmc_rand (3)
Thread:  0, Instruction:  D.2069 = &_GMC_EXTENSIONS_rand.1 (max)
...
```

Figure 7: GMC error trace example



## 4 Runtime Tools

### 4.1 ARGoS

ARGoS is a physics-based multi-robot simulator. ARGoS aims to simulate complex experiments involving large swarms of robots of different types in the shortest time possible. It is designed around two main requirements: efficiency, to achieve high performance with large swarms, and flexibility, to allow the user to customize the simulator for specific experiments. Besides ARGoS, no existing simulator meets both requirements. In fact, simulators that offer high efficiency typically obtain it by sacrificing flexibility. On the other hand, flexible simulators do not scale well with the number of robots.

To marry efficiency and flexibility, ARGoS is based on a number of novel design choices. First, in ARGoS, it is possible to partition the simulated space into multiple sub-spaces, managed by different physics engines running in parallel. Second, ARGoS' architecture is multi-threaded, thus designed to optimize the usage of modern multi-core CPUs. Finally, the architecture of ARGoS is highly modular. It is designed to allow the user to easily add custom features (enhancing flexibility) and allocate computational resources where needed (thus decreasing run-time and enhancing efficiency).

#### 4.1.1 Progress and Integration

Eight versions of ARGoS have been released in the course of the last years, with both bug fixes and new features. The webpage of ARGoS underwent extensive restyling, with the aim of simplifying its navigation. Forums have been added to the website to provide support to beginner users and collect tickets and feature requests.

Besides this, an article about ARGoS was published in the Linux4You magazine.<sup>1</sup> A journal paper about ARGoS design has been submitted to the Swarm Intelligence journal and is currently under review.

Because ARGoS is written in C++, integrating ARGoS with the SDE requires wrapping the ARGoS interfaces in Java. To achieve this result, two approaches are viable: (i) manually wrapping the relevant functions, or (ii) use automated tools such as SWIG.<sup>2</sup>

Manually wrapping the interface would potentially allow us to optimize the final result better than through an automated method. However, the process of manually wrapping is likely to be language-specific, while automated tools like SWIG allow to create wrappers to different languages easily.

We are considering the options of interfacing ARGoS not only with the SDE, but also directly to SCEL-based tools such as SCELua. Thus, it will be necessary to produce at least two language wrappings for the ARGoS interface — Java, for the SDE, and Lua, for SCEL-based tools. In addition, we are also considering a Python binding, which could prove useful to promote ARGoS as an educational tool for students and robotics enthusiasts.

For these reasons, we finally choose to pursue the SWIG approach. In practice, this approach consists in annotating the ARGoS interfaces with C++ comments written in a special syntax defined by the SWIG interpreter. Once annotated, the code is ready to be wrapped in any language supported by SWIG. SWIG supports all major languages, including Java, Python, Lua, Ruby, and even PHP and Perl. Generating the wrapper code is an automatic process performed by the command `swig`. For a simple example of this approach, we suggest the reader to refer to this webpage: <http://www.swig.org/tutorial.html>.

<sup>1</sup><http://www.linuxforu.com/2012/05/open-source-robotics-multi-robot-simulators>

<sup>2</sup><http://www.swig.org>

Currently, we are in the process of annotating the ARGoS interfaces. A first version of the annotations is almost finished. Once finished, the resulting wrappers will require extensive testing before reaching the next phase. The next phase consists in actually integrating the Java wrapper of ARGoS into the SDE.

#### 4.1.2 Installation and Usage

To install ARGoS, it is necessary to download a pre-compiled package from <http://iridia.ulb.ac.be/argos/download.php>. Currently, packages are available for Ubuntu/KUbuntu (32 and 64 bits), OpenSuse (32 and 64 bits), Slackware (32 bits) and MacOSX (10.6 Snow Leopard). A generic tar.bz2 package is available for untested Linux distributions. Once downloaded, the pre-compiled package should be installed using the standard package installation tools.

To use ARGoS, one must run the command `launch_argos`. This command expects two kinds of input: an XML configuration file and user code compiled into a library. The XML configuration file contains all the information required to set up the arena, the robots, the physics engines, the controllers, and so on. The user code includes the robot controllers and, optionally, hook functions to be executed in various parts of ARGoS to interact with the running experiment.

