ascens

www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D6.1: First Report on WP6

SCE Tooling – Tool Integration Requirements and Technology

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **7.6.2010**

Lead contractor for deliverable: **LMU**
Author(s): **Roberto Bruni (UNIPI), Jacques Combaz (UJF-Verimag), Alberto Lluch Lafuente (IMT), Michele Loreti (UDF), Philip Mayer (LMU), Carlo Pinciroli (ULB), Stephan Reiter (LMU)**

Due date of deliverable: **September 30, 2011**
Actual submission date: **November 15, 2011**
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 document describes the requirements for the *SCE Workbench*, which shall integrate most of the tools developed in the course of the ASCENS project into a single user-friendly development environment that assists researchers and programmers in the design, creation and evaluation of *Service Component Ensembles (SCEs)*. Several of these tools are being built within the ASCENS project, while others are updated versions of existing tools to make them suitable for SCE development.

Our work is based on the results from the SENSORIA project, where an IDE with similar goals and intentions was built although for a different set of tools. The *Service Development Environment (SDE)* is based on the Eclipse IDE and augments it with functionality that enables simple and light-weight integration of tools into a common user interface with optional support for orchestration, i.e. the combined use of tools for the purpose of solving a specific problem. With minor adaptions we can use the SDE as a basis for our *SCE Workbench*. Both the SDE and tools that are being integrated at this point of time will be discussed in this document.

Additionally, a discussion of the work done during the first year of the ASCENS project is provided and an outlook on the months to come is given.

Contents

1 Introduction

We live in a connected world built on the progress that has been made in information and communications technology. This progress has enabled the development of inexpensive devices equipped with powerful processors and connected to the Internet. We not only find them at our desks at home or at the office as we used to, but also in our pockets in the form of smartphones that follow us through our daily life routine, that provide us with the information we need and that allow us to stay in contact with our colleagues, friends and loved ones. Furthermore, devices with wireless connectivity built for special tasks or domains are present in our lives virtually everywhere: GPS navigation assistance systems or tracking systems for the purpose of toll collection in our cars; RFID markers attached to consumer goods for quicker check-out in stores; or weather stations that report accurate and current data for better forecasts. These are just a few examples for the omnipresence of modern ICT devices in our daily lives.

The development of software for such large-scale, open-ended, distributed systems poses a difficult problem. State-of-the-art approaches to software engineering have been shown not to scale well to these types of systems, yielding software that cannot easily adapt to changes in the environment which are, however, important in order to have well-performing systems in the light of failing components or new resources becoming available. Manual intervention is therefore normally necessary which, however, provokes mistakes that can lead to temporary system outages in the worst case.

In the ASCENS project we refer to software-intensive systems with a massive number of nodes or complex interactions between nodes that also operate in open and non-deterministic environments as *ensembles*. Ensembles have to be able to dynamically adapt to new requirements, technologies or environmental conditions without redeployment and without interruption of the system's functionality, which blurs the distinction between design-time and run-time. Systems like the ones mentioned before will be built as ensembles of self-aware, adaptive components which will be combined so that designers can control and engineer emergent behavior with static and dynamic support from formal methods.

The development of a coherent, integrated set of methods and tools to build such systems is our goal in the ASCENS project. A major pillar of our concept is based on is the development of the *Service-Component Ensemble Language (SCEL)*, a language for self-aware, autonomic *Service Components (SCs)* and *Service Component Ensembles (SCEs)* that integrates behavioral description with knowledge representation and reasoning about the environment. Around a definition of the language and its run-time environment, not only a run-time system and a compiler will be created, but also tools that allow the analysis of programs or the partial translation of existing code in other languages to SCEL and vice versa. Furthermore, the development of tools which might be helpful in the engineering of software-intensive systems, their analysis, verification and visualization is a goal of the project. All of these tools shall be integrated into a single development environment, providing users with all the necessary tools at their fingertips, which will benefit future research efforts and industrial software developers alike.

In this document, we will first describe requirements for the integration of a set of tools into a single development environment. We will subsequently describe an existing development environment, which we use as a foundation for our work in the context of the ASCENS project. Following that we will give an overview of the tools that are being integrated into the environment, describing their intended purpose, the stage of integration and potential interfaces to other tools. We will conclude with a summary of the work done in the first year of the ASCENS project and also present an outlook on the tools that we are planning to integrate in the months to come.

2 Tool Integration Requirements

In this section, we will describe a number of requirements we deem necessary for a successful and usable integration of tools into a single development environment. We will revisit these requirements in a later section to evaluate our choice of a technological basis for the ASCENS *Integrated Development Environment* (IDE), the *SCE Workbench*.

The list of requirements as presented in Table 1 was created in fruitful discussions with our colleagues from the NESSoS project [BK11], another FP7 project with a focus on the engineering of secure future Internet software services and systems, in which the integration of security-related tools into an IDE is also a goal, just like in ASCENS.

User Interface	**Req. 1:** Provide an IDE with a Familiar User Interface **Req. 2:** Support Good Documentation of Tools **Req. 3:** Support Tool Categories
Architecture	**Req. 4:** Open Source Foundation **Req. 5:** Enable Easy Integration of Tools **Req. 6:** Support Broad Range of Tools
Tool Management	**Req. 7:** Support Management of Tools **Req. 8:** Allow for Automatic Updates
Tool Integration	**Req. 9:** Aid in Development of New Tools **Req. 10:** Enable Transformations between Tool Inputs and Output **Req. 11:** Support Inter-Tool Dependencies **Req. 12:** Enable Orchestration of Tools

Table 1: Requirements for a Tool Integration Platform

Req. 1 Provide an IDE with a Familiar User Interface

The development environment, which shall be a result of the ASCENS project, should be easy to use for programmers and researchers in order to increase the acceptance of the IDE. A means of achieving this is to rely on established principles for such environments, i.e. the standard layout of the IDE controls and windows, but also the possibility to rearrange them by the users to enable customization. This can be achieved by designing a new IDE while keeping in mind these principles or by reusing a preferably open source IDE, such as CodeBlocks[1] or Eclipse[2], which can both be extended and modified to create domain-specific development environments.

Req. 2 Support Good Documentation of Tools

The use of tools, either by themselves or as parts of composing workflows, requires an understanding of both the interfaces of the tools and the semantics of their methods. Although desirable, programmers do not often invest the time to properly document their work due to time constraints. In order to reduce the time required to document a tool, automatic mechanisms shall assist the developer in building a documentation: Interfaces with their methods and datatypes shall be automatically extracted by the development environment, leaving only the documentation of their semantics to the programmer. All of this information shall be readily available in the IDE, e.g. via the list of available tools.

[1] CodeBlocks website: `http://www.codeblocks.org/`
[2] Eclipse website: `http://www.eclipse.org/`

Req. 3 Support Tool Categories

In the ASCENS project we will have different kinds of tools, such as tools for development, for the analysis and verification of models or programs, for simulations and for transformations. The IDE user interface should be able to present them grouped by their category in order to allow users to quickly find tools for their needs without having to go over a flat list of all tools.

Req. 4 Open Source Foundation

The foundation of the *SCE Workbench* shall be available under an open source license to allow new versions without any restrictions. Furthermore, with open-source software we are independent from the publisher's support to fix bugs, but can easily do this ourselves.

