

ASCENS

Autonomic Service-Component Ensembles

D6.3: Third Report on WP6

The SCE Workbench and Integrated Tools, Pre-Release 2

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): **D. Abeywickrama (UNIMORE), J. Combaz (UJF-Verimag), V. Horký, J. Kofroň, J. Keznikl (CUNI), A. Lluch (IMT), M. Loreti, A. Margheri (UDF), P. Mayer (LMU), V. Monreale, U. Montanari (UNIFI), C. Pincioli (ULB), P. Tůma (CUNI), A. Vandin (IMT), E. Vassev (UL)**

Reporting Period: **3**
Period covered: **October 1, 2012 to September 30, 2013**
Submission date: **November 8, 2013**
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, UNIFI, 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 D6.3 – the second preliminary release of the tools developed and integrated with the ASCENS project, which supersedes the ASCENS project deliverable D6.2 – the first preliminary release of the tools. 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.

The deliverable text is designed to form a standalone material. While this makes the deliverable somewhat longer, it is necessary to make the text reasonably readable without the need to refer to the past tool description deliverables. Where required, the tool descriptions from these deliverables are reproduced with updates to reflect the current project status.

Contents

1	Introduction	5
1.1	Integration Environment	5
1.2	Current Tool Landscape	6
1.3	Collaboration With Other Workpackages	7
1.4	Tool Presentation Overview	8
2	Design Cycle Tools	9
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
2.3	MESSI: Maude Ensemble Strategies Simulator and Inquirer	11
2.3.1	Progress and Integration	12
2.3.2	Installation and Usage	12
2.4	MISSCEL: a Maude Interpreter and Simulator for SCEL	13
2.4.1	Progress and Integration	13
2.4.2	Installation and Usage	13
2.5	SimSOTA	13
2.5.1	Progress and Integration	14
2.5.2	Installation and Usage	15
2.6	FACPL: Policy IDE and Evaluation Library	15
2.6.1	Progress and Integration	16
2.6.2	Installation and Usage	16
2.7	KnowLang Toolset	16
2.7.1	Progress and Integration	17
2.7.2	Installation and Usage	18
2.8	BIP Compiler	18
2.8.1	Progress and Integration	18
2.8.2	Installation and Usage	19
2.9	Gimple Model Checker	19
2.9.1	Progress and Integration	19
2.9.2	Installation and Usage	20
3	Runtime Cycle Tools	21
3.1	ARGoS	21
3.1.1	Progress and Integration	21
3.1.2	Installation and Usage	22
3.2	jDEECo: Java runtime environment for DEECo applications	22
3.2.1	Progress and Integration	23
3.2.2	Installation and Usage	23
3.3	jRESP: Runtime Environment for SCEL Programs	24
3.3.1	Progress and Integration	25
3.3.2	Installation and Usage	25
3.4	CIAO Environment	25
3.4.1	Progress and Integration	26

3.4.2	Installation and Usage	26
3.5	Science Cloud Platform	26
3.5.1	Progress and Integration	26
3.5.2	Installation and Usage	27
3.6	SPL	27
3.6.1	Progress and Integration	28
3.6.2	Installation and Usage	28
4	Conclusion and Plans for Year Four	30

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, and the deliverable D6.2, which presented the first preliminary tool release, workpackage WP6 now presents the second preliminary tool release in deliverable D6.3 and eventually plans the final tool release D6.4. Although the final tool release goal is to have all 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 second tool release within the ASCENS project, when the tools are planned to be in a beta stage. Both the user experience and the integration directions are still being tuned 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 first 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.

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. This also reflects the ASCENS approach to the software development lifecycle, illustrated on Figure 1 and described in detail in joint deliverable JD3.2.

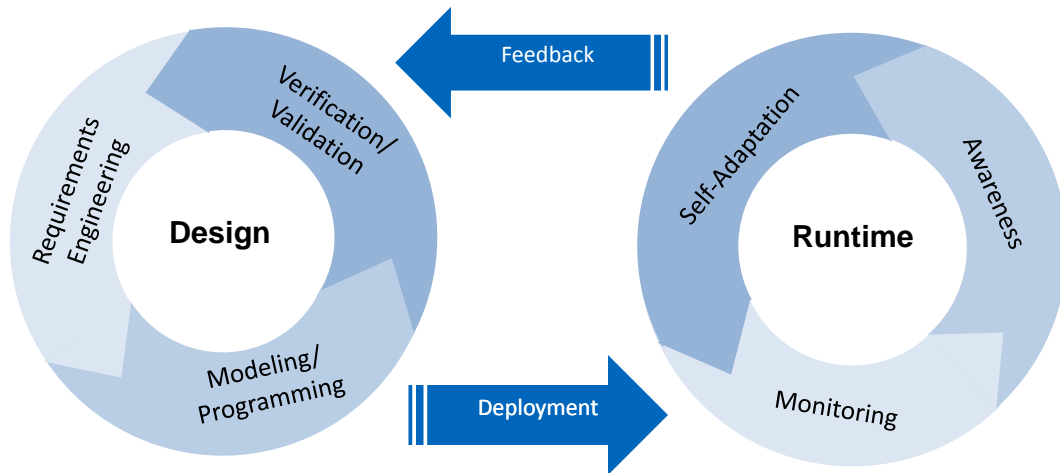


Figure 1: ASCENS Ensemble Software Development Life Cycle

On the design cycle 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 – two frameworks that rely on Maude, MESSI (Section 2.3) and MISSCEL (Section 2.4), have also been developed. The SimSOTA tool (Section 2.5) can evaluate the behavior of complex feedback driven adaptation mechanisms using simulation. The FACPL framework (Section 2.6) can be used to capture policies that regulate interaction and adaptation of SCEL components. The KnowLang Toolset (Section 2.7) serves to describe knowledge models which are then compiled into a binary knowledge base, to be used for subsequent knowledge reasoning tasks.

Where applicable, we continue with tools for transition from modeling to programming. At the current project stage, these tools include the BIP compiler (Section 2.8) 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 3.3) and jDEECo (Section 3.2) – 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 2.9).

On the runtime cycle 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 3.1). This simulation environment provides built in observation and introspection capabilities. Prototypes that act as ensemble simulators are also being built in jRESP and jDEECo. Where the ensemble execution requires reasoning support, we work on connecting the required framework to the runtime, such as with the CIAO Environment (Section 3.4), used to provide the soft constraint logic programming framework for optimization decisions.

Besides the internal introspection features of the runtime environment, we also provide a tool for explicit introspection based on the DiSL instrumentation framework [MVZ⁺12], which has enough flexibility to observe most Java applications. On top of DiSL, the SPL evaluation tool (Section 3.6) is used to reason about performance. Additionally, our work on these introspection tools is aligned with the development of the Science Cloud Platform (Section 3.5) to 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 February 2013 meeting in Prague, the May 2013 meeting in Munich, and the July 2013 meeting in Lausanne. 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 and SimSOTA are the major simulation tools incorporated 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 deliverables D7.2 and D7.3. 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, where applicable also explaining 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 ASCENS approach to the software development lifecycle, we arrange the tool descriptions into two large groups. In Section 2, we place tools that deal mostly with the design cycle side, such as the modeling activities. Section 3 contains tools that provide runtime frameworks for executing either ensembles or simulations. Of necessity, the classification categories are not entirely distinct – some tools would fall into both groups. Such tools are listed only once, but the tool description reflects the complete purpose of the tool.

