www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

JD4.2: ASCENS Tool Suite

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, M. Kit (CUNI), A. Lluch Lafuente (IMT), M. Loreti, A. Margheri (UDF), P. Mayer (LMU), V. Monreale, U. Montanari (UNIPI), C. Pinciroli (ULB), P. Tůma (CUNI), A. Vandin (IMT), E. Vassev (UL)**

Reporting Period: **4**
Period covered: **October 1, 2010 to January 31, 2015**
Submission date: **March 8, 2015**
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**

SEVENTH FRAMEWORK PROGRAMME

Executive Summary

This text and the tools listed within constitute the ASCENS project joint deliverable JD4.2 – a complete release of the tools developed and integrated with the ASCENS project. Tool development and integration is the focus of workpackage WP6, where the annual progress has been reported in deliverables D6.1, D6.2, D6.3 and D6.4.

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 from work package WP6. Where required, the tool descriptions from these deliverables are reproduced with updates to reflect the current project status.

Contents

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.

The progress of workpackage WP6 is reported in annual deliverables. The deliverable D6.1 collected the tool integration requirements. The deliverables D6.2 to D6.4 presented gradual tool releases. The goal of this deliverable, the joint deliverable JD4.2, is to provide a complete overview of the tools.

The goal of the tool release is to maximize the practical outreach beyond project scope – hence, effort has been made to have all tools as much self describing as possible, with the accompanying documentation in the usually preferable form of online help, examples and tutorials. The textual deliverables reference the online releases and inform about project progress, however, they are not meant to supplant the tool documentation.

1.1 Integration Environment

The integration of the project tools is made easier by the fact that, where applicable, the tools share the common notions of components and ensembles as introduced by SCEL. Although different tools extend these concepts with emphasis on different aspects, the existence of common conceptual foundations is obviously useful.

The integration challenge lies on the technological level, where the tools developed by the individual workpackages are rather diverse. The diversity is necessitated simply by adopting appropriate technologies for the task at hand – some tools require environments for efficient logical reasoning such as Maude or Prolog, other tools focus on bytecode manipulation and classloading particular to Java, and yet other tools rely on C libraries for physics simulation or generate C code for efficient model execution.

We tackle the technological diversity by supporting common platforms as much as is practical. In particular, our Java development is integrated with the Eclipse platform, with selected tools packaged as Eclipse plugins or wrapped as OSGi bundles. Tool orchestration is is provided through the Service Development Environment (SDE), an orchestration extension to Eclipse that has originated in the FP6 SENSORIA project, now used in the FP7 ASCENS and FP7 NESSOS projects. Data exchange is also facilitated by relying on standard modeling features, such as Ecore and Xtext.

1.2 Current Tool Landscape

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

Following the ASCENS approach to the software development lifecycle, we roughly classify the tools into design tools and runtime tools. This distinction is not strict – especially the difference between monitoring a faithful simulation at design time and monitoring a real execution at runtime can be small – but it is useful to provide a presentation structure.

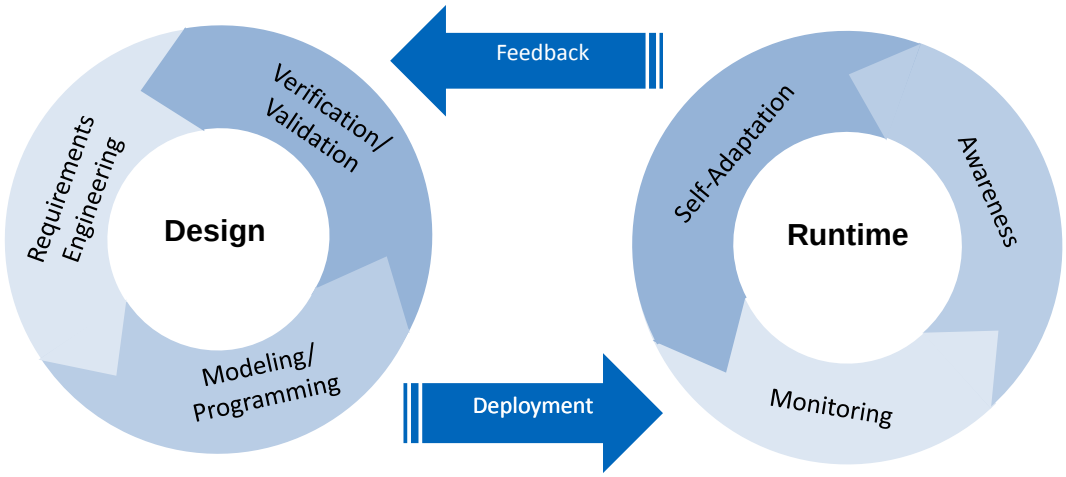

Figure 1: ASCENS Ensemble Software Development Life Cycle

On the design cycle side, our tool support starts with the early stage formal modeling tools. These tools are 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 – three tools that rely on Maude, MAIA (Section 2.5), MESSI (Section 2.3) and MISSCEL (Section 2.4), have also been developed. The SimSOTA tool (Section 2.6) can evaluate the behavior of complex feedback driven adaptation mechanisms using simulation. The FACPL framework (Section 2.7) can be used to capture policies that regulate interaction and adaptation of SCEL components. The KnowLang Toolset (Section 2.8) 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. These include the BIP compiler (Section 2.9) for the approaches that rely on correctness by construction. For manual implementation, we provide frameworks that reify the formal modeling concepts, specifically jRESP (Section 3.2) and jDEECo (Section 3.3) – 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 examine methods and tools to verify whether code complies to models. For C code, we have developed the GMC model checker (Section 2.10), for Java code, we have integrated the JPF model checker in jDEECo (Section 3.3).

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 within 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 have developed two complementary strategies for runtime support. Where possible, such as in the scientific cloud, we simply use live ensemble introspection. Where not possible, such as in the robotic swarms, we introspect ensemble simulations.

Our simulation environment for the robotic swarms is ARGoS (Section 3.1). This simulation environment provides built in observation and introspection capabilities. For the cloud case study, we have similarly developed the Science Cloud Platform (Section 3.6). Two generic runtime environments for ensemble prototypes are jRESP (Section 3.2) and jDEECo (Section 3.3).

Visualisation support for ensemble structure and component state is provided by AVis (Section 3.4). Additional ensemble introspection capabilities rely 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 3.7) is used to reason about performance.

1.3 Connections To 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. On the thematic side, we list the collaboration areas per workpackage.

- 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 SCEL language refinements. The runtime environments for ensembles based on SCEL 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 whose architecture includes editing tools, parsers and checkers, and a knowledge reasoner.
- 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.

1.4 Tool Presentation Overview

The next sections contain a brief description of each of the tools following a unified outline, where the purpose of the tool is briefly outlined and compact installation and usage directions follow.

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.