Req. 5 Enable Easy Integration of Tools

The integration of tools, both newly developed and existing ones, shall be straightforward and not require highly invasive changes to the tools' code bases. A declarative approach in which a tool's interface is annotated with additional information that allows its integration into a platform could be a way to approach this.

New tools will typically be developed in programming languages well supported by the development environment because this reduces the effort of using its interfaces. However, programmers should be free to select the language they feel most productive with. The IDE therefore needs to support the necessary means to establish a link and integrate such tools, which is also a requirement for the support of legacy tools written in other programming languages.

Packaging of tools to the required format should be made simple to perform, e.g. by providing an automation of the necessary workflows.

Req. 6 Support Broad Range of Tools

Tools can be quite different in nature: Some are computationally expensive, but only have limited interaction with the user, if any. Others are lighter on the CPU, but require complex user interfaces. Our development environment shall support all kinds of tools and optimize their use, e.g. by running computationally-expensive tools on dedicated remote machines, while keeping tools with high amounts of user interaction local in order to reduce input latency and provide a pleasant experience to the user.

Req. 7 Support Management of Tools

At any time, the user should be in control of the tools that are installed on his machine and integrated into his development environment. The IDE shall therefore provide means for managing the installation and the removal of tools in an easy and straight-forward way.

Req. 8 Allow for Automatic Updates

An automatic update mechanisms shall keep the installed tools up-to-date, which is especially important during the early lifetime of any tools, i.e. the alpha and beta stages during its development. This would enable early-adoption of tools developed with the ASCENS project and minimize frustration with keeping up-to-date with the latest developments, i.e. additions of new features or bug fixes.

Req. 9 Aid in Development of New Tools

Within the ASCENS project new tools will be developed and integrated into our development environment. This development of tools shall be supported by the development environment to enable good integration from the start. Existing tools might also be useful in the process of developing a new tool.

Req. 10 Enable Transformations between Tool Inputs and Outputs

Tools require input and produce output in certain formats. For the orchestration of tools the output of one tool needs to fulfill the requirements for input of the subsequent tool. In order to avoid having to add support for a certain input format to a tool, which is sometimes simply not possible, the use of adapters shall be made possible by the development environment. Adapters accept a certain form of input and transform it so that it can be fed to a tool that didn't support the original input.

Req. 11 Support Inter-Tool Dependencies

Tools can be dependent on other tools to function properly, e.g. the analysis of a web service description might require the presence of an XML parser and a WSDL interpreter. Our IDE shall support the annotation of tools with regard to their external dependencies and shall verify the satisfaction of them before any tool is made available to the user to guarantee that it is fully functional. External dependencies should also preferably be solved by automatically installing required software components in order to reduce the installation effort.

Req. 12 Enable Orchestration of Tools

Typically, a suite of tools is used to deal with a particular problem. Our development environment shall enable the user to use integrated tools not only via manual interaction with the graphical user interface, but also via custom workflows. A workflow is a description of a sequence of tool invocations in which the results or the output of one tool are passed as input to another tool which further processes the data. Preferably, such a composition of tools is easy to achieve, for example via a graphical editor that is based on a familiar model such as UML activity diagrams or via scripting, i.e. the description of a workflow in a textual form in a domain-specific programming language.

3 SDE Technology

Considering the requirements discussed in the previous chapter, we will present an integrated development environment from the SENSORIA project that can be a great basis for our work in the ASCENS project in the following.

The SENSORIA project [WBF+08] has provided tools and techniques for many of the tasks developers are faced with during the development of systems based on the paradigm of Service-Oriented Architectures (SOA) [Erl05]. The main consideration in SENSORIA was rigorous engineering of service-oriented systems with a specific focus on formal verification. As our verification and validation methods are often directly based on a formal model, tool support had to be created for allowing developers to use these methods while staying on their chosen level of abstraction, for example, UML. For this purpose a tooling platform, the SENSORIA Development Environment (SDE) [MR10], was developed, which integrates the various tools required in the service development process, including modeling, analysis, code generation, and runtime functionality. In short, the SDE

- offers a extensible repository of a broad range of tools and describes their functionality and area of application in a user-friendly way,
- allows developers to use tools in a homogeneous way, with the possibility to re-arrange and combine tool functionality as required, and
- enables users to stay on a chosen level of abstraction, hiding formal details as much as possible.

In section 3.1, we give a high-level overview of the SDE while Section 3.2 further details the design and implementation of the integration platform.

3.1 High-Level Overview

The SENSORIA project aimed to support developers of service-oriented software systems at various points in the development process. Specific focus was placed on (formal) verification of service artifacts, which includes appropriate modeling support for developers as well as code generation and runtime support. Through various tools, the project was thus able to offer functionality which covers the complete model-driven process of service engineering, which is shown in Figure 1.

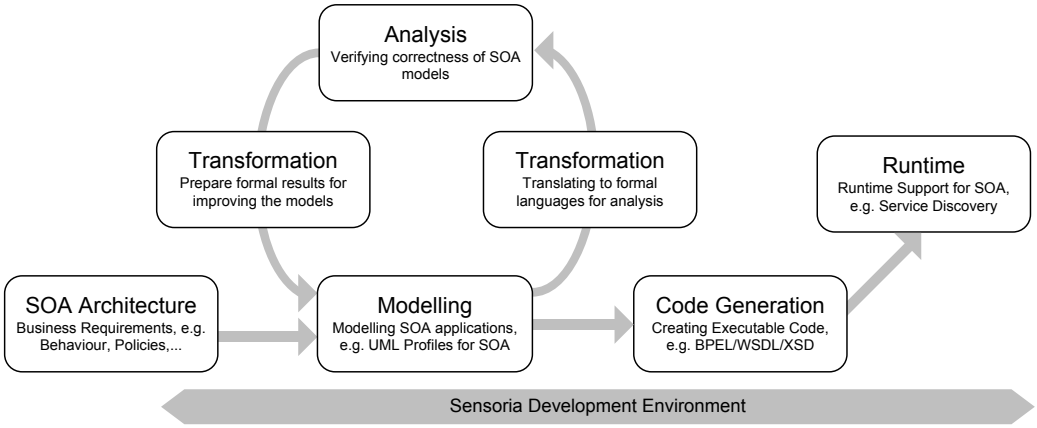


Figure 1: SENSORIA Development Approach

After starting with requirements for a SOA-based system, developers advance to the modeling phase. From this phase, various analyses of the models may be performed, many of them carried out with the help of automated model transformations. Finally, code is generated from the improved models; runtime support is available for executing this code on various platforms. The figure shows the phases which are covered by tools integrated into the SDE – Modeling, Transformation, Analysis, Code Generation, and Runtime. The following functionality is available in each of these phases:

- **Modeling:** Graphical editors for familiar modeling languages such as UML, which allow intuitive modeling at a high abstraction level, and also text- and tree-based editors for formal languages like process calculi.
- **Model Transformation Functionality, including Code Generation:** Automated model transformations from UML to process calculi and back to bridge the gap between these worlds; also, generation of executable code (for example, web service standards like BPEL).
- **Formal Analysis Functionality:** Model checking and numerical solvers for stochastic methods based on process calculi code defined by the user or generated by model transformation.
- **Runtime Functionality:** Integration of runtime platforms, for example BPEL process engines or the Java runtime, as well as run-time support for services, for example dynamic service brokering.