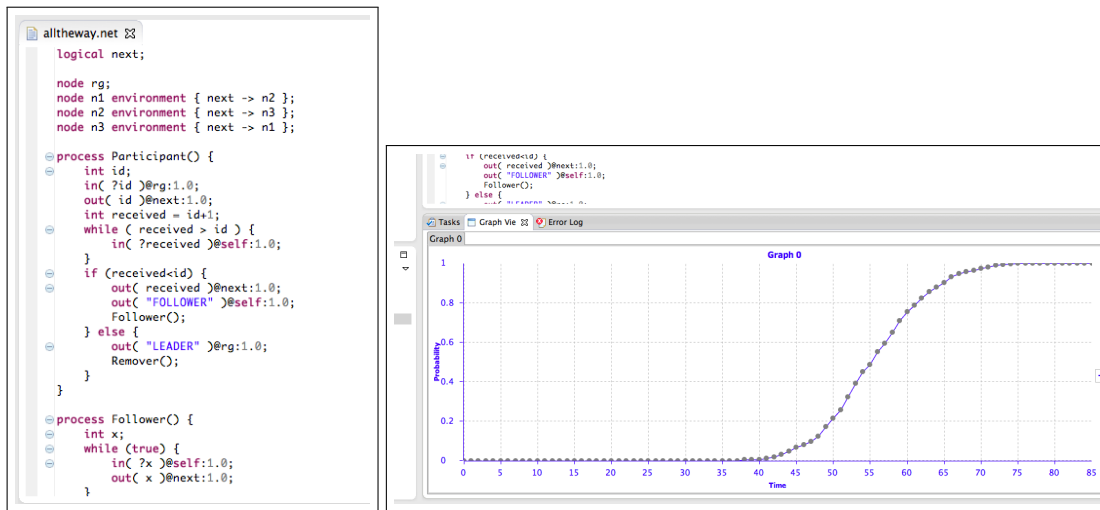


Figure 2: A jSAM specification (left) and the result of model-checking (right).

2 Design Cycle 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⁺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⁺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 2 (left) contains a portion of a STOKLAIM system. The results of stochastic analyses (both simulation and model-checking) are plotted in graphs, see Figure 2 (right).

2.1.1 Progress and Integration

In the third year of the project, we have completed the porting of SAM, an earlier version of the tool in OCaML, and we have also defined the general structure of the plug-in that can be used to integrate new calculi/languages.

In the next year, we plan to integrate a stochastic extension of SCEL. Moreover, we plan to complete the integration of the jSAM into SDE.

2.1.2 Installation and Usage

At this moment, a packaged version of jSAM is not available. It is being prepared as a part of the SDE integration.

2.2 Maude Daemon Wrapper

Maude [CDE⁺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. Maude is used within the ASCENS project as a convenient formalism for modeling and analysis of self-adaptive systems, as outlined for example in [BCG⁺12a, BCG⁺12d, BDVW]. Maude can be used to prototype semantic models and then either execute or check them. Maude can also be used as a semantic framework for SCEL dialects, for instance to develop interpreters or analysis tools for SCEL specifications. Maude can also be used to model the case studies. Sections 2.3 and 2.4 present two tools that pursue these research lines.

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. Based on the wrapper, we have integrated the Maude-based tool MISSCEL, presented in Section 2.4, and we plan to similarly integrate another Maude-based tool named MESSI, presented in Section 2.3.

The Maude Daemon Wrapper facilitates the interaction of Maude with other tools registered with the SDE by exposing those features via the function `executeMaudeCommand(command, commandType, resultType)`, which takes care of the initialization tasks, executes the Maude command `command`, and returns the 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. 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. Once installed and configured, the plugin can be tested by opening the SDE perspective.

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

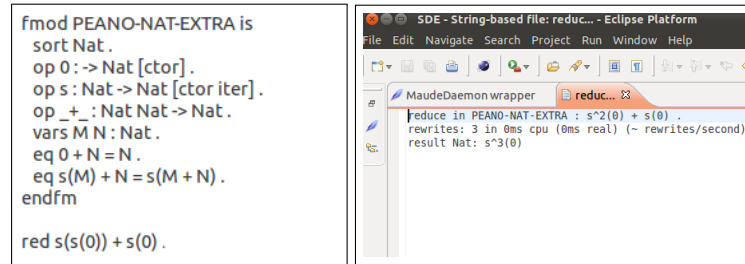


Figure 3: A Maude command (left) and its evaluation (right).

(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 3 (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).

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 3 (right) shows the whole Maude output obtained executing the command in Figure 3 (left).

2.3 MESSI: Maude Ensemble Strategies Simulator and Inquirer

As part of a research line pursued in collaboration between the project partners [BCG⁺12c, BCG⁺12a, BCG⁺12d, BCG⁺12b], we investigated the use of Maude, and of its rich toolset [CDE⁺07], to model and analyze self-assembly robotic strategies proposed by IRIDIA [OGCD10]. The obtained outcome is a framework named MESSI (Maude Ensemble Strategy Simulator and Inquirer) [BCG⁺12a, BCG⁺12d, ML] that helps model, debug and analyze scenarios where s-bots self-assemble to solve tasks (e.g. crossing holes or hills). Debugging is done via animated simulations, while analysis can be done by exploiting the Maude toolset, and in particular the distributed statistical analyzer and statistical model checker PVeStA [AM11, SVA05], or via the recently proposed MultiVeStA [SV], which extends PVeStA.

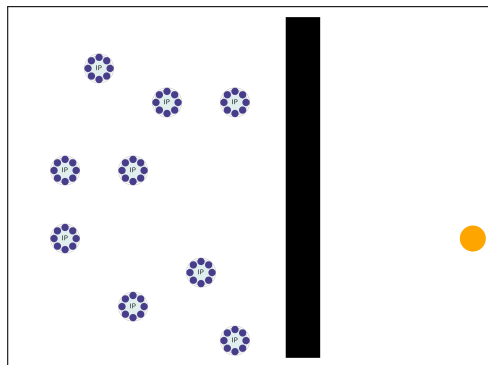


Figure 4: A pictorial representation of an initial configuration for MESSI.

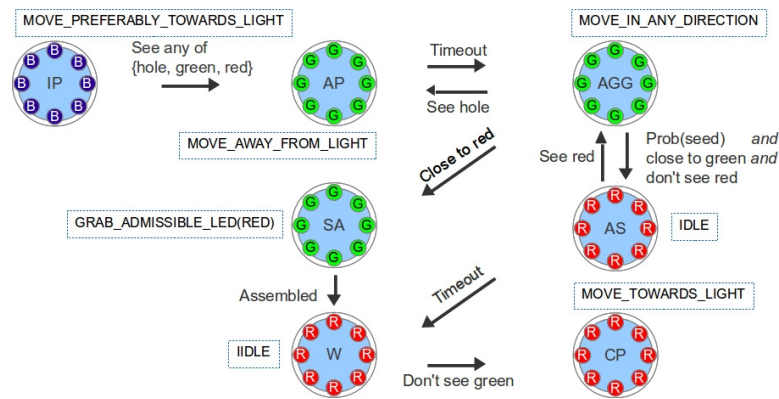


Figure 5: A pictorial representation of a self-assembly strategy for MESSI.

2.3.1 Progress and Integration

MESSI has been developed during the third year of the project. It is currently not integrated in the SDE, but the eventual integration with the help of the Maude Daemon Wrapper from Section 2.2 is planned. MESSI currently comes as a set of Maude files to be imported by the specifications of self-assembly strategies provided by the users.

2.3.2 Installation and Usage

MESSI can be downloaded from its website [ML], where the usage description is also provided. An example of the analysis activities that can be performed with MESSI is provided in the deliverable JD3.1. Additionally, the deliverable JD3.2 also discusses use of MESSI for the robotics case study.