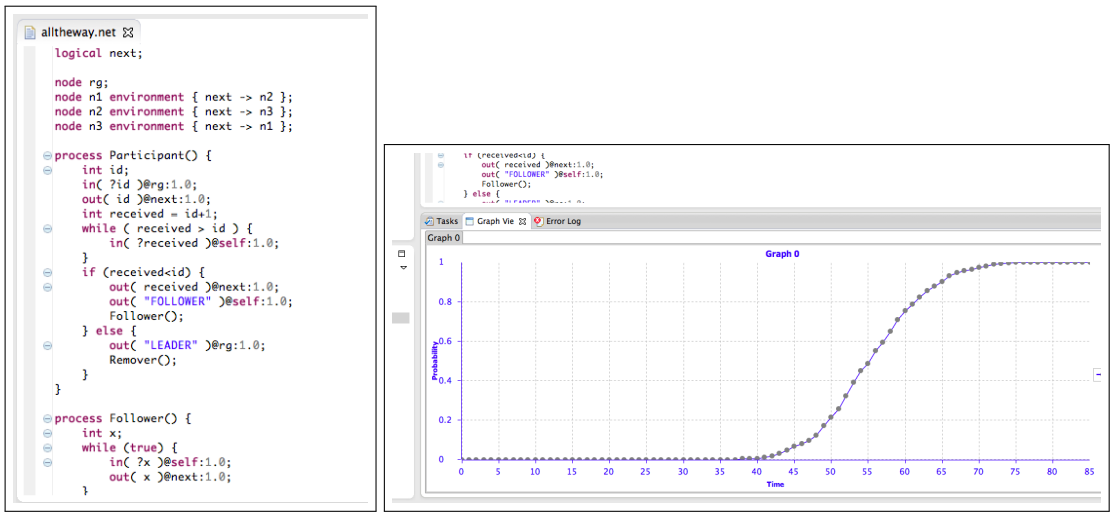

Figure 2: A jSAM specification (left) and the result of model-checking (right).

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).

On-the-fly model checking. Model checking approaches can be divided into two broad categories: global approaches that determine the set of all states in a model $\mathcal{M}$ that satisfy a temporal logic formula Φ, and local approaches in which, given a state s in $\mathcal{M}$, the procedure determines whether s satisfies Φ. When s is a term of a process language, the model-checking procedure can be executed "on-the-fly", driven by the syntactical structure of s. For certain classes of systems, e.g. those composed of many

parallel components, the local approach is preferable because, depending on the specific property, it may be sufficient to generate and inspect only a relatively small part of the state space. In [LLM14] an efficient, on-the-fly, PCTL model checking procedure that is parametric with respect to the semantic interpretation of the language has been proposed. The proposed model checking algorithm has been integrated in jSAM together with a new module for supporting specification and analysis of systems via the PRISM language.

FlyFast model checker. Typical self-organising collective systems consist of a large number of interacting objects that coordinate their activities in a decentralised and often implicit way. Design of such systems is challenging and requires suitable, scalable analysis tools to check properties of proposed system designs before they are put into operation. The exploitation of mean field approximation in model-checking techniques seems a promising approach to overcome scalability issues raised by the size of such collective systems. In [LLM13a, LLM13b] we have presented a novel scalable, on-the-fly model-checking procedure to verify bounded PCTL properties of selected individuals in the context of very large systems of independent interacting objects. The proposed procedure combines on-the-fly model checking techniques with deterministic mean-field approximation in discrete time. A prototype implementation of the model-checker, named FlyFast, has been integrated into jSAM and used to verify properties of a selection of simple and more elaborate case studies.

SCEL SDK and HL-SCEL. To support design, analysis and deployment of autonomous and adaptive systems developed in SCEL, we have integrated in jSAM a plug-in that, by relying on jRESP simulation environment, enables the use of (some of) the formal tools available in our framework. The proposed plug-in, named SCEL SDK, takes as input HL-SCEL specifications and automatically generates the Java classes used to simulate and execute the considered system.

Installation and Usage

jSAM can be downloaded from `http://j-sam.sourceforge.net`, where both the Java binaries and the source code are available. Detailed instructions and examples are available from the same site. jSAM is also available as an Eclipse plug-in that can be installed via the update site `http://j-sam.sourceforge.net/plugin`. In this case, the installation wizard automatically checks and installs the needed dependencies.

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+13, 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

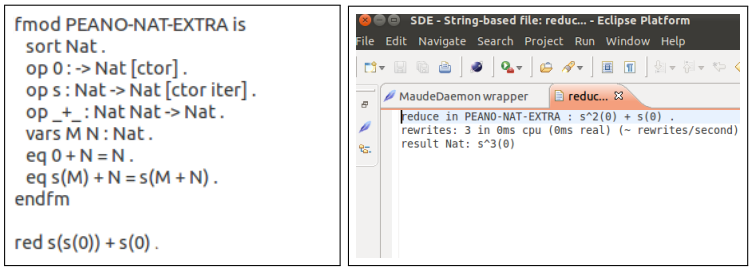

Figure 3: A Maude command (left) and its evaluation (right).

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.

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 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`. 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. 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).

A detailed description of Maude and its commands is available in the Maude manual at `http://maude.cs.uiuc.edu/maude2-manual`.

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+13, 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+13, MLb] 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.

Installation and Usage

MESSI can be downloaded from its website [MLb], 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) statisti-

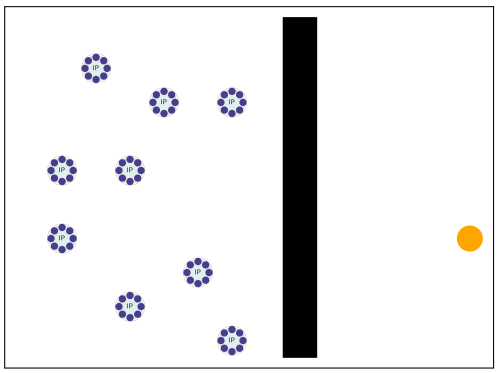

Figure 4: A pictorial representation of an initial configuration for MESSI.

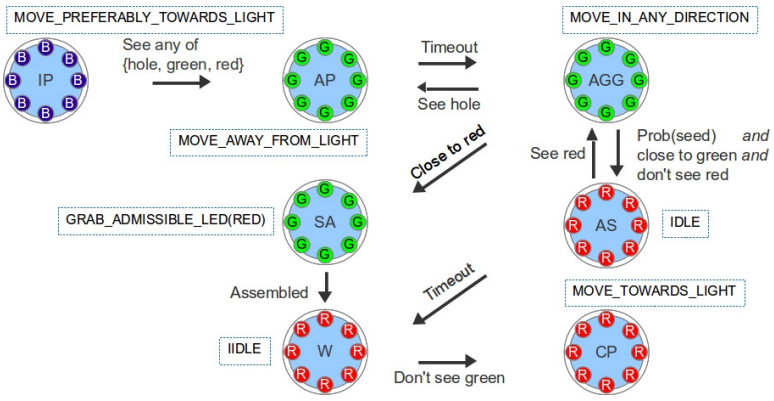

Figure 5: A pictorial representation of a self-assembly strategy for MESSI.

cal 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.

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 MAIA

As part of a research line pursued in collaboration between project partners, we presented an essential model of *adaptable transition systems* [BCG$^+$12b] inspired by white-box approaches to adaptation [BCG$^+$12c] and based on foundational models of component based systems [dAH01, dA03]. The key feature of adaptable transition systems are control propositions, a subset of the atomic propositions

labelling the states of our transtion systems, imposing a clear separation between ordinary, functional behaviours and adaptive ones. Interestingly, control propositions can be exploited in the specification and analysis of adaptive systems, focusing on various notions proposed in the literature, like adaptability, control loops, and control synthesis. We instantiated our approach on Interface Automata (IA) [dAH01, dA03], yielding Adaptable Interface Automata (AIA) [BCG+12b].

MAIA is an implementation of AIAs in Maude, allowing one to specify AIAs, to draw them, and to perform operations on them such as product, composition, decomposition and control synthesis.

Installation and Usage

MAIA can be downloaded from its website [MLa], where the usage description is also provided.

MAIA takes in input a specification of an AIA, provided as a Maude term. MAIA is invoked as follows:

```
./ATS.sh draw AN-AIA NAME-OF-THE-IMAGE
```

where `AN-AIA` is a Maude term defined in the Maude file ats.maude, which represents an AIA or an expression involving AIAs (e.g., composition, decomposition in controller and controlled systems). While NAME-OF-THE-IMAGE is the name of the file where the AIA will be drawn.