The functionality indicated in the previous list is implemented in various tools, some of which have been developed within SENSORIA, some developed outside of the project. The tools are not only developed at different sites, but are also vastly different with regard to user interface, functionality, required computing power, execution platform and programming language. However, all of the tools contribute to the development process and in many cases deliver artifacts which may serve as input to other tools.

The SDE provides this functionality through a carefully designed, lightweight integration architecture. This is achieved through the following core features:

- **A SOA-based Platform:** The SDE itself is based on a Service-Oriented Architecture, allowing easy integration of tools and querying the platform for available functionality. The tools hosted in the SDE are installed and handled as services.
- **A Composition Infrastructure:** As development of services is a highly individual process and may require several steps and iterations, the SDE offers a composition infrastructure which allows developers to automate commonly used workflows as an orchestration of integrated tools.
- **Hidden Formal Methods:** To allow developers to use formal tools without requiring them to understand the underlying formal semantics, the SDE encourages the use of automated model transformations which translate between high-level models and formal specifications.

As with services in a SOA, tool composition in the integration tool is a lightweight one, i.e. the connection between tools is not a priori fixed and adding additional tools requires only minimal change to the integrated tools. Using the tool-as-a-service metaphor, tools are services, each consisting of functions which can be invoked by the user or other services. Contrary to web services [WCL+05], user interaction is very important for some software development tools. For example, a modeling tool requires a lot of user interaction – ideally, the modeling tool runs on the computer of the user. A model checker, on the other hand, requires a lot of computing power and thus will most likely run on

a dedicated server to be accessed remotely with none or only a minimal, generated UI available. Both use cases are supported in the SDE.

By using a SOA-based infrastructure, combining tools into more complex tool chains is straightforward, i.e. via dedicated orchestration languages. A typical scenario for tool composition can be found in the analysis and verification of software; for example, model checkers require a certain input format into which most source models first need to be transformed; the same applies to the output. The SDE contains both a textual (JavaScript) and a graphical (UML-based) orchestration language, allowing users to integrate various tools, thereby handling the data flow between these tools. Having encapsulated the integrating steps, they can be run over and over again for performing the same steps with different input and output data.

Finally, the SDE aims at providing formal verification tools to pragmatic developers. This requires, as indicated above, the use of model transformations to allow developers to stay on their chosen level of abstraction while still enjoying the results available through rigorous verification methods.

Figure 2 shows the architecture of the SDE. As discussed previously, the integration platform hosts a number of tools as services. Through its dedicated orchestration infrastructure, the SDE allows developers to orchestrate tools to be used in combination, which includes using model transformations and a remote invocation functionality for invoking tools hosted on different machines.

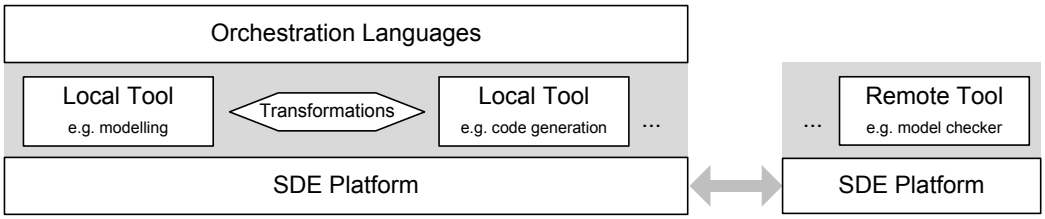


Figure 2: SDE Architecture

3.2 Design and Implementation

The aim of SENSORIA was to support the creation of service-oriented software by augmenting existing development processes and tools. A requirement for the SDE was therefore to integrate with existing tools and platforms for the development of SOA systems. For this reason, the SDE is based on the well-known Eclipse platform [Ecl11] and its underlying, service-oriented OSGi framework [The07]. OSGi is based on so-called bundles, which are components grouping a set of Java classes and meta-data providing among other things name, description, version, exported and imported packages of the bundle. A bundle may provide arbitrary services to the platform.

3.2.1 SDE Core and UI

The technical architecture of the SDE is depicted in Figure 3, which shows the SDE Platform as an OSGi bundle, its dependencies and dependent bundles.

Fundamentally, all tools are integrated as OSGi bundles which offer certain functions for invocation by the platform. As indicated above, the tools integrated into the SDE are vastly different, ranging from user-driven graphical modeling tools to computationally intensive analysis tools with very basic interaction mechanisms. Thus, it is not possible to define a common API for all tools. In the SDE, this problem is solved by using (declarative) OSGi services for each tool. Furthermore, the SDE allows tools to provide their own UI, but also provides a generic invocation mechanism which enables users to invoke arbitrary functions, either directly or through an orchestration. Finally, tool

integration requirements should be kept low to ensure integration of as many tools as possible. The SDE re-uses OSGi and Eclipse technology and declarative service descriptions which are generated from Java annotations for a fast and straightforward integration process.

As can be seen in Figure 3, the SDE platform and the integrated tools are based on (R-)OSGi only or, more specifically, the Equinox implementation of OSGi [Fou09] (use of R-OSGi [RAR07] with SDE allows remote execution of tools). This means that fundamentally, tools must be implemented in Java, although they may wrap native code or remote invocations as they wish. Being only based on OSGi, they can be invoked completely independently from Eclipse. If they additionally choose to provide a UI, this UI is integrated into and based on the Eclipse platform, as is the UI for the SDE platform itself.

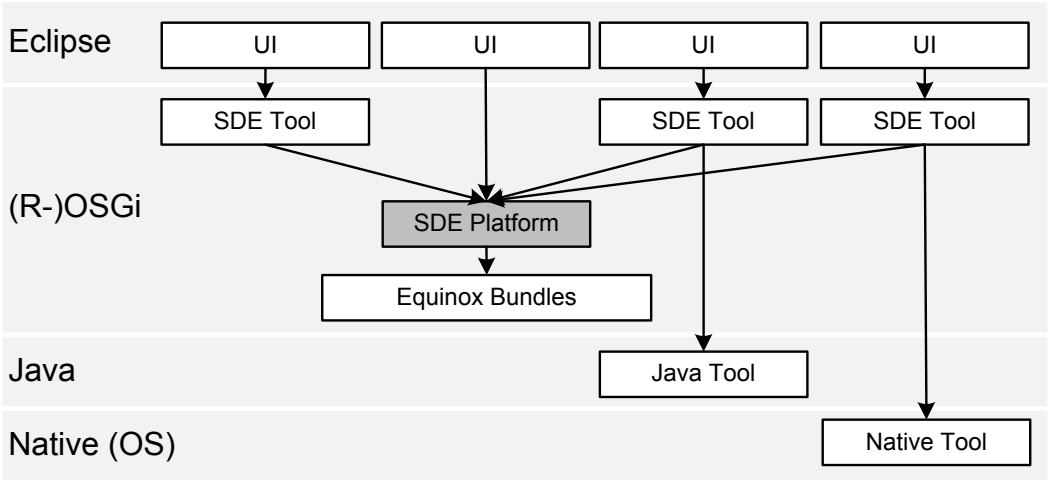


Figure 3: SDE Technical Architecture

Figure 4 shows a screenshot of the SDE UI. On the left hand side, the tool browser shows installed tools available for invocation and automation. Tools are grouped by category, allowing quick access by application area. Double-clicking a tool in the browser yields more information about the tool and its functionality. This information is shown in the view in the middle: As an example, an integrated tool for qualitative analysis (WS-Engineer) is shown in more detail. Each tool function displayed here can be invoked by clicking the link and providing the parameters. Finally, on the right, the SENSORIA Blackboard is shown, which is a storage area where tools may place arbitrary objects for later use. Finally, at the bottom, the SENSORIA Shell is displayed, which is a live JavaScript execution environment.