The inputs of MESSI are the initial configuration and the self-assembly strategy, provided as Maude modules. The former provides information about the environment (an arena), specifying the presence of obstacles and targets (e.g. particular sources of light), and about the numbers and positions of the robots. The latter specifies the behaviour of the robots in the form of a finite state machine, which will be independently executed by each robot. Figures 4 and 5 provide a pictorial view of the two inputs. Figure 4 depicts an initial configuration with 9 robots distributed in an arena. The robots have to reach the target (the orange circle) situated behind a hole too large to be crossed by any single robot. Figure 5 depicts the *basic self-assembly response strategy* (BSRS) proposed in [OGCD10]. The strategy specifies the possible states (each circle is a bird-eye view of a robot) of the robots (i.e. the different mode of operation that the robots have) and the status of the robots LED signals (used to communicate with other robots) in each state. The transitions among the states provide the conditions that trigger a change of state of a robot, i.e., an adaptation.

MESSI provides a library of predefined basic behaviours (e.g. *move towards light*, or *search a given color emission and grab its source*), thus a self-assembly strategy is specified by just providing the list of states, the correspondence between the states and the basic behaviours, the status of the LED signals in each state, and a conditional rewrite rule for each transition of the finite state machine, with the condition as the label of the transition.

Given an initial configuration and a self-assembly strategy, MESSI allows to generate probabilistic simulations. As discussed, such simulations can be used to debug the strategy, or to measure its performance via statistical quantitative analysis.

2.4 MISSCEL: a Maude Interpreter and Simulator for SCEL

The SCEL language comes with solid semantics foundations laying the basis for formal reasoning. MISSCEL, a rewriting-logic-based implementation of the SCEL operational semantics is a first step in this direction. MISSCEL is written in Maude, which allows to execute rewrite theories – what we obtain is an executable operational semantics for SCEL, that is, an interpreter. Given a SCEL specification, thanks to MISSCEL it is possible to use the rich Maude toolset [CDE⁺07] to perform (i) automatic state-space generation, (ii) qualitative analysis via Maude invariant and LTL model checkers, (iii) debugging via probabilistic simulations and animations generation, (iv) statistical quantitative analysis via the recently proposed MultiVeStA [SV] statistical analyser that extends PVeStA [AM11, SVA05].

A further advantage of MISSCEL is that SCEL specifications can now be intertwined with raw Maude code, exploiting its great expressiveness. This allows to obtain cleaner specifications in which SCEL is used to model behaviours, aggregations, and knowledge manipulation, leaving scenario-specific details like environment sensing abstractions or robot movements to Maude.

2.4.1 Progress and Integration

MISSCEL has been developed during the third year of the project. In principle, the tool can be used as a standalone Maude file, however, by using the Maude Daemon Wrapper from Section 2.2, we have developed an Eclipse plugin wrapping MISSCEL – here called jMISSCEL – which we have integrated in the SDE.

2.4.2 Installation and Usage

The jMISSCEL plugin can be installed in Eclipse using the <http://sysma.lab.imtlucca.it/updateSiteJMISSCEL> update site. 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. Once installed and configured, the plugin can be tested by opening the SDE perspective.

The jMISSCEL plugin offers several methods that allow tools registered with the SDE to exploit the Maude toolset to perform several actions on SCEL specifications. As their inputs, all the methods accept a SCEL specification plus other necessary or specific parameters (e.g. the root module containing the SCEL specification or the LTL formula to be checked). We provide methods to generate the state space of a SCEL specification by exploiting the Maude *search* command (these can also be just the states satisfying boolean conditions definable as Maude operations on SCEL configurations). After the generation of the state space, it is possible to obtain the path that generated one of the returned states, or the whole search graph (similar to a labelled transition system). Moreover, it is possible to model-check SCEL specifications, resorting to the LTL model checker. Finally, by resorting to a set of schedulers that we defined to transform the non-determinism of SCEL in probabilistic choices, it is possible to generate probabilistic simulations of a SCEL specification. We have also defined an exporter from SCEL configurations to DOT terms [AL], using which we can obtain images from SCEL configurations and animate the simulations.

2.5 SimSOTA

Engineering a decentralized system of autonomous service components and ensembles is very challenging for software architects. This is because there are a number of service components and managers that close multiple, interacting feedback loops. To better understand this complex setup, solid

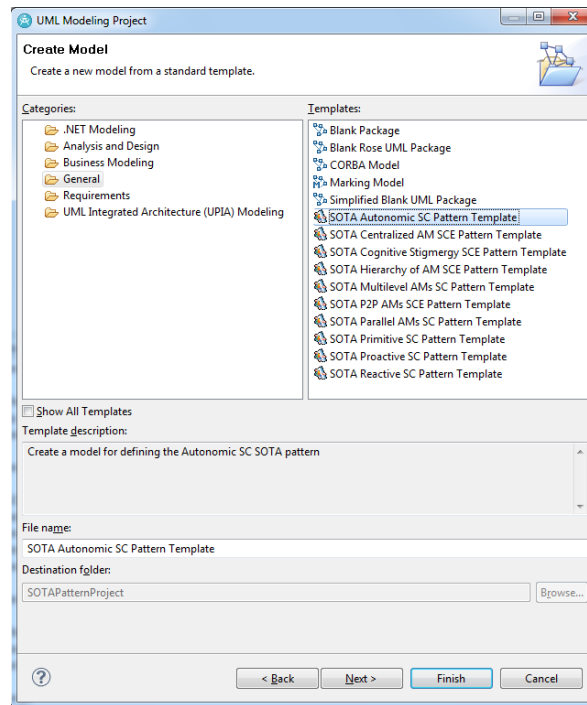


Figure 6: SOTA pattern templates available to facilitate modeling.

software engineering methods and tool support are highly desirable. Although several existing works (e.g. [MPS08, HGB10, VWMA11, RHR11, WH07, LNGE11, VG12]) have addressed the need to make feedback loops explicit or first-class entities, very little attention has been given to providing actual tool support for the explicit modeling of these feedback loops, their simulation and validation. This provides motivation for SimSOTA.

The SimSOTA tool has been developed using the IBM Rational Software Architect Simulation Toolkit. It supports modeling, simulating and validating of self-adaptive systems based on the feedback loop-based approach, and the generation of pattern implementation code using transformations.

2.5.1 Progress and Integration

The initial results of using SimSOTA to support engineering (modeling, animating and validating) of self-adaptive systems based on feedback loops was reported in [AHZ13, AZH12], and is detailed in deliverable D4.2. At present, the SimSOTA tool allows to architect, engineer and implement self-adaptive systems with feedback loops. We adopt the model-driven development process to model and simulate complex self-adaptive architectural patterns, and to automate the generation of Java implementation code for the patterns. Our work integrates both decentralized and centralized feedback loop techniques to exploit their benefits.

The SimSOTA tool provides a set of pattern templates for the key SOTA patterns, depicted on Figure 6. This facilitates general-purpose and application-independent instantiation of models for complex systems based on feedback loops. The SimSOTA tool applies model transformations to automate the application of UML architectural design patterns and generate infrastructure code for the patterns in Java. The generated Java files of the SOTA patterns can be further adjusted by the engineer to derive a complete implementation for the patterns. To assist this process, we provide a set of context-independent Java templates, which can be instantiated to a particular domain.