For more information, documentation and examples, refer to the ARGoS website at <http://iridia.ulb.ac.be/argos>.

## 4.2 jDEECo: Java runtime environment for DEECo applications

jDEECo is a Java-based implementation of the DEECo component model [BGH<sup>+</sup>12] runtime framework. It allows for convenient management and execution of jDEECo components and ensemble knowledge exchange.

The main tasks of the jDEECo runtime framework are providing access to the knowledge repository, storing the knowledge of all the running components, scheduling execution of component processes (either periodically or when a triggering condition is met), and evaluating membership of the running ensembles and, in the positive case, carrying out the associated knowledge exchange (also either periodically or when triggered). In general, the jDEECo runtime framework allows both local and distributed execution; currently, the distribution is achieved on the level of knowledge repository.

The jDEECo runtime framework can be initialized and executed either manually, via its Java API, or inside the OSGi infrastructure [HPMS11]. In the latter case, the modules of the jDEECo runtime framework are managed as regular OSGi services (building upon the OSGi Declarative Services). Integration into OSGi also facilitates integration into SDE.

The input of the jDEECo runtime framework is a set of definitions of the components and ensembles to be executed. In general, such definitions are represented as specifically annotated Java classes [BGH<sup>+</sup>12]. Thus, technically, the input of the jDEECo runtime framework is either a set of Java class files, a JAR file containing the class files, or a set of class objects (in case the jDEECo runtime is accessed directly via its Java API). Thanks to the OSGi integration, component and ensemble definitions may be also packaged into OSGi bundles, each containing any number of the definitions. This way, component and ensemble data can be automatically loaded whenever the bundle is deployed in an OSGi context (the SDE platform).

### 4.2.1 Progress and Integration

The jDEECo runtime framework has been wholly developed during the second year of the project, following the design of the DEECo component model.

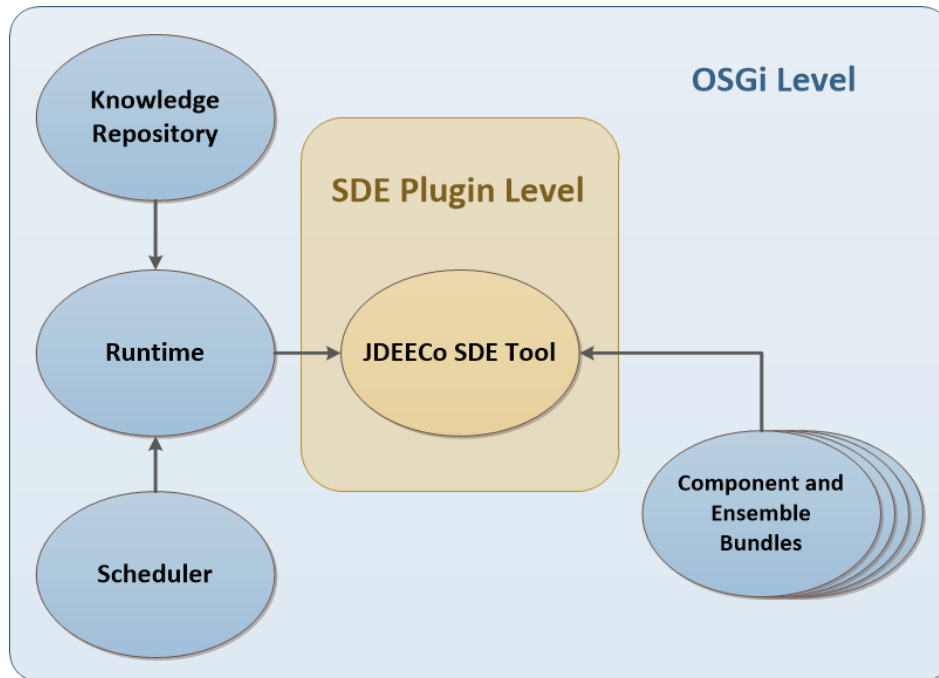


Figure 8: jDEECo SDE Tool - OSGi-SDE Integration

The integration of the jDEECo runtime into SDE allows for rapid deployment, prototyping and debugging of DEECo SCs and SCEs. Furthermore, the SDE integration platform enables easy integration with other related SC/SCE design tools such as SPL.