For example, if we want to draw the predefined AIA `Exe` in the file `exe`, we have to type:

```
./ATS.sh draw Exe exe
```

If we want to compose three AIAs `Mac`, `Que` and `Exe`, we have to type:

```
./ATS.sh draw 'composition(Mac,composition(Exe,Que))' MaxIExeIQue
```

Finally, if we want to decompose an AIA `S` so to obtain a manager component (a controller) and a base controller as described in [BCG+12b], we have to type:

```
./ATS.sh draw 'W(nonTrivialDecomposition(S,("u","d"),("u","d")))' W
```

with `W` being either `manager` or `base`, in order to draw the obtained controller or base component, respectively.

2.6 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 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. 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.

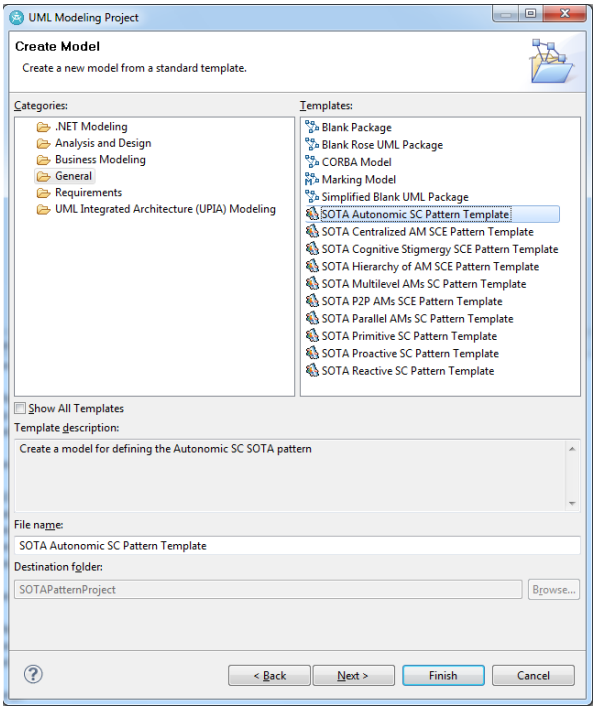

Figure 6: SOTA pattern templates available to facilitate modeling.

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.

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.

2.7 FACPL: Policy IDE and Evaluation Library

FACPL [MMPT13a] is a policy language for writing policies and requests. It has a mathematically defined semantics and can be used to regulate interaction and adaptation of SCEL 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

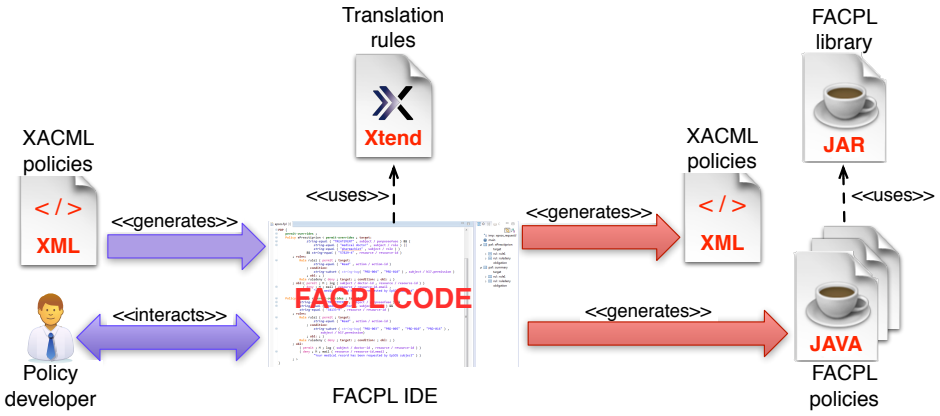

Figure 7: FACPL Toolchain

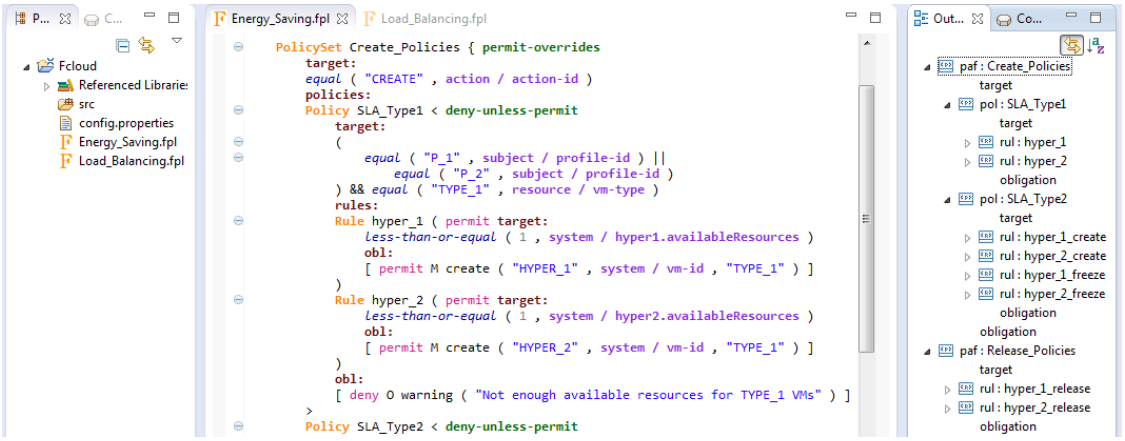

Figure 8: FACPL Eclipse IDE

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. As shown in the left-hand side of Figure 7, by using the IDE, FACPL code can be also automatically created starting from policies and requests written in XACML[1] 3.0 syntax. Moreover, by exploiting some translation rules developed using the Xtend language, the IDE can also generate the low-level XML code (compliant with the XACML 3.0 syntax) corresponding to any given FACPL code. The translation from and to XACML ensures a two-ways connection of our FACPL toolchain with external tools supporting XACML.

[1]XACML is a wide-used standard for access control systems.

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.

2.8 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.

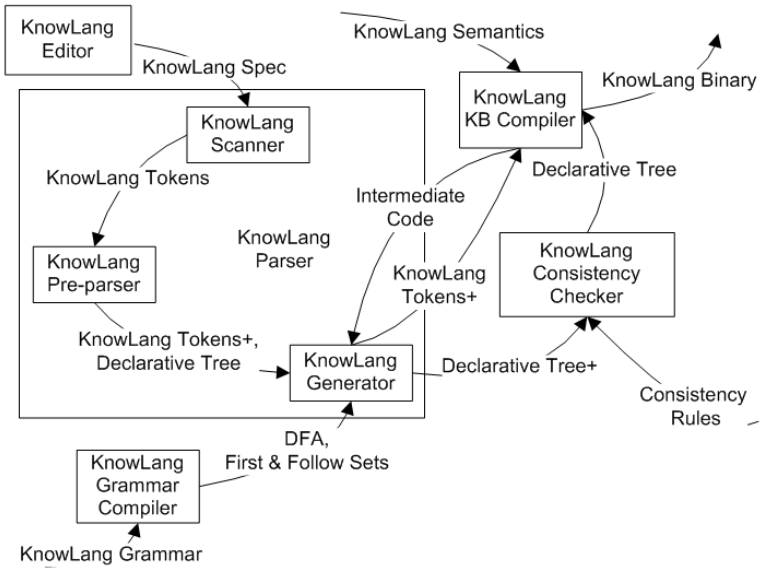

Figure 9: KnowLang Specification Processor

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.