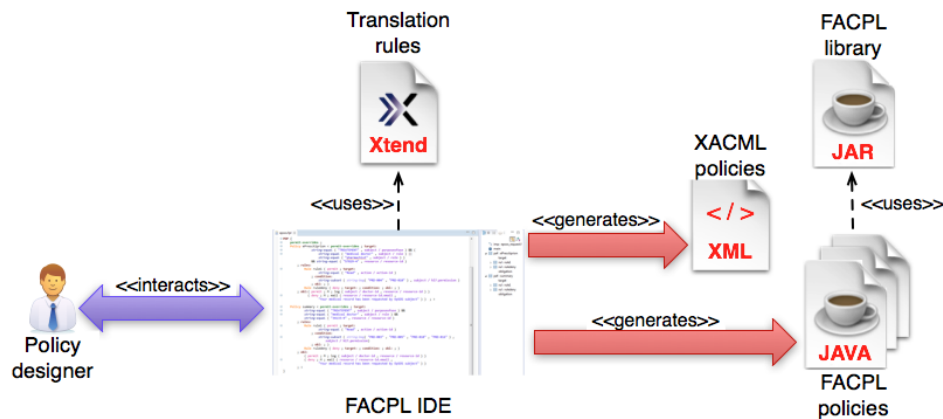


Figure 7: FACPL Toolchain

2.5.2 Installation and Usage

The distribution scheme adopted for SimSOTA relies on the Eclipse platform feature export. The entire tool can be downloaded as a plug in using the standard Eclipse update site mechanism. At this moment, however, a packaged version of SimSOTA is not publicly available, due to its dependencies on the (non free) IBM Rational Software Architect environment.

2.6 FACPL: Policy IDE and Evaluation Library

FACPL [MMPT13a] is a tiny policy language for writing policies and requests. It has a mathematically defined semantics and can be used to regulate interaction and adaptation of SCCL components. FACPL provides user-friendly, uniform, and comprehensive linguistic abstractions for policing various aspects of system behaviour, as e.g. access control, resource usage, and adaptation. The result of a request evaluation is an authorisation decision (e.g. permit or deny), which may also include some obligations, i.e. additional actions to be executed for enforcing the decision.

The development and the enforcement of FACPL policies is supported by practical software tools – an Integrated Development Environment (IDE), in the form of an Eclipse plugin, and a Java implementation library. Figure 7 shows the toolchain supporting the use of the language. The policy designer can use the IDE for writing the desired policies in FACPL syntax, by taking advantage of the supporting features provided, e.g. code completion and syntax checks. Then, the tool automatically produces a set of Java classes implementing the FACPL code by using the specification classes defined in the FACPL library. The library, according to the rules defining the language semantics, implements the request evaluation process, given as input a set of Java-translated policies and the request to evaluate.

The policy and request specification are facilitated both by the high abstraction level of FACPL and by the graphical interface provided by our IDE, of which Figure 8 shows an example. By exploiting some translation rules, written using the Xtend language, the IDE generates also the corresponding low-level policies in XML. This format obeys the XACML¹ 3.0 syntax and can be used to connect our toolchain to external tools supporting XACML.

¹XACML is a standard for access control systems.

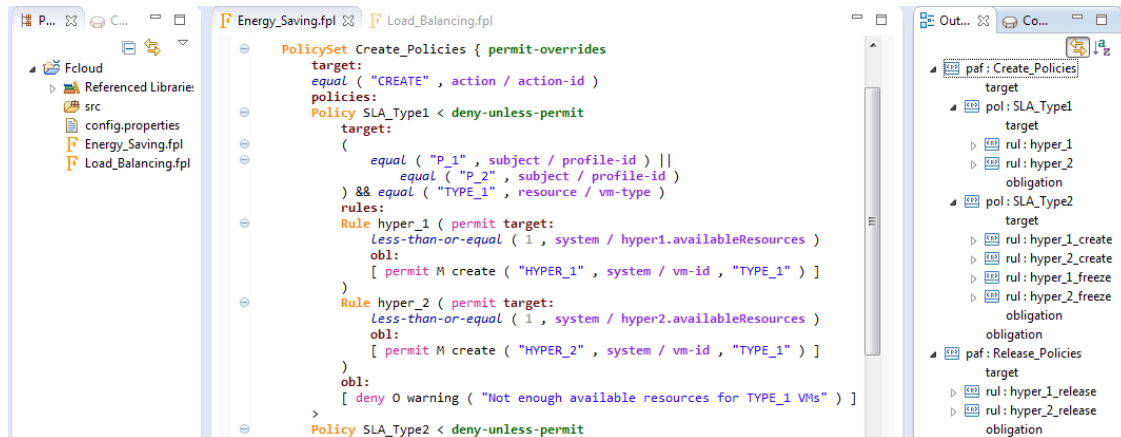


Figure 8: FACPL Eclipse IDE

2.6.1 Progress and Integration

The FACPL library and plugin have been developed during the third year of the project. In the last year of the project, the Java evaluation environment will be integrated within the jRESP runtime environment, thus enabling a full evaluation of the policy layer when programming ensembles using SCEL. We also plan to integrate an analysis tool, currently under development, that will permit to decide if a set of policies respect some given system behavioural properties. The fully integrated FACPL plugin will be added in the SDE.

2.6.2 Installation and Usage

The FACPL language has a dedicated web site at [PTMM13], which provides full information on the installation process and on the usage of the supporting software tools. In short, the plugin can be installed within Eclipse by adding the update site <http://rap.dsi.unifi.it/facpl/eclipse/plugin>. The installation wizard adds automatically the Xtext framework dependencies and the FACPL evaluation library needed for requests evaluation. The binaries and source code of the library can be also manually downloaded from the FACPL web site.

Detailed installation and usage instructions can be found in the FACPL user guide [MMPT13b]. By means of simple examples, the guide introduces policies and requests syntax and explains how the request evaluation process is performed. The guide also illustrates the design principles at the basis of the implementation of the evaluation library, and the supporting features provided by the IDE.

2.7 KnowLang Toolset

The KnowLang Toolset is a comprehensive environment that delivers tools for creating and reasoning with the KnowLang notation – a suite of editors, parsers, compilers and checkers. The KnowLang knowledge representation (KR) can be written using either text editing tools or visual modeling tools, and then checked for syntactic integrity and model consistency.

The KnowLang Toolset organizes its tools in five distinct components (or modules), outlined in Figure 9. These are the KnowLang Editor (which combines both the Text Editor and the Visual Editor), the Grammar Compiler, the KnowLang Parser, the Consistency Checker and the Knowledge Base (KB) Compiler. These components are linked together to form a special *Know Lang Specification Processor* that checks and compiles the KR models specified in KnowLang into a KnowLang Binary. As the output of the KnowLang Toolset, the KnowLang Binary is a compiled form of the specified KB

which the KnowLang Reasoner (a distinct KnowLang component to be integrated within the system that uses KR) operates upon.