The jDEECo SDE plugin, integrating jDEECo into SDE, includes the jDEECo runtime implementation and an extension to the SDE management console, featuring commands for controlling the jDEECo runtime.

The jDEECo runtime interacts with the extension to the SDE management console at the OSGi level, as illustrated on Figure 8. During the SDE startup, both the jDEECo runtime and all of its modules (such as the knowledge repository) are started automatically by the OSGi layer of the SDE platform. Similarly, OSGi bundles containing the component and ensemble definitions that are deployed in the SDE platform (bundle jar files are placed inside the `plugins` folder of the SDE installation) will be automatically loaded and registered within the jDEECo runtime. Sample components and ensembles packaged into the OSGi-compliant bundles are available on the project website.

Due to technical and usability reasons, the version of jDEECo included in the jDEECo SDE plugin does not support distribution of components.

#### 4.2.2 Installation and Usage

The following instructions concern using the jDEECo runtime framework through the SDE plugin. Instructions for using the jDEECo runtime framework through the Java API are available on the project website at <https://github.com/d3scomp/jdeeco/wiki>.

To use jDEECo from SDE, download both the jDEECo SDE plugin and the jDEECo runtime framework jar files from the project website at <https://github.com/d3scomp/jdeeco> and place them in the `plugins` folder of the SDE installation.

After starting the SDE with the jDEECo plugin installed, the *jDEECo runtime manager tool*

entry will be shown in the tool browser window. The functions of the tool can be accessed either via the tool description window or via the SDE shell. The main functions include `start()` and `stop()` to start and stop the jDEECo runtime framework and execution of the registered components and ensembles. The `listAllComponents()`, `listAllEnsembles()` and `listAllKnowledge()` functions facilitate introspection of the executing components and ensembles. The full list of functions is available in the SDE shell.

### 4.3 jRESP: Runtime Environment for SCEL Programs

jRESP is a runtime environment that provides Java programmers with a framework for developing autonomic and adaptive systems based on the SCEL concepts. SCEL [DFLP11, DFLP12] identifies the linguistic constructs for modelling the control of computation, the interaction among possibly heterogeneous components, and the architecture of systems and ensembles. jRESP provides an API that permits using the SCEL paradigm in Java programs.

In SCEL, some specification aspects, such as the *knowledge representation*, are not fixed but can be customized depending on the application domain or the taste of the language user. Other mechanisms, for instance the underlying communication infrastructure, are not considered at all and remain *abstracted* in the operational semantics. For this reason, the entire framework is parametrised with respect to specific implementations of these particular features. To simplify the integration of new features, recurrent patterns are largely used in jRESP.

The jRESP communication infrastructure has been designed to avoid any *centralised control*. Indeed, a SCEL program typically consists of a set of (possibly heterogeneous) components, equipped with a knowledge repository. The components execute and cooperate in a highly dynamic environment to achieve a set of goals. The underlying communication infrastructure is not fixed, but can change dynamically during the computation. Hence, components can interact with each other by simply relying on the available communication media.

Finally, to simplify the integration with other tools and frameworks, like ARGoS and jDEECo, jRESP relies on open data interchange technologies, including json. These technologies simplify interactions between heterogeneous network components and provide the basis on which different runtimes for SCEL programs can cooperate.

#### 4.3.1 Progress and Integration

The first version of jRESP has been completely developed during the second year of the ASCENS project. The current version of jRESP is not yet integrated with SDE. The integration will take the form of a high-level programming language, which will enrich SCEL with standard programming primitives and thus simplify the development of SCEL programs.

#### 4.3.2 Installation and Usage

jRESP can be downloaded from <http://code.google.com/p/jresp>, where both the Java binaries and the source code are available. Detailed instructions and examples are available from the same site.