As an example for a function invocation, clicking on the `bpelToFSP()` function in the WS-Engineer tool yields the following dialogs, where the data for the single parameter `bpel` can be selected from various sources (Figure 5).

Finally, the SDE core integrates with R-OSGi to provide the ability to host tools for external invocation, and connect to remote SDE cores. The tools in the tool view in Figure 4 (left), for example, are listed under the local core. Further (remote) cores may be added as required, and their tools are then listed and used in the same way as described above. Furthermore, the blackboard (right) also distinguishes between the various cores.

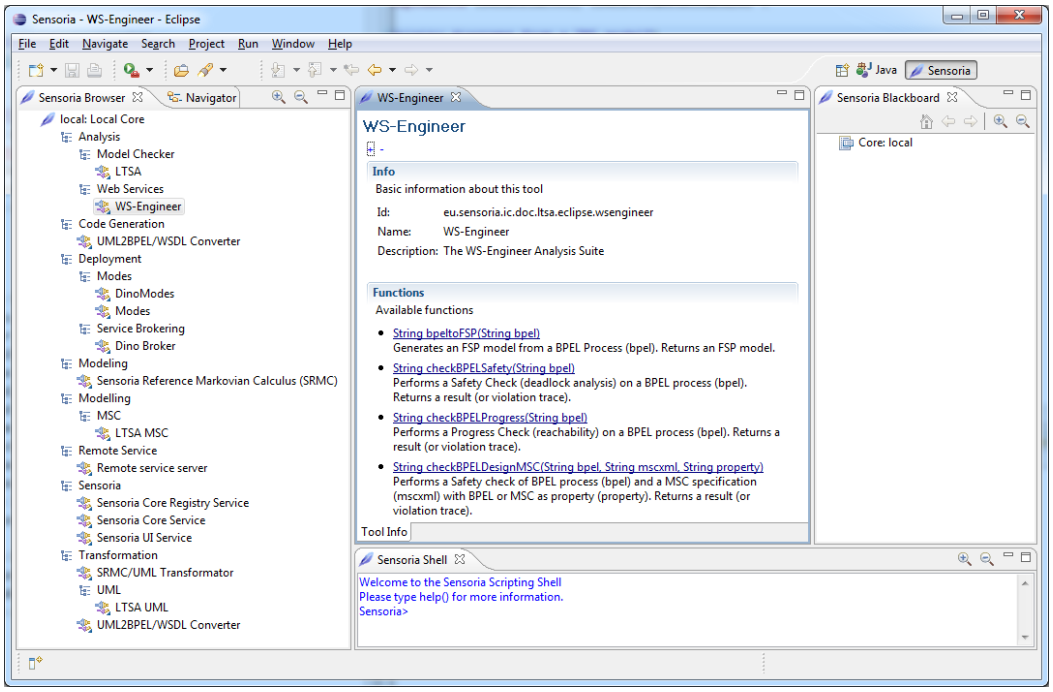


Figure 4: SDE Screenshot

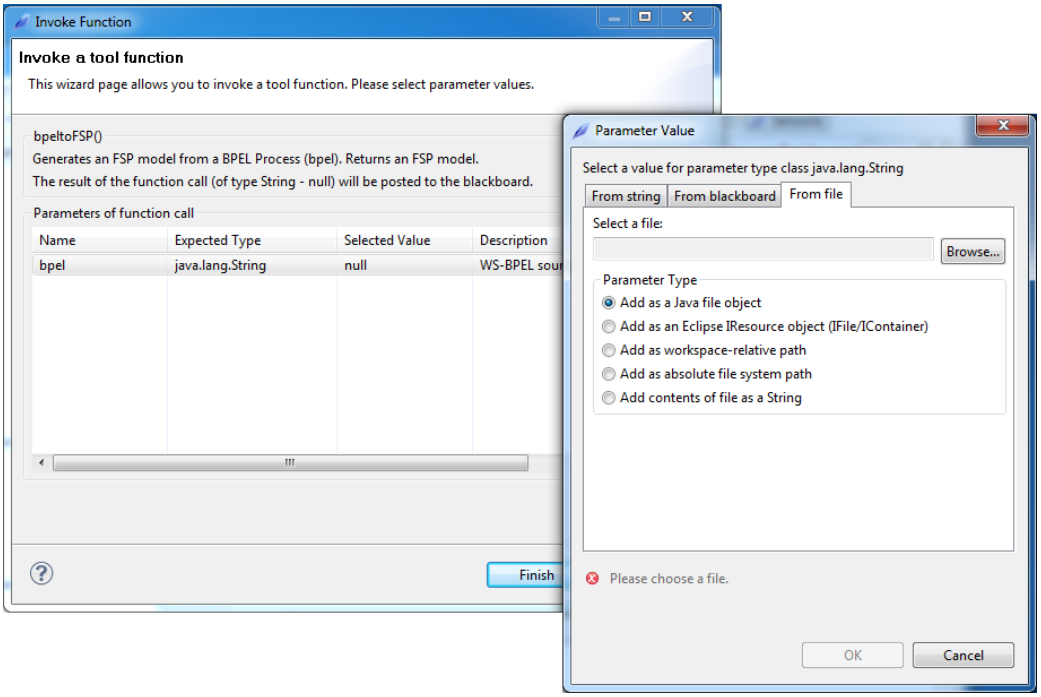


Figure 5: SDE Wizard

3.2.2 Composing Tools

The SDE provides the ability to compose new tools out of existing ones, a process known as orchestration in the SOA world. Creating orchestrations is possible using two mechanisms: A textual, JavaScript-based approach, and a graphical, UML-activity-diagram-like workflow approach.

Orchestrating with JavaScript. The ability to use tool APIs directly within JavaScript enables developers to create a workflow by simply invoking tool functions and passing data in-between those functions. To enable the newly created workflow to be usable as a tool in its own right, two things are required: Instead of simply creating a workflow, a JavaScript function definition is required which states a function name and parameters. As each tool, function, parameters, and return types may have descriptions and additional metadata attached, this metadata must be specified in some way in the JavaScript source files. Both points have been addressed in the SDE. The first is simple; function definitions are already part of the JavaScript specification. The second was solved by employing a JavaDoc-comment-style approach to metadata specification. Tags like `@description` are used to convey metadata information.

As an example, Figure 6 (left) shows a script for converting UML2 activity diagrams to BPEL, then analyzing them using the WS-Engineer tool, and finally converting the result back to UML2 sequence diagrams showing the error trace. Figure 6 (right) shows the converted tool inside the SDE tool browser. Scripts created like this can be used on any SDE installation which has the required tools installed. No particular deployment is necessary, copying the script and registering it with the core suffices.