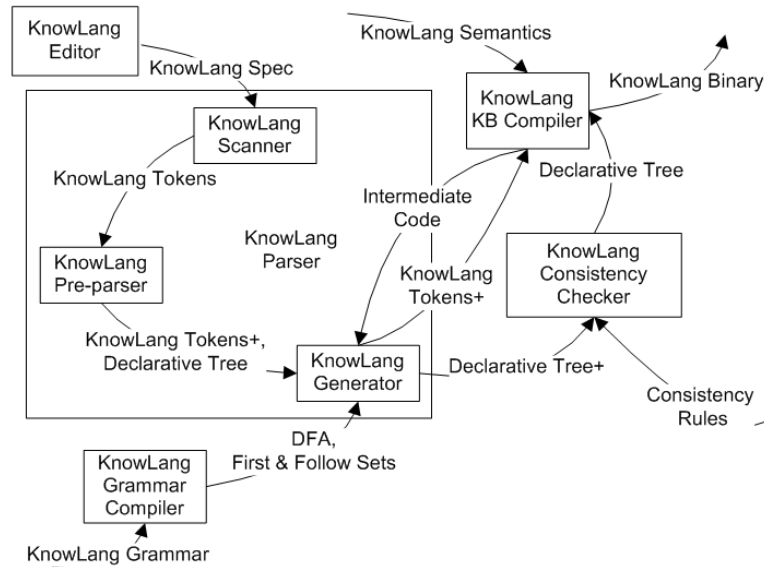


Figure 9: KnowLang Specification Processor

Figure 9 presents an abstract view where the KnowLang Toolset operation is broken down into the *data source group* (KnowLang Editor + KnowLang Grammar Compiler), which prepares the input data (grammar and specification), the *analysis group* (KnowLang Parser + Consistency Checker), which performs the lexical analysis, syntax analysis and semantic analysis, and the *synthesis group* (KnowLang KB Compiler), which is responsible for generating output. Deliverable D3.3 can be consulted for more technical details about the KnowLang Toolset.

2.7.1 Progress and Integration

At the current stage of the project, we have fully implemented the KnowLang Text Editor, Grammar Compiler and Parser. Work is progressing on the implementation of the Visual Editor, KB Compiler and Consistency Checker.

In the course of the third year, work focused on the KnowLang Framework implementation and on improving the efficiency in knowledge representation in KnowLang. We have also started implementing the KnowLang Reasoner, where the main efforts were on the implementation of the ASK and TELL operators along with the awareness control loop. Currently, we are implementing the operational semantics of these operators. As for the awareness control loop, we are using a super loop architecture that helps us realize different levels of awareness exhibition and eventually a degree of awareness. Basically, this architecture introduces a deadline to each of the awareness loop tasks and the levels of awareness are a product of the different number of loop iterations.

Our plans for the fourth year are mainly concerned with further development of the KnowLang Reasoner. This will be complemented by the implementation of the awareness prototypes based on the new knowledge representation models for the ASCENS case studies, developed with the ARE approach as described in deliverable D3.3.

2.7.2 Installation and Usage

At this moment, a packaged version of the KnowLang Toolset is not available. It is being prepared as a part of the toolset development process.

2.8 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.

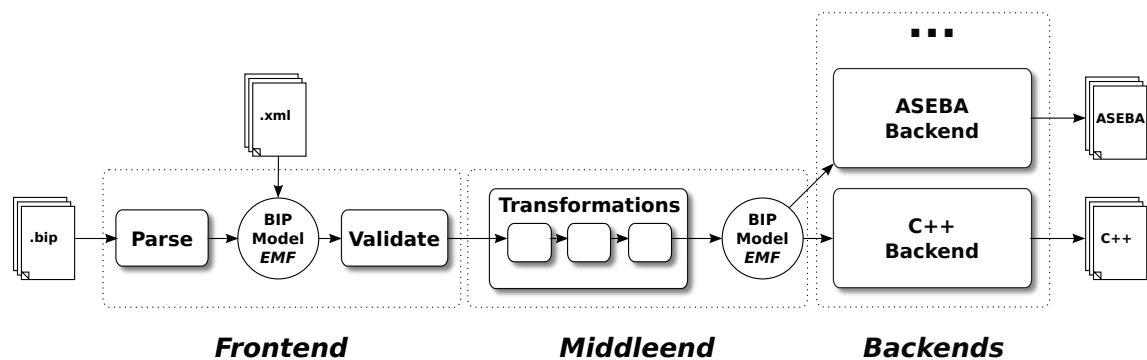


Figure 10: The BIP Compiler tool-chain.

The BIP framework is supported by a tool-chain including model-to-model transformations and code generators (see Figure 10).

2.8.1 Progress and Integration

The BIP compiler and the core BIP tools have been rewritten in the second year of the ASCENS project. The BIP compiler is organized in Java packages in a modular way, allowing the dynamic invocation of model-to-model transformers and backends. 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.

2.8.2 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 the binaries needed for executing the tool. 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. More tool details and tool examples are available on the same page, a detailed BIP documentation is available at <http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/index.html>.

2.9 Gimple Model Checker

GMC is an explicit-state code model checker for C and 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.

2.9.1 Progress and Integration

Work on GMC in the third project year included extensions for the C++ language features, and support for custom listeners – the support for C++ has been almost finished. Now the tool supports classes including all types of C++ inheritance, plus exceptions – currently there is a limitation that the type of the exception being thrown can only be a primitive type. 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 bug fixes and code optimizations have been implemented also in the C-part of the GMC tool.

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. In the final project year, we plan to include support for templates and finish integration of the tool into SDE.

2.9.2 Installation and Usage

The source code of GMC is available from http://d3s.mff.cuni.cz/projects/formal_methods/gmc/. 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. The integrated model checker tests provide the basic usage examples.

3 Runtime Cycle Tools

3.1 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).

3.1.1 Progress and Integration

In the course of the third year, we have developed a new major version of ARGoS (version 3), publishing 15 beta releases. The development of ARGoS 3 has been a major effort to improve on the already successful previous version. The main goals in this direction were (i) support for adding custom robot types, (ii) integrating ARGoS with other tools, and (iii) generally improving ARGoS as the current state-of-the-art physics-based multi-robot simulator.

To address goal (i), we redesigned the architecture of ARGoS from scratch. The final design, based on advanced concepts from C++ templates, allows users to extend any aspect of ARGoS without touching its core. In addition, we rewrote the compilation configuration environment (based on CMake scripts) to make it easier to cross-compile control code from simulation to real robots.

For goal (ii), we improved the ARGoS API both in terms of structure and naming, and in terms of documentation. Most importantly, with ARGoS 3 it is now possible to code robot behaviors also with the Lua scripting language, besides the traditional C++ approach. Further integration activities are ongoing:

- A code draft is available to interface the MultiVeStA distributed statistical analyzer [SV] with ARGoS. When this work will be finished, it will be possible to perform complex statistical analysis like distributed statistical model checking of large robot swarms automatically and with ease.
- ARGoS is being integrated with the camera-based robot tracking system installed at IRIDIA. This integration will bring our analysis capabilities to a new level. In fact, we will be able to apply complex performance measures to real-robot experiments with the same ease as in simulated experiments.
- ARGoS 2 was integrated with the well-known network simulator ns3². This work will be ported to ARGoS 3, allowing for hybrid simulations involving both the physics and the communication dynamics of robot swarms. No other simulator is currently providing this feature.

²<http://www.nsnam.org>

For goal (iii), we have profiled the **ARGoS** code closely, identified performance bottlenecks, and found solutions to solve them. As a result, **ARGoS 3** significantly improves in performance. **ARGoS 3** is more efficient both in terms of memory usage and in terms of experiment run-times.

Finally, we are introducing a new website. In this new version of the website, we will devote space not only to **ARGoS** itself, but also to the content (code, extensions, robot behaviors) contributed by the community around **ARGoS**. The aim of this work is to consolidate the role of **ARGoS** as the main development tool in swarm robotics.

3.1.2 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 `argos3`. 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>.

3.2 jDEECo: Java runtime environment for DEECo applications

jDEECo is a Java-based implementation of the **DEECo** component model [BGH⁺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 local version of **jDEECo** also supports verification of application properties using Java PathFinder, as detailed in the deliverable D5.3.

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⁺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).

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 11. 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.