### 4.4 Science Cloud Platform

The Science Cloud Platform (SCP) is the software system developed as part of the science cloud case study of ASCENS. This implementation is intended for three purposes:

- First, it serves as an industry-inspired example of a cloud implementation and thus as a testbed for the languages, methods, and tools developed within ASCENS.
- Second, the implementation itself is highly flexible and allows for different approaches and ideas to be tested by integration into the platform.
- Third, the case study is intended as an application platform for end users, the end users again being scientists.

#### 4.4.1 Progress and Integration

During the last year, the specification of the science cloud platform has been finalized and is presented in [vRA<sup>+</sup>12]. The first implementation aimed at creating a baseline adaptivity functionality in the form of a failover system has been created. This system is based on OSGi and is thus able to dynamically install, start, stop, and uninstall applications in the form of bundles. Moreover, the system itself is based on OSGi bundle class loading and the OSGi service-oriented component infrastructure and can thus be easily used for testing different implementations of adaptivity and self-awareness.

Since the science cloud platform is still under development, it is not yet integrated into the SDE. It is, however, envisioned that an SDE facade can be provided for both basic lifecycle functionality (starting, stopping) as well as runtime monitoring and control.

#### 4.4.2 Installation and Usage

The progress of the Science Cloud Platform prototype created within the second year of ASCENS is being tracked on <http://svn.pst.ifi.lmu.de/trac/scp>. While this is not the final implementation which will be implemented as part of task 7.2.4 in months M32 to M48, it still serves as a target to test the ASCENS methods and tools against. On this web site, the source code is also provided under an EPL license.

The prototype is built on top of Java and OSGi and comes with a modular structure which allows for replacement of individual parts as required by the research efforts. The platform can be installed as a standalone JAR and started via command line or installed via update site into Eclipse.

To use the science cloud platform, it needs to be started multiple times, preferably on different machines. After startup, a web frontend allows querying the individual nodes, adding new links, and deploying applications. Each instance uses two ports, one for inter-client communication and one for the UI. At startup, the port numbering starts at 8000, the port number gets increased if multiple instances are started on the same host. To view the web UI of the first instance, visit <http://localhost:8001>. Port 8000 is reserved for inter-client communication.

## 5 Introspection Tools

### 5.1 SPL

SPL is a Java framework for implementing application adaptation based on observed or predicted application performance [BBH<sup>+</sup>12]. The framework is based on the Stochastic Performance Logic, a many-sorted first-order logic with inequality relations among performance observations. The logic allows to express assumptions about program performance and the purpose of the SPL framework is to give software developers an elegant way to use it to express rules controlling program adaptation.

The SPL framework internally consists of three parts that work together but can be (partially) used independently. The first part is a Java agent that instruments the application and collects performance data. The agent uses the Java instrumentation API [Ora12], the actual byte code transformation is done using the DiSL framework [MZA<sup>+</sup>12]. The second part of the framework offers an API to access the collected data and evaluate SPL formulas. The third part of the framework implements the interface between the application and the SPL framework. This API is used for the actual adaptation.

The purpose of the SPL framework is to support the adaptation of an application, however, the adaptation itself happens through means provided by the application. The framework itself does not add the actual ability to adapt. An example of an adaptation action is replicating a component in face of load changes – this action can even be provided by the platform running the application, and is considered in some of the scientific cloud use cases.

The highlights of the SPL framework are:

- The rules controlling the adaptation are described in an elegant manner using simple-to-understand formulas.
- The performance measurements use run-time bytecode instrumentation without any need to change (or even to access) the existing source code.
- The framework can be used with any Java application.

#### 5.1.1 Progress and Integration

The SPL framework has been entirely developed during the second year of the project. The integration of the SPL tool into the SDE platform is a work in progress.

#### 5.1.2 Installation and Usage

The latest version of the SPL framework can be obtained from the source repository at <http://github.com/vhotspur/spl-adaptation-framework>. There are two examples available, both can be run using the provided `build.xml` file and Apache Ant.

To use the tool on a different application than the provided examples, the user needs to prepare the agent JAR package and start the agent together with the application. As an argument to the agent, the user specifies the class that takes care of the monitoring and adaptation activities. This class should use the provided SPL framework API, which can evaluate an SPL formula over data from modular data sources, as described in [BBH<sup>+</sup>12].