Installation and Usage

The KnowLang Toolset is hosted at `http://knowlang.lero.ie`, where the development snapshot is available for download together with additional material.

2.9 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.

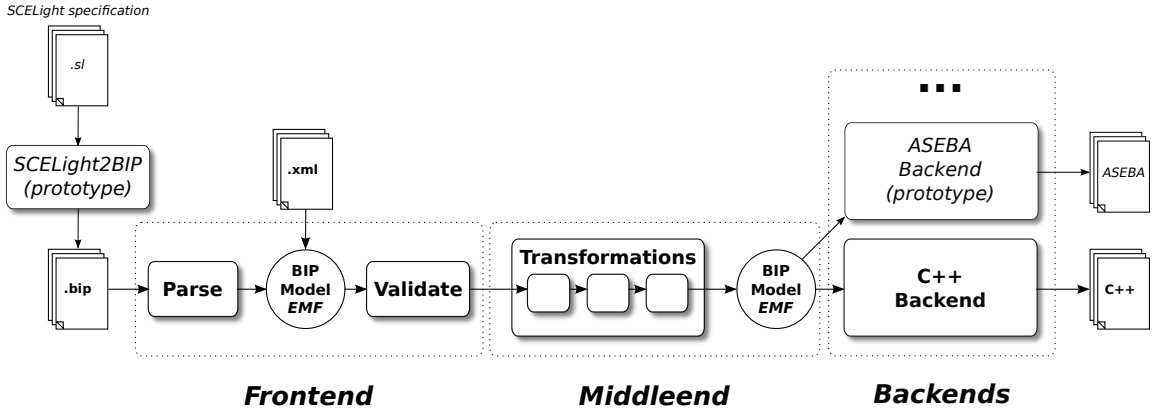

Figure 10: The BIP Compiler tool-chain.

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

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`.

2.10 Gimple Model Checker

Gimple Model Checker (GMC) is an explicit-state code model checker for C and C++ programs. This means that it can reveal errors manifesting themselves just in particular (usually rare) interleavings which are hard to find via testing. GMC supports multi-threaded programs 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 assertion statements in the code).

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 [pth], but also offers means to add support for other thread libraries based on the same principles.

Similarly to other explicit model checkers, GMC requires that the actions (steps) of the verified program are revertible, which is not always the case (for example if 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 [gcc], 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. GMC is fully integrated into SDE [sde], a development environment based on Eclipse.

Installation and Usage

Prerequisites The source code of GMC is available from [gmca]. 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.

In order to use GMC from the SDE, an GMC extension has to be installed into SDE. Follow the instruction for installation of SDE and then install the GMC extension. The update site for the extension is to be found at [gmcb]. Once the GMC extension is installed, it needs to be configured; see Fig. 11. Go to Windows → Preferences and at the GMC page the path to the patched GCC used to create gimplexx files as well as th path to the GMC model checker have to be specified. Note that you can use the GMC version built-in into the installed plugin.

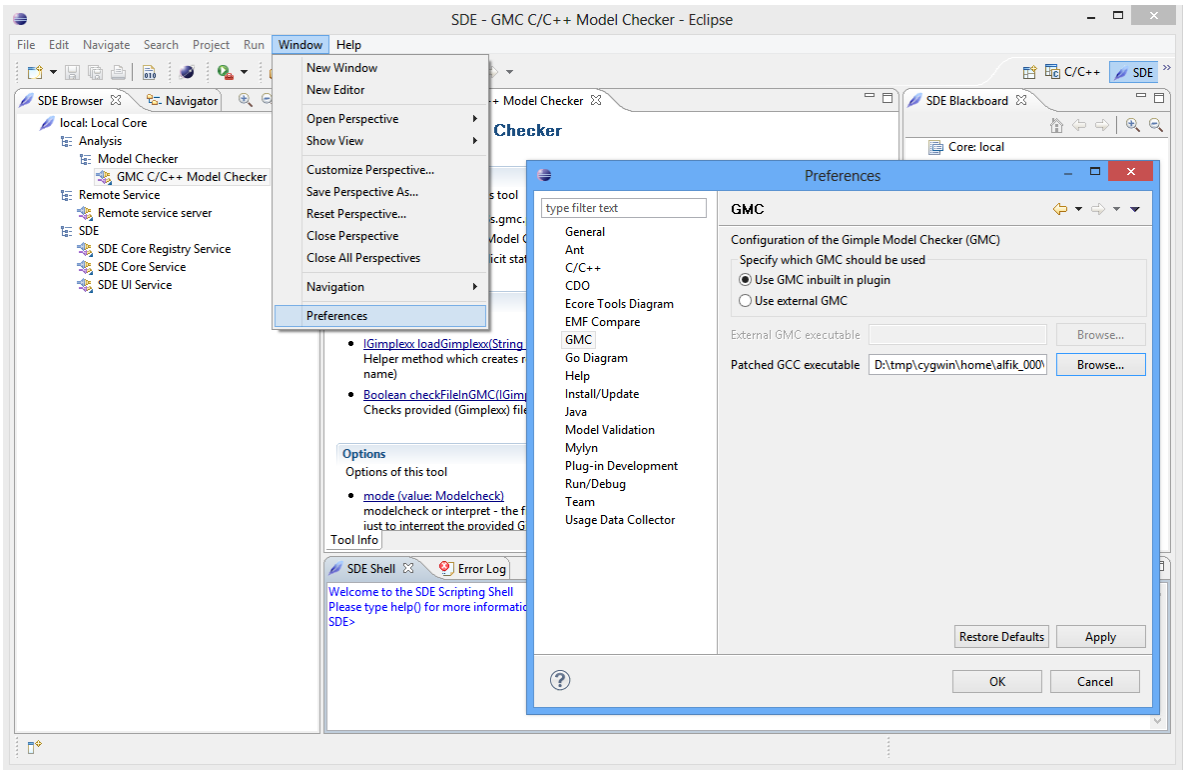

Figure 11: Eclipse extension settings

Command line First, it is necessary to specify the path to the extended GCC to be used. To use the default GCC, it is possible to execute the script `source ./setEnv.sh` in the GMC root directory.

To run the model checker, the script `dist/GMC` can be used. This script uses the extended GCC to create a gimplexx file containing intermediate representation from the provided sources code files of the program. Then the script runs a model checker on the gimplexx file. The `GMC` script takes three or more parameters.

The first parameter of the the script specifies the compiler to be used. It can be either:

`gcc` for treating the source code as C, or

`g++` for treating the source code as C++.

The second parameter specifies the mode in which GMC works,

`-i` for Interpret mode – checks one random thread interleaving, or

`-m` for Model-check mode – explores all thread interleavings.

The model-check mode (`-m`) has to be used in order to exhaustively search for potential errors.

The remaining parameters are the files with the source code of the program to be checked. The 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. This file can be later passed to the actual model checker executable `dist/ModelChecker`.