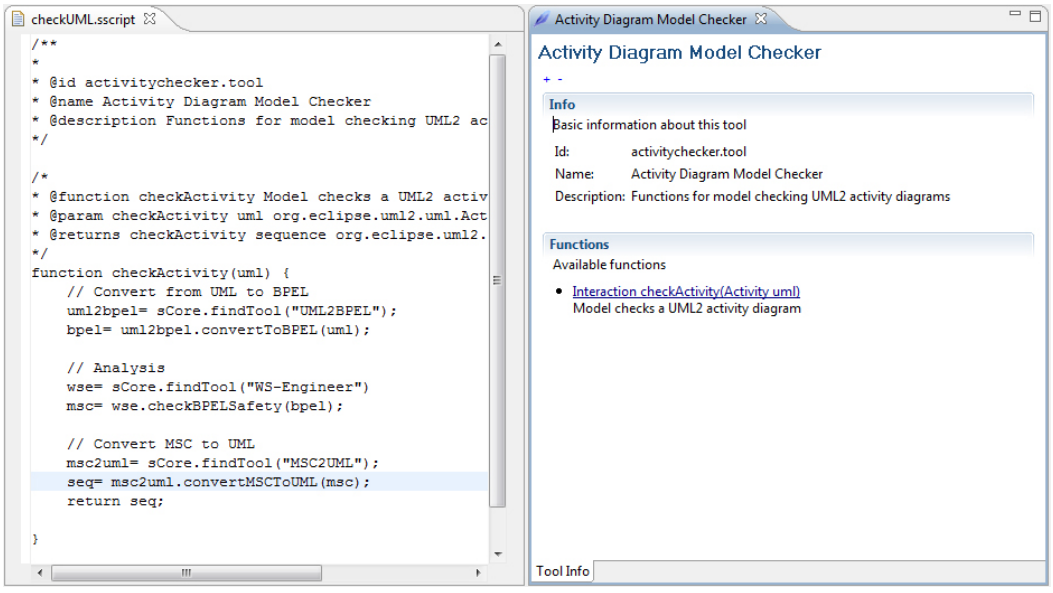


Figure 6: Orchestration with JavaScript

For testing purposes, the SDE also contains a JavaScript live execution environment, the SDE Shell (Figure 4), where JavaScript commands can be executed without compiling a complete script.

Graphical Orchestration. Besides the ability to use JavaScript for orchestration as indicated above, the SDE also contains the ability to orchestrate tools graphically. The syntax used is that of UML2 activity diagrams, where the main focus is on data flow, i.e. the flow of information from pin to pin.

An activity in the diagram represents one function in the tool to be generated which has input pins (parameters) and one output pin (return type). Inside the activity, actions represent function calls to arbitrary (installed) tools. These actions have pins themselves; data flow edges model the data transfer.

As an example, consider the screenshot in Figure 7, which shows the orchestration introduced in the previous paragraph as a graphical workflow, including the editor which supports it. The function `checkActivity(uml)` is modeled as an UML2 activity, and each call to a particular function of an installed tool is modeled as an action. On the right-hand side, the palette shows all available tools and the functions they provide. Once modeled, an orchestration such as the one above is converted to a Java class, compiled in-memory and installed as a tool in the SDE.

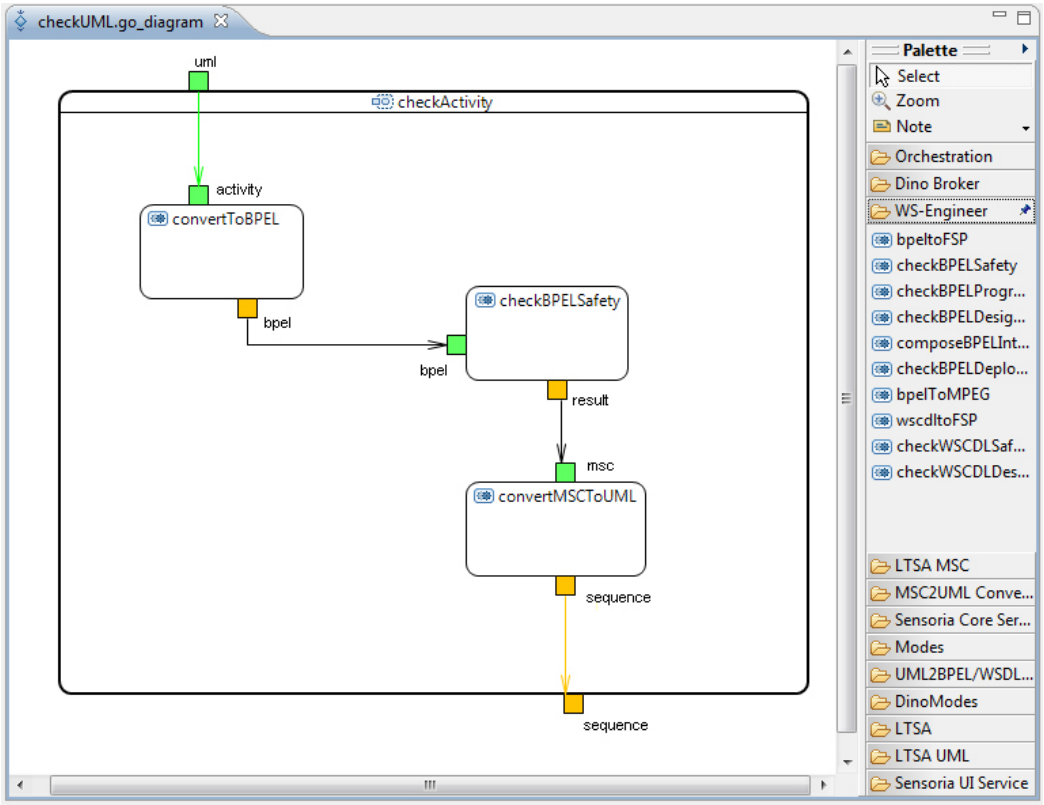


Figure 7: Graphical Orchestration

3.3 Extending the Platform

The SOA-based architecture of the SDE makes it easy to add new tools – the SDE publishes a core API and an extension point for registering tools. Basically, each tool is an OSGi bundle with some published API and metadata XML to register the tool with the SDE core. Thus, creating a wrapper class and registering the class with the SDE extension point enables tool functionality to be immediately available within the SDE, both for manual invocation and automation. Tools within the SDE are loosely coupled, as they are fundamentally independent from each other and interact through their published service interfaces only. They may, of course, require other tools to be installed for them to work. This is defined in a declarative way through the Equinox extension mechanism and checked by the platform prior to tool installation. The SDE core also contains a set of Java 5 annotations, which enable tool developers to define their tools and functions without writing any XML. As an ex-

ample, consider Figure 8: On the left-hand side, a tool interface with SDE annotations is shown; on the right-hand side, the corresponding tool view in the SDE is presented.