To demonstrate how the SPL framework can be used to control application adaptation, a demo application is also included. The demo simulates a web store where the clients browse and purchase the available goods. The adaptation mechanism replicates certain components of the store when the number of clients increases and the store becomes overloaded.

## Performance self-awareness demo

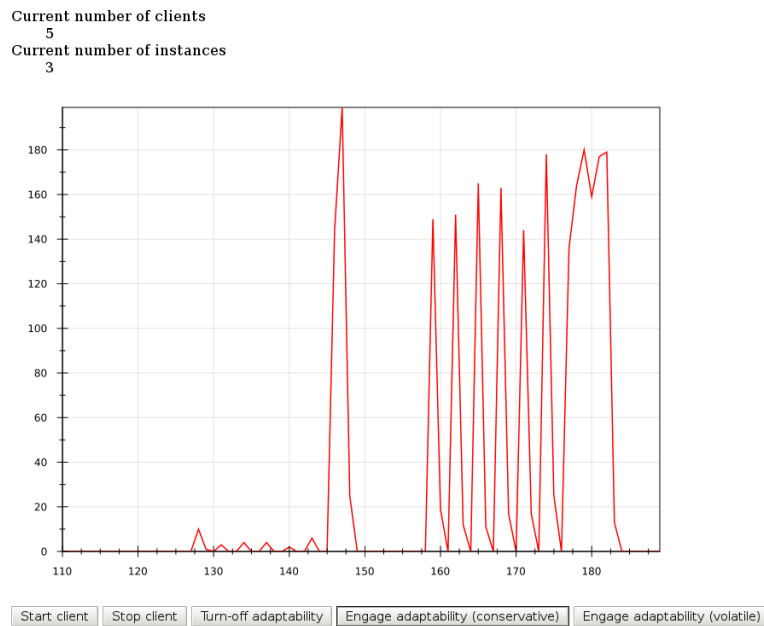


Figure 9: Adaptation demo web monitor.

The demo uses the iPOJO [EHL07] component framework for both the individual application components and the adaptation logic. The application also offers a web-based monitoring interface, see Figure 9.

The adaptation logic of the demo uses a simple SPL formula to detect when the average time to process a user request exceeds a given limit. This event triggers the component replication. Conversely, if the time falls well below the limit, one replica is stopped. The number of simulated clients and the adaptation strategy can be configured through the web interface, which also shows the number of requests that were not satisfied within the time limit.

The demo is available as a Git repository <http://github.com/vhotspur/spl-adaptation-demo>. The demo is started from the command line using `ant run`. When the demo runs, the web interface is available at `http://localhost:8888`.

## 6 Conclusion

The project structure is organized so that the tool development process is driven by two factors, namely the development of the methods and techniques for engineering ensembles, and the application of the methods and techniques on the case studies. As outlined in the relevant deliverables, both directions have progressed – the methods and techniques for engineering ensembles have progressed enough for the first models to appear and be applied on the case studies (Subtasks T7.x.2). The tool development reflects this and from the project management perspective is therefore generally on track.

The integration and simulation activities of the case studies (Subtasks T7.x.3) have begun and will progress for the next year. This progress is reflected in the tool implementation and integration efforts, which place emphasis on opening the tool development to all project partners and providing sufficient support and documentation to facilitate tool integration. Technically, this constitutes having public tool source repositories where possible, providing tool usage examples, and organizing meetings between tool authors and tool users within the project whenever necessary (the meetings take place both within and outside the framework of the regular project meetings).

From the tool integration perspective, an important project step is the availability of the first tool application examples. These make it possible to move the work on tool integration from the stage of conceptual interoperability to the stage of implementing and debugging the interoperability support in the context of the individual examples. Currently, prominent directions in tool integration include application of the GMC tool on the ARGoS robot controllers, application of the SPL tool on the jDEECo, jRESP and SCP runtime platforms for implementation of performance awareness, and application of the jSAM tool on the relevant models developed within the case studies ; additional tool integration opportunities are envisioned as the tool implementation progresses.