Fig. 12 shows how to use GMC and its output when no error is found. When running GMC, the output of the checked program as well as the overall result is printed to the standard output. In Fig. 13 there is an example of the GMC output with an error. If an error is found, the output of GMC contains a description of the error and a sequence of the GIMPLE instructions (and location in the source code) that leads to the error.

```
~/GMC/dist$ source ../setEnv.sh
exporting GMC_DIR= "~/GMC"
exporting GCC_VERSION= "4.6.4"
exporting GCC_DIR_INSTALL= "~/GMC++/gcc-4.6.4-patched.install"
~/GMC/dist$ ./GMC gcc -m examples/fib.c
Computes fibonacci numbers by recursion
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
...
fib(25) = 75025
Model checker does not found any error. See ModelChecker_stderr for its output.
~/GMC/dist$ cat ModelChecker_stderr
Everything is OK!
Deserializing...OK.
*********
No errors detected.
```

Figure 12: GMC usage example

```
~/GMC/dist$ ./GMC gcc -m examples/rand_mc_2.c
Rand example:
  computes k = 1 / ( i + j - 4)
---------------------------------------
   i=0
   i=0    j=0
   i=0    j=0    k=0
...
   i=1    j=3
Model checking failed, see ModelChecker_stderr for more details.
~/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(&"\nRand example:"[0])
             at examples/rand_mc_2.c:16
Thread: 0, Instruction: &__builtin_puts(&"\tcomputes k = 1 / ( i + j - 4)"[0])
             at examples/rand_mc_2.c:17
Thread: 0, Instruction: &__builtin_puts(&"---------------------------------------"[0])
             at examples/rand_mc_2.c:18
...
```

Figure 13: GMC error trace example

Eclipse IDE The Eclipse integration represents a GMC ready toolchain which helps to create the gimplexx files during the build and integration of the model checker itself. First, the Cygwin (resp.

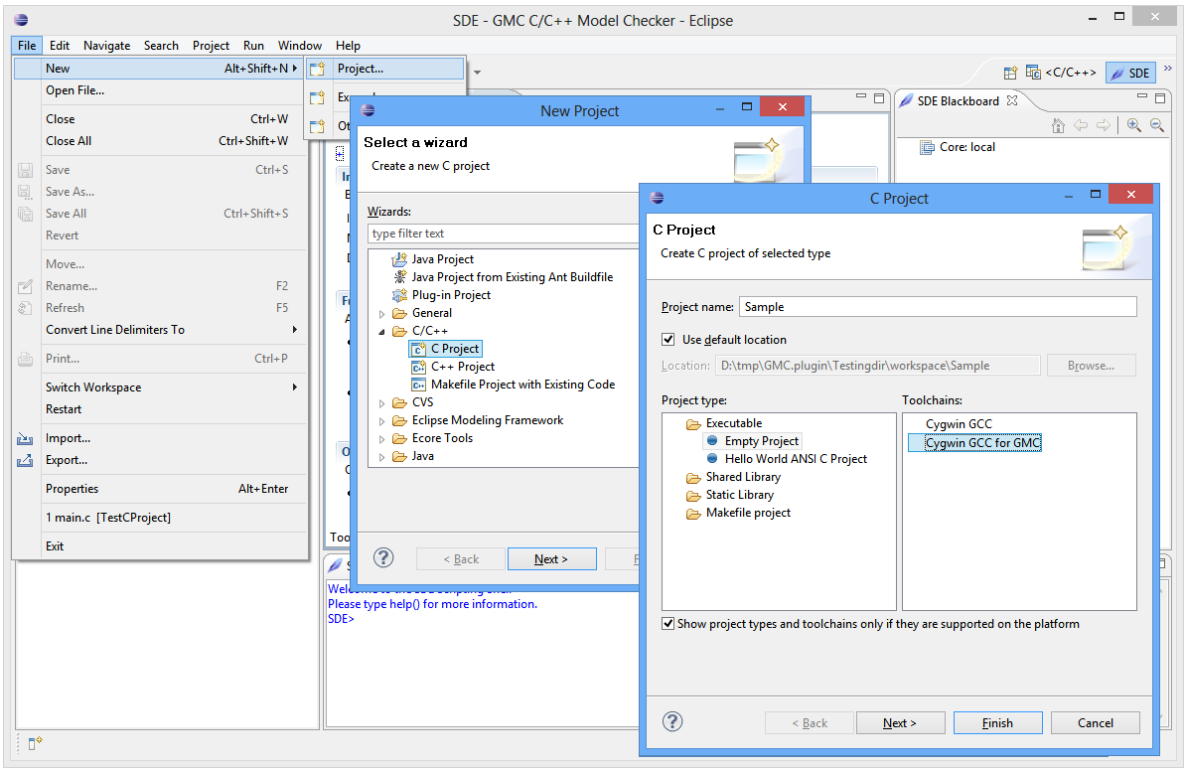

Figure 14: GMC ready project

Linux) GCC for GMC toolchain must be specified when C/C++ project is created; see Fig. 14. Once such a project is built, the gimplexx file (named according to the resulting executable file) is created in the project root. Then it is possible to execute GMC on that file by choosing Verify from the context menu. The GMC output is shown in a specialized console view; see Fig. 15.

The GMC model checker can also be used through the Service Development Environment – a common integration platform for the ASCENS project. GMC is exposed as one of the analysis tools. The tool can be configured via configuration options – it is possible to specify the interpreting (INTERPRET) or model checking (MODELCHECK) mode. The gui options specify whether the console containing the GMC output should be shown. The implementation also provides helper methods to load existing gimplexx files either from a provided file name or via the GUI dialog. Fig. 16 shows some basic settings of the GMC tool in SDE.