3.2.1 Progress and Integration

In the third year, we have focused primarily on maturing the jDEECo implementation and developing the case studies on top of jDEECo. The development of case studies validated the jDEECo concepts and language mapping. The experience with the case studies (in particular with the cloud case study) provided new directions of improvement. In this respect, we have proposed and implemented an extension of jDEECo which increases the expressive power of ensemble specification by allowing to restrict membership to n -best members.

The extension gives users the ability to create memberships based on an arbitrary quality attribute. This attribute can be for example the connection speed between individual nodes in a cloud, or the physical distance between a car and the parking lots. Even before this extension was implemented, it was possible to create memberships based on conditions such as “choose 3 computers that have the best connectivity and put them into an ensemble” or “create an ensemble with 2 closest parking lots”. However, implementing such membership required a non-trivial amount of code that was actually very similar regardless of the scenario. Such code needed to collect the information from all possible member candidates and then evaluate which candidates are the best. Our implementation of this extension adds a new annotation – `@Selector` – that allows the user to focus solely on the “quality metric” evaluation and hides all the book-keeping code inside the framework itself.

The implementation exists as a separate project at <https://github.com/JulienFr/JDEECo>, with examples of the new annotations in the `jdeeco-demo` package, in the `cloud/scenarios` subfolder. Integration into the main jDEECo branch is underway.

3.2.2 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

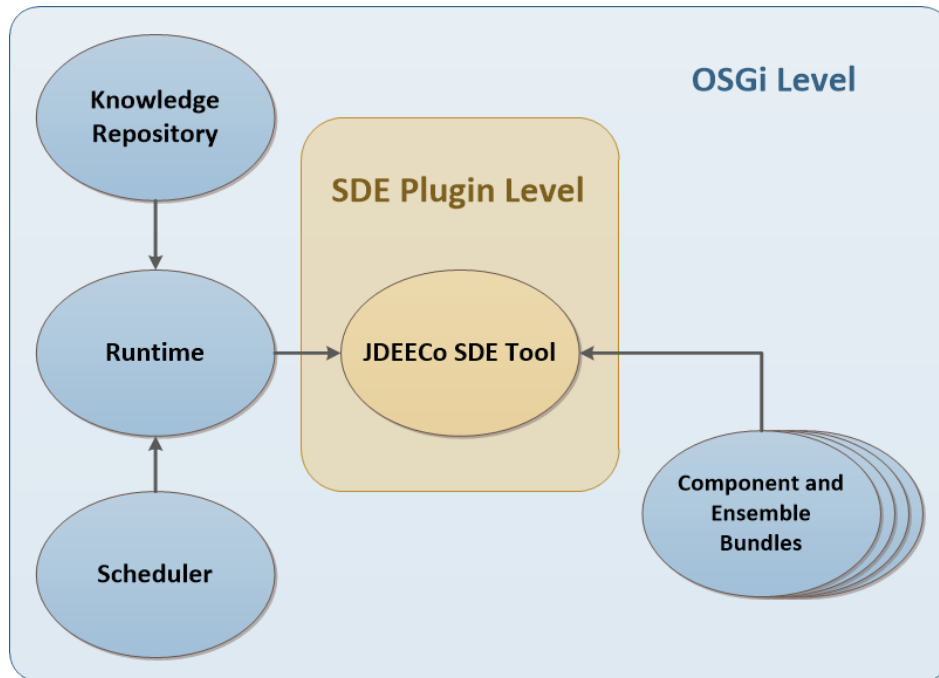


Figure 11: jDEECo SDE Tool - OSGi-SDE Integration

`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.

3.3 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, NFLP13] 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. Moreover, 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.

To support analysis of adaptive systems specified in SCEL, jRESP also provides a set of classes that permits simulating jRESP *programs*. These classes allow the execution of *virtual components* over a simulation environment that is able to control component interactions and to collect relevant simulation data.

By relying on jRESP simulation environment, a prototype framework for *statistical model-checking* has been also developed. Following this approach, a randomized algorithm is used to verify whether the implementation of a system satisfies a specific property with a certain degree of confidence. Indeed, the *statistical model-checker* is parameterized with respect to a given *tolerance* ε and *error probability* p . The used algorithm guarantees that the difference between the value computed by the algorithm and the exact one is greater than ε with a probability that is less than p .

The model-checker included in jRESP can be used to verify *reachability properties*. These permits evaluating the probability to reach, within a given deadline, a configuration where a given predicate on collected data is satisfied.

3.3.1 Progress and Integration

In the third year of the project, we have continued the development of jRESP by focusing on two main aspects: a new implementation of the SCEL group oriented communication and the integration of external reasoners for supporting adaptivity.

To provide a more efficient and reliable support to group-oriented interactions, we have included specific classes that realize these interactions in terms of the *P2P* and *multicast* protocols provided by Scribe [CDKR03], a generic, scalable and efficient system for group communication and notification, and FreePastry [RD01a], a Scribe substrate for peer-to-peer applications.

Moreover, to support the integration of external reasoners, the internal knowledge-handling mechanisms have been rearranged. Processes executed at a given component can now transparently interact with external reasoners.

In the next year, we plan to design a high-level programming language (HL-SCEL), that, by enriching SCEL with standard programming constructs (e.g. control flow constructs such as while or if-then-else or structured data types), simplifies the programming tasks. We also develop an SDK that will provide a compiler which will generate jRESP code from a HL-SCEL program. This SDK will be integrated in the SDE.

3.3.2 Installation and Usage

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

3.4 CIAO Environment

CIAO [BCC⁺97] is a multiparadigm programming language developed by an international team for over 20 years. CIAO offers an advanced programming environment that supports the ISO-Prolog standard, however, its modular design allows both restricting and extending the basic language through libraries. Complete information about CIAO can be found at <http://www.ciao-lang.org>, together with complete documentation.

Within ASCENS, we use CIAO to implement the Soft Constraint Logic Programming Framework [BMR97] to naturally model and solve the e-Mobility optimization problems. Besides lo-

cal travel optimization problems of individual users, such as trips and journeys [MMH12], we propose to use CIAO also to solve global optimization problems involving large ensembles of vehicles [BDG⁺13].

In both cases, we need to implement the SCLP framework explicitly by means of predicates. In particular, in order to implement the soft framework, we need to implement the semiring modeling the levels of satisfiability or the costs of the constraints. To this end, we define two predicates respectively modeling the additive and the multiplicative operations of the chosen semiring.

3.4.1 Progress and Integration

So far, our use of CIAO has focused on experimental prototypes. To tackle the global optimisation problems encountered in [BDG⁺13], we have proposed a framework based on the coordination of declarative and procedural knowledge. Our demo application in deliverable D7.3 illustrates how the parking optimization problem (consisting of finding the best parking lot for an ensemble of vehicles) can be solved using CIAO. The global problem is decomposed into several local problems solved by CIAO programs, whose parameters are iteratively determined by a Java coordinator in order to obtain an acceptable global solution.

Our eventual aim is to provide a general way to embed the soft framework in CIAO, using for example a library offering a more general implementation of the operations of several semirings, or – more interestingly – a meta-level implementing efficiently this framework.

Future plans also include providing a specialized component for the Java coordinator, or, possibly, an implementation of such a component in another procedural language with specific knowledge representation features, like OZ, to make the interaction with the SCLP component more direct.

3.4.2 Installation and Usage

The latest stable version of CIAO can be downloaded from http://ciao-lang.org/download_stable.html. However, to implement our demo application, where we need to interface CIAO with Java, we have used the latest development version that can be downloaded from http://ciao-lang.org/download_latest.html. In this version, some problems we found during the development of the demo were fixed (an official stable release should be available soon).