Finally, the implementation and evaluation activities of the case studies (Subtasks T7.x.4), which are scheduled for the concluding phase of the project, will also impact the tool implementation and integration efforts. To prevent unexpected disruptions in later project stages, which might be difficult to reflect at implementation level due to additional development effort requirements, we already stress the application of the tools on examples motivated by the case studies – such as the use of the robotic playground example in the jDEECo tool tutorial.



## References

- [ASSB00] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous time Markov chains. *Transactions on Computational Logic*, 1(1):162–170, 2000.
- [BBH<sup>+</sup>12] Lubomir Bulej, Tomas Bures, Vojtech Horky, Jaroslav Keznikl, and Petr Tuma. Performance Awareness in Component Systems: Vision Paper. COMPSAC '12, 2012.
- [BCG<sup>+</sup>12] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with maude. In *Proceedings of the 9th International Workshop on Rewriting Logic and its Applications (WRLA 2012)*, number 7571 in LNCS, pages 18–138, 2012.
- [BGH<sup>+</sup>12] Tomas Bures, Ilias Gerostathopoulos, Vojtech Horky, Jaroslav Keznikl, Jan Kofron, Michele Loreti, and Frantisek Plasil. Language Extensions for Implementation-Level Conformance Checking. ASCENS Deliverable D1.5, 2012.
- [BKH] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. pages 146–162.
- [CDE<sup>+</sup>07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude*, volume 4350 of LNCS. Springer, 2007.
- [CL10] Francesco Calzolari and Michele Loreti. Simulation and analysis of distributed systems in klaim. In Dave Clarke and Gul A. Agha, editors, *Coordination Models and Languages, 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, volume 6116 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2010.
- [DFLP11] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. Languages primitives for coordination, resource negotiation, and task description. ASCENS Deliverable D1.1, September 2011. <http://rap.dsi.unifi.it/scel/>.
- [DFLP12] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A language-based approach to autonomic computing. In *Proc. of the 10th International Symposium on Software Technologies Concertation on Formal Methods for Components and Objects (FMCO 2011)*, *Lecture Notes in Computer Science*. Springer, 2012. To appear.
- [DKL<sup>+</sup>06] R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, and M. Massink. Klaim and its stochastic semantics. Technical report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2006.
- [DKL<sup>+</sup>07] Rocco De Nicola, Joost-Pieter Katoen, Diego Latella, Michele Loreti, and Mieke Massink. Model checking mobile stochastic logic. *Theoretical Computer Science*, 382(1):42–70, 2007.
- [EHL07] C. Escoffier, R.S. Hall, and P. Lalanda. ipoyo: an extensible service-oriented component framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474–481, july 2007.
- [HPMS11] R. Hall, K. Pauls, S. McCulloch, and D. Savage. *Osgi in Action: Creating Modular Applications in Java*. Manning Pubs Co Series. Manning Publications, 2011.

- [HYP06] G. Norman H. Younes, M. Kwiatkowska and D. Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, June 2006.
- [MVZ<sup>+</sup>12] Lukas Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. DiSL: a domain-specific language for bytecode instrumentation. In *AOSD '12: Proceedings of the 11th International Conference on Aspect-Oriented Software Development*, pages 239–250, 2012.
- [MZA<sup>+</sup>12] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Walter Binder, Zhengwei Qi, and Petr Tuma. DiSL: An extensible language for efficient and comprehensive dynamic program analysis. In *Proc. 7th Workshop on Domain-Specific Aspect Languages, DSAL '12*, pages 27–28, New York, NY, USA, 2012. ACM.
- [Ora12] Oracle. `java.lang.instrument` (java platform se 6), 2012.
- [QS10] Paola Quaglia and Stefano Schivo. Approximate model checking of stochastic COWS. In *Proc. of TGC 2010*. To appear., 2010.
- [SD09] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.
- [vRA<sup>+</sup>12] Nikola Šerbedžija, Stephan Reiter, Maximilian Ahrens, José Velasco, Carlo Pinciroli, Nicklas Hoch, and Bernd Werther. D7.2: Second Report on WP7: Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility. ASCENS Deliverable, November 2012.