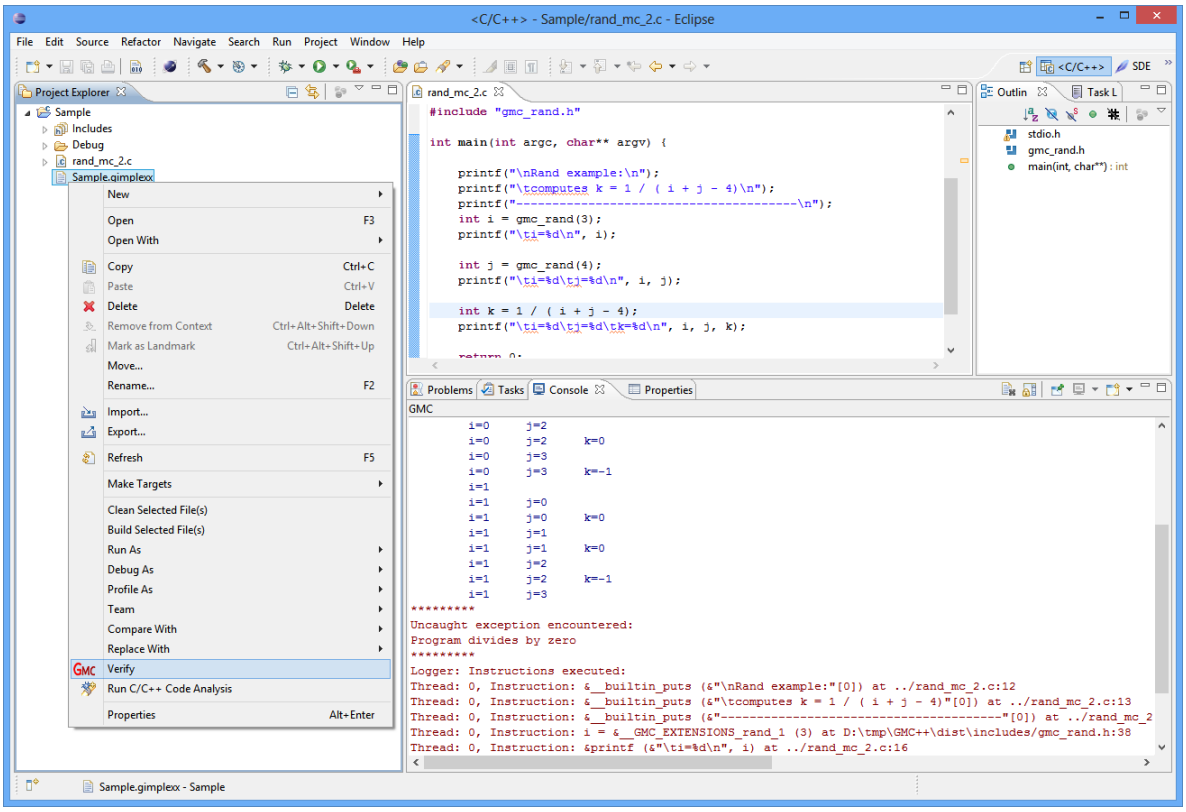

Figure 15: GMC context menu command and console output

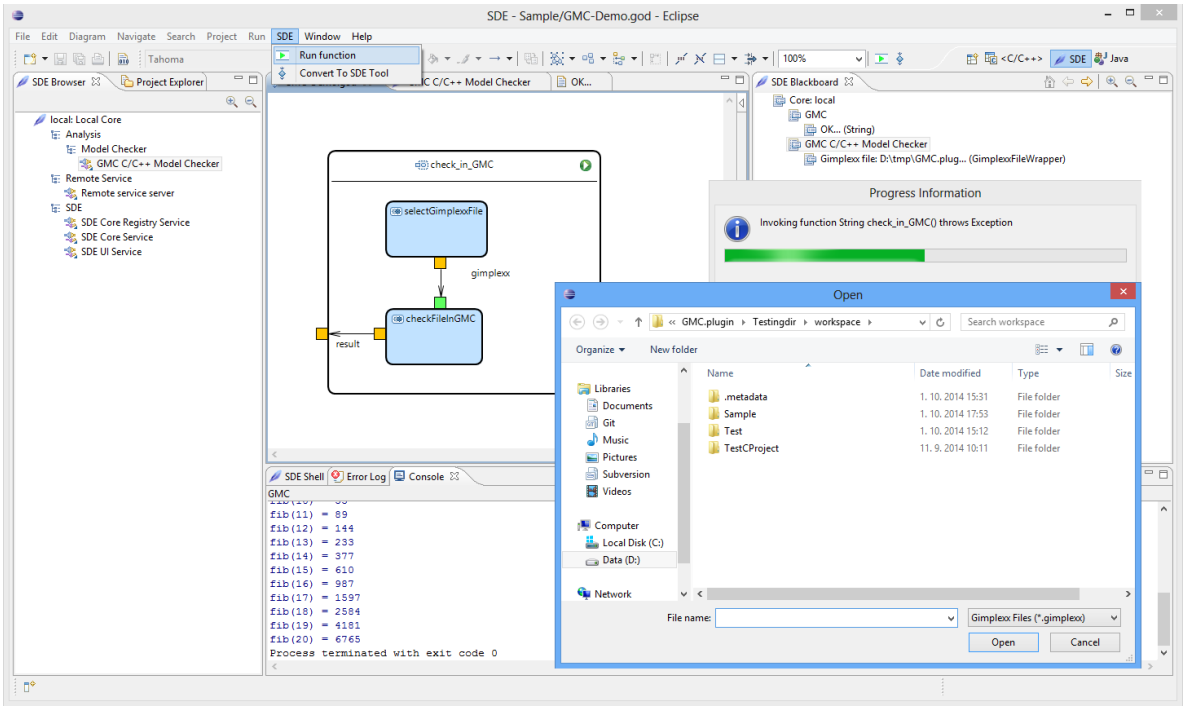

Figure 16: Usage of GMC from SDE

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 perfomance 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).

The ARGoS architecture, based on advanced concepts from C++ templates, allows users to extend any aspect of ARGoS without touching its core. With ARGoS 3, it is possible to code robot behaviors also with the Lua scripting language, besides the traditional C++ approach. ARGoS 2 is also integrated with the well-known network simulator ns3[2], allowing for hybrid simulations involving both the physics and the communication dynamics of robot swarms.

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 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

[2] `http://www.nsnam.org`

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.

One essential aspect of autonomic and adaptive systems in SCEL is communication. The SCEL components execute and cooperate in a highly dynamic environment to achieve a set of goals. A SCEL program typically consists of a set of components that exchange information through a knowledge repository. The jRESP communication infrastructure reflects this architecture, avoiding any centralised control. 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.

To simplify the integration with other tools and frameworks, 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.

In jRESP, policies can be used to authorise local actions and to regulate the interactions among components. Policies can authorise or prevent the execution of an action and, possibly, adapt the agent behaviour by returning additional actions to be executed. jRESP provides an interface for integrating different kinds of policies.

jRESP integrates FACPL as one policy specification mechanism. A compiled FACPL policy is consulted through the jRESP policy interface. The policy returns not only a *decision* (permit or deny), but also a set of *obligations*, which are rendered as a sequence of actions that must be performed just after the completion of the current event – if the decision is permit, the corresponding agent can continue as soon as all the obligations are executed, but if the decision is deny, the requested action cannot be performed. Possible obligations must still be executed, and after their completion, the previously forbidden action can be evaluated again.

To support analysis of adaptive systems specified in SCEL, jRESP also provides a set of classes that permit 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.

Relying on jRESP simulation environment, a prototype framework for statistical model-checking has also been developed. A randomized algorithm is used to verify whether the implementation of a system satisfies a specific property with a certain degree of confidence. 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 permit evaluating the probability to reach, within a given deadline, a configuration where a given predicate on collected data is satisfied.

To simplify the development process and to simplify the use of formal tools, we find it useful to have a high level programming language that enriches SCEL with standard programming constructs (for example control flow constructs such as loops or branches, or structured data types). For this reason, we have defined HL-SCEL, a SCEL inspired high level programming language for simplifying design, development and deployment of autonomous and adaptive system. We have also developed an Eclipse plug-in named SCEL SDK that uses XText to automatically generate jRESP code that can be used to simulate and execute the programmed system. SCEL SDK is integrated with jSAM (see Section 2.1) to provide a simplified interface for supporting quantitative analysis and statistical model checking.

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. jRESP is also available as an Eclipse plug-in that can be installed via the update site `http://jresp.sourceforge.net/eclipse/plugin`. In this case, the installation wizard automatically checks and installs the needed dependencies.

3.3 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 is integrated with the OMNeT++[3] network simulator and with the MATSim[4] traffic simulator. Integration with OMNeT++ allows to realistically simulate jDEECo applications with respect to network infrastructure behavior – OMNeT++ provides detailed models of hardware used in nowadays wired and wireless networks together with implementations of different communication protocols recognized so far as standards.

Similarly, integration with MATSim makes it possible to simulate the mobility of jDEECo deployment nodes. MATSim comes with an extensive agent-based framework. We leveraged its transport simulation functionality by adding the concept of sensors and actuators in jDEECo. With these, each component is capable of retrieving the current geographical location of the node it is deployed on as well as set its position to the desired one.

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 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 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.