3.5 Science Cloud Platform

The Science Cloud Platform (SCP) is the software system developed as part of the science cloud case study of ASCENS. The SCP is a platform-as-a-service cloud computing infrastructure which enables users to run applications while each individual node of the cloud is voluntarily provided (i.e., may come and go), data is stored redundantly, and applications are moved according to current load and availability of server resources.

As such, the SCP serves as the main technical demonstrator for the science cloud case study of ASCENS.

3.5.1 Progress and Integration

During year 3, the specification of the science cloud platform as well as our first prototype have been used to drive an updated implementation, which is based on existing state-of-the-art network algorithms and protocols to create an OSGi-based hybrid cloud platform which combines, as detailed in deliverable [vHP⁺13], the domains of voluntary, peer-to-peer, and cloud computing.

Lessons learned from the first implementation presented last year have been used in the new implementation, specially the use of OSGi and its ability to dynamically install and use application code. The entire network layer, however, has been swapped out; we now use the peer-to-peer substrate Pastry [RD01a] and accompanying protocols for the communication and data layers, which includes the DHT Past [RD01b] and the publish/subscribe mechanism Scribe [CDKR02]. On top of these layers, a variant of the ContractNET [Fou13] protocol has been used to implement application failover.

The science cloud platform will be finalized in the last year and integrated into the SDE.

3.5.2 Installation and Usage

As before, the progress of the Science Cloud Platform prototype is being tracked on <http://svn.pst.ifi.lmu.de/trac/scp>. As shown in the source view, both the original direct implementation as well as the pastry-based implementation are available for testing and runtime.

Both prototypes are built on top of Java and OSGi. The new installation contains a multi-node startup mechanism which, for testing purposes, can start many nodes on one machine. To start up this instance, it must be run with all dependencies inside an OSGi container like Equinox. The easiest way of doing this is from Eclipse itself, where a launch configuration is provided.

The UI for the started nodes is available in a web-based manner on the ports starting from 10001 (and continuing with 10002, etc.). As before the UI allows complete control over the individual SCPI and contains monitoring functionality.

To test the failover functionality, the repository also contains a demo project which implements a chat application. This project can be exported as a JAR from Eclipse and deployed via the web UI. Subsequent changes to the network — for example, by terminating the instance the app runs on — will lead to the proper reaction by the ensemble running this application.

3.6 SPL

SPL is a Java framework for implementing application adaptation based on observed or predicted application performance [BBH⁺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⁺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.

The instrumentation itself is controlled by a high-level API that allows the user to specify which parts of the application should be measured and how. The simplest approach is to measure single method duration every time the method is invoked, however, the framework also offers a tunable approach for situations where collecting duration times of single methods does not provide a detailed enough information.

The measurement granularity can be configured in several orthogonal directions. One is whether to measure the duration of a single method or the duration between invocations of different methods. This allows to measure, for example, request processing time in callback-oriented frameworks where a single request is processed in several methods, often in context of different threads. In such frameworks, there is no single method “wrapping” the whole processing pipeline.

The user can also specify custom filters to preprocess the measured data. One example of such preprocessing is when different criteria are to be applied based on data size. The SPL formulas then reflect this distinction, allowing more precise decisions to be made. The filters are inserted together with the measurement code during the instrumentation and thus can access any data structures available in the measured method, including their arguments or class fields.

The granularity can be also specified on the Java class level. It is possible to limit the instrumentation not only to certain classes, but also only to classes from certain class-loaders – a necessity in component-oriented environments such as OSGi.

The pluggable data sources described in [BBH⁺12] allow the user to combine different performance metrics across the application, possibly even integrating them in a single formula. Some data sources are provided by the framework itself – for example the method duration times obtained through the instrumentation or access to the current system load. Other sources can be provided by the user and can include wrappers to already existing performance indicators in the application (such as request-queue length) or platform-specific information such as the processor frequency.

3.6.1 Progress and Integration

The SPL framework has been mostly developed during the second year of the project, the new extensions related to the measurement granularity and the custom filters – motivated mostly by the case studies – were added in the third year of the project. The integration of the SPL tool into the SDE platform is a work in progress.

3.6.2 Installation and Usage

The latest version of the SPL framework can be obtained from <http://github.com/vhotspur/spl-java>. The source code is distributed with Apache Ant `build.xml`, which allows building the entire package and running unit tests. The framework provides a JVM agent, which can evaluate an SPL formula with modular data sources [BBH⁺12].

The framework can be used in two modes. In one, SPL acts as an external mechanism controlling the application adaptation. In the other, adaptation rules are contained in the business logic of the application.

When SPL is used as an external mechanism, the source code of the application does not need to be modified. As a matter of fact, source code is not needed at all and even the bytecode is modified

at run-time only. However, the application itself must expose interfaces for run-time configuration changes.

When SPL is incorporated into the application itself, the rules for adaptation are part of the business logic. This can provide fine-grained performance tuning, however, source code modification are necessary. This is illustrated in the example below.

An SPL demonstration example is provided together with the source code. The example shows a monitoring application that adjusts the output quality to reflect load – it draws a graph that normally contains a data point for each hour, however, under high system load only a data point for each day is used – the output is still useful but processing time is reduced. See Figure 12 for an example.

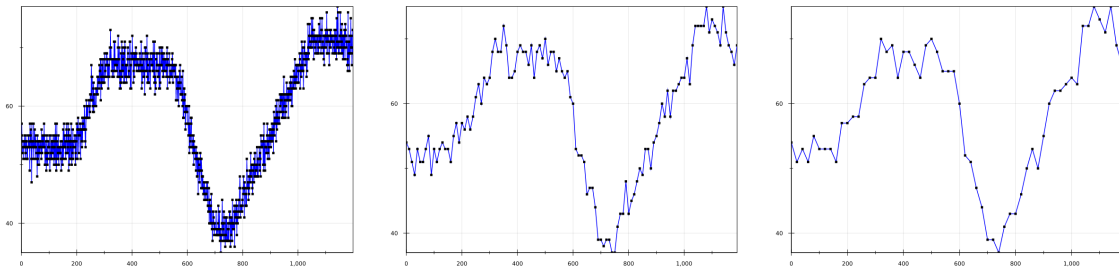


Figure 12: Graphs of different quality provided for different monitoring application load.

The demo is available in the `src/demo-java` folder, in the `imagequality` package, and can be started through the `run-demo-imagequality` Ant target of the framework build file. The demo uses the HTTP server provided by JVM to respond to requests on port 8888.

We used the Pylot performance tool³ to roughly evaluate the advantage of the performance adaptation – with no adaptation, the demo could handle 33 requests per second and 95 % of requests finished in 3 seconds, whereas with adaptation, the demo handled 44 requests per second and 95% of all requests were finished in 2 seconds. The code itself is intentionally simple, serving to illustrate the benefits of adding an external SPL adaptation to an application.

³<http://www.pylot.org>

4 Conclusion and Plans for Year Four

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, the methods and techniques for engineering ensembles have progressed enough for multiple models to be developed and applied on the case studies (Subtasks T7.x.2 and T7.x.3). The tool development reflects this and from the project management perspective is therefore generally on track.

The tool implementation and integration efforts 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, the availability of the first tool application examples made 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 connection of reasoners such as the CIAO based constraint solver or the FACPL based policy evaluation to the runtime platforms for online adaptation, or application of the SPL tool on the jDEECo, jRESP and SCP runtime platforms for performance awareness ; additional tool integration opportunities are sketched in the tool presentations.