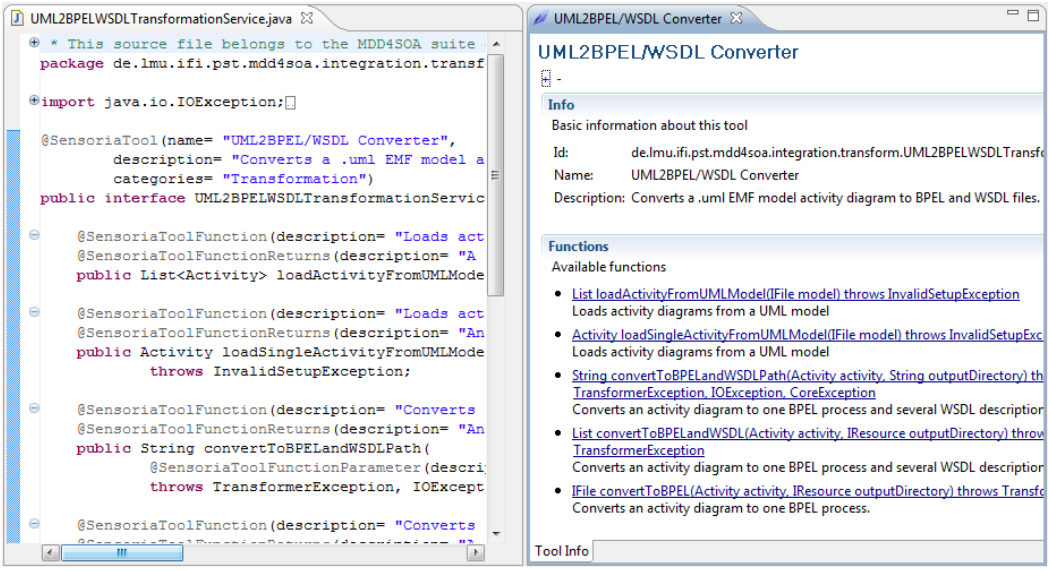


Figure 8: SDE Tool Registration

The API defined within the integration tool service bundle provides access to all installed tools. A tool may use this API to verify installation of required tools, search for tools based on meta-data, and invoke functionality as needed. Therefore, it serves as a discovery service which moderates between the tools. Once the connection has been made, communication between tools is done directly.

3.4 Use of the SDE in ASCENS and further Development

After describing the SDE in the previous section, we will now discuss how it fulfills our requirements for a tool integration platform as laid out in Section 2.

3.4.1 Evaluation of the SDE

Req. 1: Provide an IDE with a Familiar User Interface. The SDE is based on the popular Eclipse development environment which features a user interface that is standard in this field and can be customized by programmers. As such the SDE is no reinvention of the wheel and can be used with little effort by programmers familiar with software development environments and the Eclipse environment in particular. Basing our work on the SDE and Eclipse increases the acceptance of our IDE by potential users.

Req. 2: Support Good Documentation of Tools. The SDE automatically lists installed tools with the interfaces they expose for orchestration (i.e. methods and data types). Programmers can also annotate methods with information about their purpose and the semantics of their parameters which will also be displayed in the tool information provided by the SDE.

Req. 3: Support Tool Categories The SDE has support for tool categories as envisioned by us: Tools are displayed in groups, which allows the user to quickly find tools for a certain task like the verification of models, just to give an example.

Req. 4: Open Source Foundation The SDE is open source software and its code base is available under the *Common Public License (CPL)*. The CPL was published by IBM and has been approved by the Free Software Foundation and the Open Source Initiative. It fits our requirements to allow us to modify the SDE and to redistribute it as an enhanced version named the *SCE Workbench*. For details of the license, we refer to the original license document [IBM09].

Req. 5: Enable Easy Integration of Tools. The SDE enables simple integration of tools, newly developed tools and legacy tools via a declarative and minimally-invasive approach: For legacy tools, wrapper classes must be created that wrap the tool's functionality in an OSGi container, including annotations that allow the SDE to discover the tool and to integrate it into the development environment. Newly developed tools should be written in Java and also need to be packaged in OSGi containers, also utilizing annotations of classes for the integration into the IDE.

Req. 6: Support Broad Range of Tools. The SDE is an open and generic tool integration platform and can thus support all kinds of tools. It also supports the execution of code on remote computers, thereby allowing us to offload computationally intensive tasks to more powerful computers while keeping user interaction intensive tasks on the local machine.

Req. 7: Support Management of Tools. Automatic discovery of tools is supported by the SDE, allowing for simple and straightforward installation of new tools. Installed tools are displayed in a list with additional information about them, also with controls that allow the removal of tools in an easy manner.

Req. 8: Allow for Automatic Updates Automatic updates are supported by the SDE. The user is notified about new versions of a particular tool and does not need to check manually, e.g. by going to the tool's website in regular intervals.

Req. 9: Aid in Development of New Tools. The SDE can be used for Java-based development of tools just like the Eclipse environment. Tools that shall be created in the context of the ASCENS project can therefore utilize the SDE directly, giving programmers easy access to already integrated tools that may aid their software development efforts.

Req. 10: Enable Transformations between Tool Inputs and Outputs. The SDE supports a special class of tools that is responsible for the translation of data from one format to another. Such tools can be used in workflows to connect one tool to another, even if the output data from the first tool cannot be read directly by the second tool.

Req. 11: Support Inter-Tool Dependencies. The SDE supports tools that have external dependencies. The OSGi run-time SDE is based on checks whether these dependencies are satisfied. Furthermore, tools with dependencies on external binaries, e.g. dynamic-link libraries or drivers for legacy tools, can request an automatic installation of these dependencies by implementing a Java interface offered by the SDE extension framework.

Req. 12: Enable Orchestration of Tools. Orchestration via JavaScript is supported by the SDE, which opens powerful opportunities to combine the use of multiple tools in a single workflow. Furthermore, a graphical editor is provided, which allows users to model such workflows as UML activity diagrams, thereby sparing the user from writing JavaScript code and letting him model a workflow in

a more intuitive manner. Simple access to the creation of workflows yields higher acceptance of them and we hope to see more productive use of the SDE as a result.

3.4.2 Summary

As can be seen from the evaluation of the SDE based on the requirements as laid out in Section 2, the SDE is a great basis for tool integration in the ASCENS project. We will extend it to fulfill any new requirements that come up during the project and rebrand it to the new name *SCE Workbench*. We have already distributed the necessary information to our project partners to utilize it for the integration of the ASCENS tools into a single development environment.

In the next section, we will describe tools whose integration is underway and give an outlook on tools whose integration is planned in the next months.

4 Integration of ASCENS tools

In this section of the document, we will present tools (see Table 2) that are currently being integrated into the *SCE Workbench* or are planned to be integrated in the next months. We will describe their purpose and functionality, present usage scenarios and discuss the current state of the integration.

Maude (Section 4.1)	A high-performance, extensible system supporting both equational and rewriting logic specification, programming and analysis. Support for reflection and meta-programming techniques makes it suitable for ASCENS applications.
SAM (Section 4.2)	A tool supporting the stochastic analysis of mobile and distributed systems specified in StoKLAIM.
ARGoS (Section 4.3)	A high-performance and extensible simulator for large swarms of heterogeneous robots.
D-Finder (Section 4.4)	A tool for verifying safety properties of component-based systems described in the BIP language.