[3] http://www.omnetpp.org
[4] http://www.matsim.org

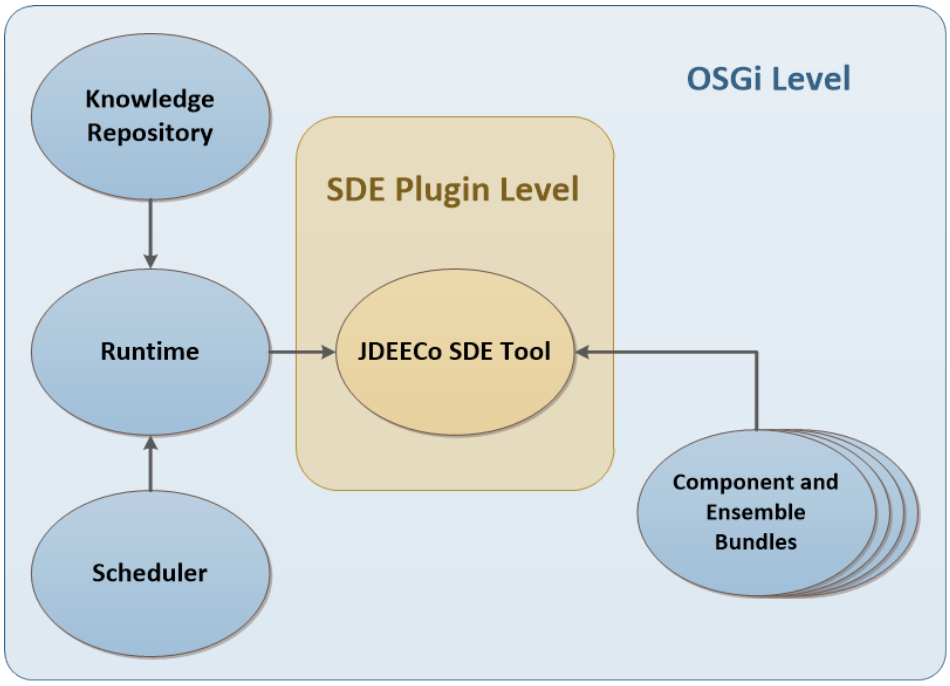

Figure 17: jDEECo SDE Tool - OSGi-SDE Integration

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 17. 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.

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.

3.4 AVis

Monitoring in the EDLC is an activity performed at runtime to observe and collect awareness data of the system and environment to trace awareness and adaptation capabilities. The monitored awareness data can be a component's status (e.g. its current location) or information about the environment in which the components are executing (e.g. monitored sensor data), and adaptation is the runtime modification of the awareness data in a component's knowledge repository.

In this context, the Awareness Visualizer (AVis) is an Eclipse plug-in we have developed for tracing the awareness and adaptation capabilities of an application executing in the jRESP runtime environment. The AVis plug-in, which contains three main components (i.e. model, view and controller), has been developed as a rich client application with Graphical Editing Framework (GEF) capabilities.

The AVis plug-in facilitates:

- Monitoring of changes to awareness data of an autonomic system at runtime. For this, the plug-in is integrated with the jRESP runtime framework.
- Visualization of the changes to the awareness data using graph-like representation. To this end, the plug-in implements several visualization features using GEF to record and highlight the adaptation.

A key benefit here is to provide feedback to the engineer about the behavior of the complex awareness mechanism used, thus helping the decision making process. This feedback can also improve any offline activities on the redesign of the system, verification and redeployment. We have validated and assessed our plug-in using two scenarios of the Swarm robotics case study in jRESP.

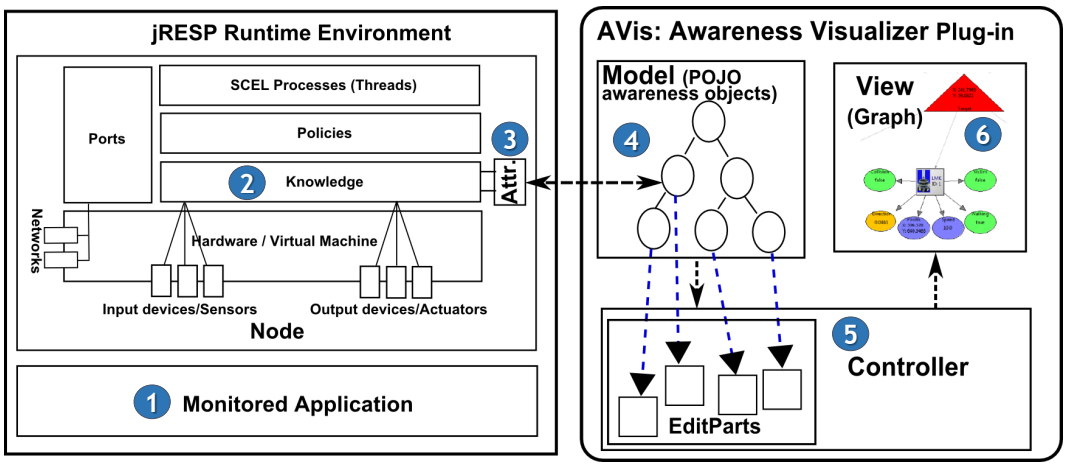

Figure 18: AVis plug-in system architecture and jRESP.

The AVis plug-in has been integrated with the jRESP runtime environment to facilitate the monitoring of changes to awareness data. Here, the monitored application (see step 1 in Fig. 18) can be any application scenario executing in jRESP. The Model encompasses the data portion of the plug-in architecture, containing POJOs (Plain Old Java Objects) created for the monitored awareness attributes. These are created at runtime using the knowledge attributes in the interface of a node in jRESP. We

employ the Observer-observable pattern in Java for listening and notifying the state of the POJO awareness objects in our visualizer plug-in when the corresponding state of the attributes in the node's interface are updated (see 3–4, Fig. 18).

Installation and Usage

The AVis plug-in can be executed in the Eclipse environment (e.g. Eclipse Java EE IDE for Web Developers, Luna Release/4.4.0). The user is required to install the Graphical Editing Framework (GEF 4) in the Eclipse environment via the update site mechanism for GEF 4 (`http://download.eclipse.org/tools/gef/gef4/updates/integration`). Also, the jRESP runtime environment with monitored case study examples needs to be downloaded and installed.

The AVis plug-in can be executed concurrently by running the relevant monitored case study example in jRESP. The AVis plug-in project can be downloaded from `https://sourceforge.net/p/avisplugin/code`. After downloading the AVis plug-in project, it needs to be imported to the Eclipse workspace. Add the AVis plug-in project and jRESP project to the other project's Java Build Path and set the Project References. Then, to execute the plug-in, add the provided two lines of code at the end of the instantiateNet() method of the Main class of the jRESP case study example. After the modifications, recompile the Main class and execute it.

```
AVisViewMain visualizer = new AVisViewMain(nodes);
visualizer.run();
```

3.5 Iliad

Iliad is a framework for building awareness mechanisms [HG15] for open-ended, distributed systems based on machine learning and reasoning techniques. It supports deep learning and hierarchical reinforcement learning, predicate-logic reasoning with integrated support for constraint processing, inference in Bayesian networks, and heuristic planning.

Iliad's input language is called POEM. In POEM, programmers can leave choices of actions or values partially unspecified and indicate which learning or reasoning mechanisms should resolve the non-determinism of each choice. Therefore developers can either establish fixed behaviors, indicate design-time preferences or simply state the possible actions. Iliad will optimize these choices either by reasoning or by learning from feedback provided by the environment. Given sufficient knowledge or training, the actions determined by Iliad will converge to those with the highest expected value for the environment in which the ensemble is operating.

Iliad is based on a flexible communication protocol called Hexameter. Hexameter implements the SCEL **get**, **qry** and **put** operators on top of the cross-platform, open source networking library ØMQ (`zeromq.org`). At the moment of writing Hexameter front-ends for Lua, Common Lisp, Java and JavaScript are available and can seamlessly interoperate. Therefore, Iliad can not only be used as a stand-alone reasoner but also as a knowledge repository for SCEL or as learning component for other reasoning systems such as the KnowLang reasoner.