For the concluding project year, we expect that the implementation and evaluation activities of the case studies (Subtasks T7.x.4) will bear a strong influence on the tool implementation and integration efforts – a major purpose of the case studies is to provide an experimental platform where the individual project contributions converge, and the tool integration process is a necessary part of this convergence. To prevent unexpected disruptions in concluding 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 and the vehicle mobility example in the jDEECo tool tutorial.

Work on specific tools will follow the directions specified in the research workpackages – among other items, the major plans include supporting the stochastic extensions to SCEL and supporting high-level extensions for implementing SCEL programs – and also integrating the reasoning tools more tightly with the runtime platforms. A major task of this workpackage will be the coordination of the final integration activities, where the ultimate goal is to have the developed tools introduced on a central portal within the project website, available for download and use in the integrated development environment, together with the necessary documentation. Finally, the project partners will pay attention to the potential for continuous exploitation of the prototype tools beyond the conclusion of the project.

References

- [AHZ13] D. B. Abeywickrama, N. Hoch, and F. Zambonelli. SimSOTA: Engineering and simulating feedback loops for self-adaptive systems. In *Proceedings of the 6th International C* Conference on Computer Science & Software Engineering (C3S2E'13) (In Press)*. ACM, 2013.
- [AL] Inc. AT&T Labs. Graphviz - Graph Visualization Software. <http://www.graphviz.org>.
- [AM11] Musab AlTurki and José Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In *CALCO*, pages 386–392, 2011.
- [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.
- [AZH12] D. B. Abeywickrama, F. Zambonelli, and N. Hoch. Towards simulating architectural patterns for self-aware and self-adaptive systems. In *Proceedings of the 2nd Awareness Workshop co-located with the SASO'12 Conference*, pages 133–138. IEEE, 2012.
- [BBH⁺12] Lubomir Bulej, Tomas Bures, Vojtech Horky, Jaroslav Keznikl, and Petr Tuma. Performance Awareness in Component Systems: Vision Paper. COMPSAC '12, 2012.
- [BCC⁺97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), 1997.
- [BCG⁺12a] 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.
- [BCG⁺12b] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Adaptable transition systems. In Narciso Martí-Oliet and Miguel Palomino, editors, *WADT*, volume 7841 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2012.
- [BCG⁺12c] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. A conceptual framework for adaptation. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2012.
- [BCG⁺12d] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with maude. In Franciso Durán, editor, *Rewriting Logic and Its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 118–138. Springer Berlin Heidelberg, 2012.
- [BDG⁺13] Tomas Bures, Rocco De Nicola, Ilias Gerostathopoulos, Nicklas Hoch, Michal Kit, Nora Koch, Valentina Monreale, Ugo Montanari, Rosario Pugliese, Nikola Serbedzija, Martin Wirsing, and Franco Zambonelli. A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase. 2013. submitted.

- [BDVW] Lenz Belzner, Rocco De Nicola, Andea Vandin, and Martin Wirsing. Reasoning (on) Service Component Ensembles in Rewriting Logic. To appear in the proceedings of SAS 2014, Springer LNCS Festschrift.
- [BGH⁺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.
- [BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Satisfaction and Optimization. *J. ACM*, 44(2):201–236, 1997.
- [CDE⁺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.
- [CDKR02] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.
- [CDKR03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. Scalable Application-Level Anycast for Highly Dynamic Groups. In *ICQT*, LNCS 2816, pages 47–57. Springer, 2003.
- [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/>.
- [DKL⁺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⁺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.
- [Fou13] Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification. <http://www.fipa.org/specs/fipa00029/SC00029H.html>, March 2013.
- [HGB10] R. Hebig, H. Giese, and B. Becker. Making control loops explicit when architecting self-adaptive systems. In *Proc. of the 2nd International Workshop on Self-Organizing Architectures*, pages 21–28. ACM, 2010.

- [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.
- [LNGE11] M. Luckey, B. Nagel, C. Gerth, and G. Engels. Adapt cases: extending use cases for adaptive systems. In *Proceedings of the 6th International SEAMS Symposium*, pages 30–39. ACM, 2011.
- [ML] MESSI. System Modelling and Analysis @ IMT Lucca. Maude ensemble strategies simulator and inquirer. <http://sysma.lab.imtlucca.it/tools/ensembles/>.
- [MMH12] G. V. Monreale, U. Montanari, and N. Hoch. Soft Constraint Logic Programming for Electric Vehicle Travel Optimization. *CoRR*, abs/1212.2056, 2012.
- [MMPT13a] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. A Formal Software Engineering Approach to Policy-based Access Control. Technical report, DiSIA, Univ. Firenze, 2013. <http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf>.
- [MMPT13b] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Formal Access Control Policy Language (FACPL) User’s Guide, 2013. <http://rap.dsi.unifi.it/facpl/guide/FACPL-guide.pdf>.
- [MPS08] H. Muller, M. Pezze, and M. Shaw. Visibility of control in adaptive systems. In *Proceedings of the 2nd International Workshop on Ultra-large-scale Software-intensive Systems*, pages 23–26. ACM, 2008.
- [MVZ⁺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⁺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.
- [NFLP13] Rocco Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In Bernhard Beckert, Ferruccio Damiani, FrankS. Boer, and MarcelloM. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer Berlin Heidelberg, 2013.
- [OGCD10] Rehan O’Grady, Roderich Groß, Anders Lyhne Christensen, and Marco Dorigo. Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots*, 28(4):439–455, 2010.

- [Ora12] Oracle. `java.lang.instrument` (Java Platform, Standard Edition 6, API Specification), 2012. <http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html>.
- [PTMM13] Rosario Pugliese, Francesco Tiezzi, Massimiliano Masi, and Andrea Margheri. Formal Access Control Policy Language (FACPL), 2013. <http://rap.dsi.unifi.it/facpl/>.
- [QS10] Paola Quaglia and Stefano Schivo. Approximate model checking of stochastic cows. In *Proceedings of the 5th international conference on Trustworthy global computing, TGC'10*, pages 335–347, Berlin, Heidelberg, 2010. Springer-Verlag.
- [RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Middleware '01*, pages 329–350, London, UK, UK, 2001. Springer-Verlag.
- [RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 188–201. ACM, 2001.
- [RHR11] P. Van Roy, S. Haridi, and A. Reinefeld. Designing robust and adaptive distributed systems with weakly interacting feedback structures. Technical report, ICTEAM Institute, Catholic University Louvain, 2011.
- [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.
- [SV] Stefano Sebastio and Andea Vandin. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. Submitted. <http://eprints.imtlucca.it/1798>.
- [SVA05] Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In Christel Baier, Giovanni Chiola, and Evgenia Smirni, editors, *QEST 2005*, pages 251–252. IEEE Computer Society, 2005.
- [VG12] T. Vogel and H. Giese. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In *Proceedings of the 7th International SEAMS Symposium*, pages 129–138. IEEE/ACM, 2012.
- [vHP⁺13] Nikola Šerbedžija, Niklas Hoch, Carlo Pinceroli, Michal Kit, Tomas Bures, Giacomo Valentini Monreakle, Ugo Montanari, Philip Mayer, and Jose Velasco. D7.3: Third Report on WP7: Integration and Simulation Report for the ASCENS Case Studies. ASCENS Deliverable, November 2013.
- [VWMA11] P. Vromant, D. Weyns, S. Malek, and J. Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th SEAMS Symposium*, pages 202–207, 2011.
- [WH07] T. De Wolf and T. Holvoet. Using UML 2 activity diagrams to design information flows and feedback-loops in self-organising emergent systems. In T. De Wolf, F. Saffre, and R. Anthony, editors, *Proceedings of the 2nd International Workshop on Engineering Emergence in Decentralised Autonomic Systems*, pages 52–61, 2007.