Table 2: Tools that are being integrated or are planned to be integrated into the *SCE Workbench*

4.1 Maude

Maude [CDE+07] is a project developed mainly at of Computer Science Laboratory at SRI International and at the Department of Computer Science of the University of Illinois at Urbana-Champaign in collaboration with other academic institutes. It consists of a high-level, declarative language that supports both equational and rewriting logic computation and several tools for analyzing, transforming and executing Maude specifications.

The Maude framework[3] enjoys a couple of features which make it very suitable for the purpose of our project. We mention among others its rule-based language typical of the types of systems under study, the availability of tools for the analysis of Maude specifications, and the flexibility of the Maude language as a semantic framework where to implement other languages. Most notably, Maude comes with an efficient implementation of reflection which enables advanced meta-programming capabilities useful both, for programming autonomic systems and for tool development.

The success of Maude is witnessed by its many applications which includes models of concurrent computation (equational programming, lambda calculi, Petri nets, process algebras, actors), operational semantics of languages (Java, C, Python, Haskell, agent languages, active networks languages, hardware description languages), use as logical framework and metatool (linear logic, translations between theorem provers, type systems, open calculus of constructions, tile logic), models of distributed architectures and components (UML, OCL, MOF, Service architectures and middlewares, open distributed processing), specification and analysis of communication protocols (active networks, wireless sensor networks, firewire leader election protocol), modeling and analysis of security protocols (cryptographic protocol specification language CAPSL, MSR, security specification formalism, Maude-NPA), and specification of real-time, biological and probabilistic systems (real-time Maude, pathway logic, PMaude). Most of these applications exploit Maude support tools for automatic analysis that address aspects such as confluence, termination, sufficient completeness, coherence, reachability, invariants, and temporal logic properties.

[3] Maude website: `http://maude.cs.uiuc.edu/`

Our own experience includes the use of Maude to model and analyze software architectures and their reconfigurations [BLM09, BBGL08], encoding process algebras in graphs [BGL10], modeling the semantics of long-running transactions [BKLSar], verifying systems with evolving topologies with modal logics for graphs [LV11], and implementing and evaluating structured model transformations [BLM11, BLar].

4.1.1 Usage Scenarios and Functionality

We envision various applications of Maude within the ASCENS project. First of all, it can be used to prototype semantic models and be able to execute or check them. Second, Maude can be used as a semantic framework for SCEL dialects, for instance to develop interpreters or analysis tools for SCEL specifications. More trivially, implementing a SCEL semantics can help to spot ambiguities in the formal semantics. Finally, Maude can also be used to model interesting scenarios of the case studies. Indeed, Maude has been promoted as a semantic model for adaptive distributed systems in the last years [MT02], basically due to the suitability of declarative languages for such systems and to the meta-programming features of Maude supported by reflection. A preliminary step in this regard has also been started and consists of a simple model of a morphogenesis scenario of the robot case study.

4.1.2 Integration with other Tools

The Maude framework comes with a couple of built-in tools like a debugger, a reachability analyzer and an LTL model checker. Other tools[4] have been developed which enrich the Maude framework with useful analysis capabilities like theorem proving or linguistic features like real-time and quantitative aspects.

Another interesting tool integration effort is offered by the MOMENT project[5] which provides the so called *Maude Development Tools*, a set of plug-ins to embed the Maude system into the Eclipse environment. These tools comprise two main plug-ins. The first one (Maude Daemon) encapsulates a Maude process into a set of Java classes. It provides two APIs to control the Maude process in batch or interactive mode. It also configures the Maude process according to the user preferences. The second plug-in, called Maude Simple GUI, is a simple IDE to develop Maude programs with a user friendly graphical interface.

A couple of tools use Maude as a back-end for different purposes, for instance to analyze and execute model transformations (MOMENT2, MOMENT-MT), specifications given in architectural languages (MOMENT-AADL), or biological systems (Pathway Logic).

Our own contribution in this regard is a prototypical graphical editor based on Eclipse's EMF technology which allows for a visual specification of Maude modules [Gue11].

4.1.3 SCE Workbench Integration

The integration of Maude into the SCE Workbench is under development. We are investigating various options. On the one hand, we could integrate the Maude Development Tools mentioned above in order to let other tools launch and control Maude process at will. Another option is to integrate our prototypical Maude editor. By the end of the project, we aim to have a Maude-based toolset for the analysis of SCEL applications, whose integration in the SCE Workbench could facilitate its use.

[4] For other tools see `http://maude.cs.uiuc.edu/maude-tools.html` and `http://maude.cs.uiuc.edu/other-tools.html`.

[5] MOMENT website: `http://moment.dsic.upv.es/`

4.2 SAM

SAM (Stochastic Analyser for Mobility)[6] is a *command-line* tool, developed in OCAML, that supports the stochastic analysis of STOKLAIM specifications. STOKLAIM [DKL+06] is the stochastic extension of KLAIM, an experimental language that is 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 0.92". 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.

In [LPT11] SAM has been used to support the analysis of a scenario of the WP7 robotics case study in which three marXbots are in charge of collectively transporting an object to a goal area.

4.2.1 Functionality

SAM can be used for:

- executing interactively specifications;
- simulating stochastic behaviors;
- model checking MOSL formulae.

Running a specification SAM provides an environment for the interactive execution of a STOKLAIM specification. When a specification is executed, a user can select interactively possible computations.

Simulating a specification To analyze the behavior of distributed systems specified in STOKLAIM, SAM provides a simulator. This module randomly generates possible computations. A simulation continues until in the considered computation either a *time limit* or a deadlock configuration is reached.

Given a *sampling time*, each computation is described in the term of the number of resources available in the system during the computation. At the end of a simulation, the average amount of resources available in the system at specified time intervals is provided.

Model checking SAM permits verifying whether a given STOKLAIM specification satisfies a MOSL formula or not. This module relies on an existing state-based stochastic model checker, the Markov Reward Model Checker (MRMC) [KKZ05], that is wrapped in the MOSL model-checking algorithm. After loading a STOKLAIM specification and a MOSL formula, SAM verifies, by means of one or more calls to MRMC, the satisfaction of the formula by the specification.

Unfortunately, even simple STOKLAIM specifications can generate a very large number of states. For this reason, *numerical model checking* cannot always be applied. To overcome the state explosion problem, a *statistical model checker* has also been implemented in SAM. The statistical approach has been successfully used in existing model checkers [HYP06, QS10].

[6] SAM website: `http://rap.dsi.unifi.it/SAM/`

While in a numerical model checker the exact probability to satisfy a path-formula is computed up to a precision ϵ, in a *statistical model checker* the probability associated to a path-formula is determined after a set of independent observations. This algorithm is parameterized with respect to a given *tolerance* ε and *error probability* p. The algorithm guarantees that the difference between the computed values and the exact ones is greater than ε with a probability that is less than p.

4.2.2 SCE Workbench Integration

The integration of SAM into the SCE Workbench is currently under development. The integration procedure has been organized in two phases. The first phase consists in the development of a Java porting of SAM, named JSAM (that is currently under test). As soon as this phase will be completed, the second step, consisting of the concrete integration into the SCE Workbench, will be performed.

4.3 ARGoS

ARGoS is a novel simulator designed by IRIDIA-ULB laboratory within the EU-funded Swarmanoid project[7]. Its design focus is to simulate large heterogeneous swarms of robots and to enable fast prototyping and testing of robot controllers.