Installation and Usage

The main components of Iliad are: the Hexameter communication infrastructure, implementations of extended behavior trees in several languages, a Common Lisp-based blackboard system and different reasoning and learning engines, and a jRESP/Hexameter binding. Each component is developed in its own repository. To facilitate the installation of the complete Iliad system, the

Academia (meta-)project provides a single repository that integrates all Lua, Lisp and JavaScript-based components of Iliad. For most users it is therefore sufficient to clone the Git repository at `https://github.com/hoelzl/Academia` and follow the installation instructions contained in the `Readme.asciidoc` file. The Java-binding for Hexameter, which is also required to access the Iliad reasoning and learning engines from jRESP, is contained in the repository `https://github.com/thomasgabor/hexameter-java`. For most users it is sufficient to download the file `hexameter-java.jar` contained in this repository and to add it to their classpath.

The directory `Scenario` inside the *Academia* project provides several example scenarios. The `obstacle` scenario illustrates the complete ASCENS toolchain for swarm robotics. It contains a robot controller written in jRESP that uses the Hexameter protocol to control robots running in the ARGoS simulator. A screencast showing how to run this example is contained in the `Doc` folder of the *Academia project*.

3.6 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.

At the network layer, SCP provides an implementation based on 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. An alternative implementation for communication on the application level integrates a gossip (endemic) strategy, which uses dedicated roles at each node as specified in the Helena approach [KMH14]. This increases scalability since it does not depend on global broadcasts as in the ContractNET implementation, and serves to structure the implementation along role-based lines.

The SCP can also take advantage of IaaS (Infrastructure-as-a-Service) platform such as the Zimory Cloud [Zim14], when available. If no node is available for executing a certain application, a new virtual machine with the required capabilities is started on demand. Once online, the application is moved to this machine. If the application is later shut down or a non-virtualized machine becomes available, the virtual machine is shut down again, thus conserving energy.

The Science Cloud Platform serves as the main technical demonstrator for the cloud case study of ASCENS, integrating many of the newly researched methods and techniques into one software system.

Installation and Usage

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, all version of the science cloud are available for testing and runtime.

The latest version is built on top of Java, OSGi, the Pastry library, and can use the Zimory IaaS when available. The 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.). The UI allows complete control over the individual SCPi and

contains monitoring functionality. A new *monitoring* functionality is available which shows the *big picture* of the cloud at runtime and contains demonstration scripts.

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.

A user guide which contains the steps required to test the system is available on the web site along with screencasts demonstrating the functionality, including the Zimory integration.

3.7 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.

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 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 19 for an example.

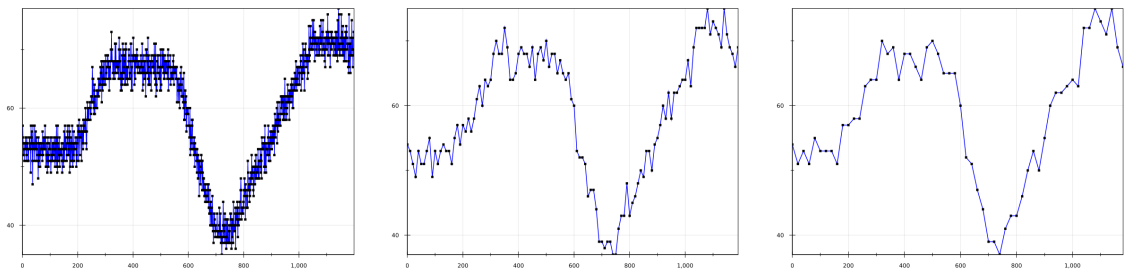

Figure 19: Graphs of different quality provided for different monitoring application load.

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 tool[5] 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.

[5] `http://www.pylot.org`

4 Conclusion

The ASCENS tools provide a collection of features that cover multiple phases of the ensemble development lifecycle, briefly outlined in the introduction. The features emerged from two major activities pursued in the project, namely the theoretical development of the methods and techniques for engineering adaptive ensembles, and the practical application of the methods and techniques on the case studies.

The tools reflect the explorative character of the ASCENS project and the entire FET program – as out understanding of both theoretical foundations and practical essentials of ensembles developed, so did the tools change. Thus, the tools should not be viewed as definite products, but as research prototypes that encourage further research and development.

We believe the open ended nature of our tool development effort is essential. Obviously, no fixed set of shrink wrapped research tools can anticipate the needs of future research and development in the domain of adaptive systems. Throughout the project, we have therefore focused on opening the tool development process, providing and maintaining public access to both code and documentation as much as practically possible, so that both the individual project partners and the research community at large could benefit.

At the very end of the project, we add emphasis on another aspect of result dissemination – the continuous existence of our tools beyond the project conclusion. We are proud to report that all our major tools are backed by one or more of the project partners, who continue extending the tools alongside their particular research directions.

Finally, we point out that the tools are linked from a central portal within the project website, `http://www.ascens-ist.eu`, where their most current versions are available.

References

[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. *Transations on Computational Logic*, 1(1):162–170, 2000.

[BBH+12] Lubomir Bulej, Tomas Bures, Vojtech Horky, Jaroslav Keznikl, and Petr Tuma. Performance Awareness in Component Systems: Vision Paper. COMPSAC '12, 2012.

[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+13] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with maude. *Science of Computer Programming*, 2013.

[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.

[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.

[CL10] Francesco Calzolai 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.

[dA03] Luca de Alfaro. Game models for open systems. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 269–289. Springer, 2003.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC/SIGSOFT FSE 2001*, volume 26(5) of *ACM SIGSOFT Software Engineering Notes*, pages 109–120. ACM, 2001.

[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.

[gcc] GNU Compiler Collection. `http://gcc.gnu.org/`.

[gmca] Gimple Model Checker. `http://d3s.mff.cuni.cz/projects/formal_methods/gmc/`.

[gmcb] GMC Eclipse Plugin. `http://d3s.mff.cuni.cz/projects/formal_methods/gmc/plugin/update/`.

[HG15] Matthias Hölzl and Thomas Gabor. Continuous Collaboration: A Case Study on the Development of an Adaptive Cyber-Physical System. In *Proc. of the International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), Firenze, Italy*, 2015. to appear.

[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.

[KMH14] Annabelle Klarl, Philip Mayer, and Rolf Hennicker. Helena@work: Modeling the science cloud platform. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 8802 of *Lecture Notes in Computer Science*, pages 99–116. Springer Berlin Heidelberg, 2014.

[LLM13a] Diego Latella, Michele Loreti, and Mieke Massink. On-the-fly fast mean-field model-checking. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 297–314. Springer, 2013.

[LLM13b] Diego Latella, Michele Loreti, and Mieke Massink. On-the-fly fast mean-field model-checking: Extended version. *CoRR*, abs/1312.3416, 2013.

[LLM14] Diego Latella, Michele Loreti, and Mieke Massink. On-the-fly probabilistic model checking. In Ivan Lanese, Alberto Lluch-Lafuente, Ana Sokolova, and Hugo Torres Vieira, editors, *Proceedings 7th Interaction and Concurrency Experience, ICE 2014, Berlin, Germany, 6th June 2014.*, volume 166 of *EPTCS*, pages 45–59, 2014.

[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.

[MLa] MAIA. System Modelling and Analysis @ IMT Lucca. A maude tool for adaptable interface automata. `http://sysma.lab.imtlucca.it/tools/maia/`.

[MLb] MESSI. System Modelling and Analysis @ IMT Lucca. Maude ensemble strategies simulator and inquirer. `http://sysma.lab.imtlucca.it/tools/ensembles/`.

[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`.

[pth] POSIX Threads. `http://en.wikipedia.org/wiki/POSIX_Threads`.

[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 Trustworthly 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.

[sde] Service Development Environment (n.d.). `http://svn.pst.ifi.lmu.de/trac/sde`.

[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.

[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.

[Zim14] Zimory Software. Zimory Cloud Suite. http://www.zimory.com/, August 2014.