4.3.1 Functionality

A simulator is a fundamental tool to support the development of robot behaviors for swarms of robots. A simulator allows for cheaper and faster collection of experimental data, without the risk of damaging the (often expensive) real hardware platforms. In addition, simulated experiments can potentially involve a quantity of robots that would be impossible to manufacture for reasons of cost.

In traditional simulator designs, such as those of Webots [Mic04], USARSim [CLW+07] and Gazebo [KH04], accuracy is the main driver, at the cost of limited scalability. Simulators designed for scalability, such as Stage [Vau08], are focused on very specific application scenarios, thus lacking flexibility. To achieve both scalability and flexibility, in the design of ARGoS we made a number of innovative choices.

ARGoS' architecture is depicted in Figure 9. Its core is the *simulated space*, that contains all the data about the current state of the simulation. Such data is organized into sets of *entities* of different types. Each entity type stores a certain aspect of the simulation. For instance, *positional entities* contain the position and orientation of each object in the space. Entities are also organized into hierarchies. For example, the *embodied entity* is an extension of the *positional entity* that includes a bounding box. Robots are represented as *composable entities*, that is, entities that can contain other entities. Each individual robot feature is stored into dedicated entity types. For instance, each robot possesses an embodied entity and a *controllable entity*, that stores a pointer to that robot's sensors, actuators and control code.

Organizing data in the simulated space in this way provides both scalability and flexibility. Scalability is achieved by organizing entities into type-specific indexes, optimized for speed. For instance, all positional entities are organized into space hashes, a simple and state-of-art technique to store and retrieve spatial data. Flexibility is ensured because entities are implemented as modules. In addition to the entities offered natively by ARGoS, the user can add custom modules, thus enriching ARGoS' capabilities with novel features.

Analogously, the code accessing the simulated space is organized into several modules. Each individual module can be overridden by the user whenever necessary, thus ensuring a high level of flexibility. The modules are implemented as plug-ins that are loaded at run-time.

[7] Swarmanoid website: `http://www.swarmanoid.org/`

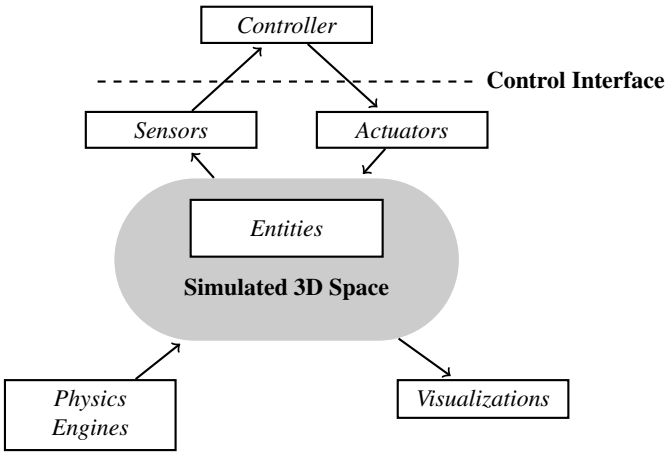
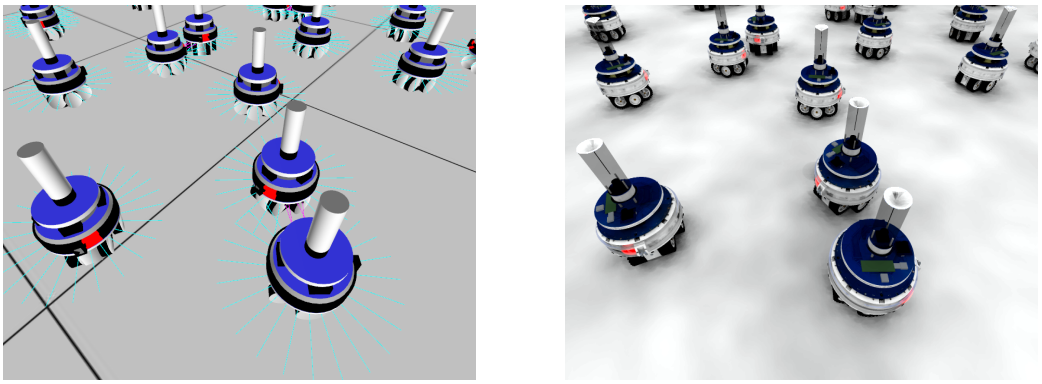


Figure 9: The architecture of the ARGoS simulator.

Controllers are modules that contain control code developed by the user. Controllers interact with a robot's devices through an API called the *common interface*. The common interface API is an abstraction layer that can make underlying calls to either a simulated or a real-world robot. In this way, controllers can be seamlessly ported from simulation to reality and back, making behavior development and its experimental validation more efficient.

Sensors and *actuators* are modules that implement the common interface to mediate between controllers and the simulated space. Sensors read from the simulated space and actuators write on it. The optimized entity indexes ensure fast data access. For each sensor/actuator type, multiple implementations are possible, corresponding to models that differ in computational cost, accuracy and realism. In addition, sensors and actuators are tightly coupled with robot component entities. For instance, the foot-bot wheel actuator writes into the *wheeled equipped entity* component of the foot-bot. Such coupling greatly enhances code reuse. New robots can be inserted by combining existing entities, and the sensors/actuators depending on them work without modification.

Visualizations read the simulated space to output a representation of it. Currently, ARGoS offers three types of visualization: (i) an interactive GUI based on Qt and OpenGL, (ii) a high quality off-line 3D renderer based on POV-Ray, and (iii) a textual renderer designed to interact with data analysis and plotting software such as Matlab and GNUPlot. Figure 10 shows some exemplary outputs of the first two visualization engines.



(a) (b)

Figure 10: Screen-shots from different visualizations. (a) Qt-OpenGL; (b) POV-Ray.

One of the most distinctive features of ARGoS is that the simulated space and the physics engine are separate concepts. The link between them is the embodied entity, which is stored in the simulated space and updated by a physics engine. In ARGoS, multiple physics engines can be used simultaneously. In practice, this is obtained by assigning sets of embodied entities to different physics engines. The assignment can be done in two complementary ways: (i) manually, by binding directly an entity to an engine, or (ii) automatically, by assigning a portion of space to the physics engine, so that every entity entering that portion is updated by the corresponding engine. *Physics engines* are a further type of module. Currently, three physics engines are available: (i) a 3D dynamics engine based on the ODE library, (ii) a 2D dynamics engine based on the Chipmunk library, and (iii) a custom-made 2D kinematic engine.

To further enhance scalability, the architecture of ARGoS is multi-threaded. The simulation loop is designed in such a way that race conditions are avoided and that CPU usage is optimized. The parallelization of the calculations of sensors/actuators and of the physics engines provides high levels of scalability. Results reported in [PTO+] show that ARGoS can simulate 10 000 simple robots 40% faster than real time.

ARGoS has been released as open source software[8] and currently runs on Linux and MacOSX.

4.3.2 SCE Workbench Integration

The integration of ARGoS into the SCE Workbench is currently under development. ARGoS is a native tool and, in principle, different integration strategies are possible. However, those strategies that involve interfacing the ARGoS API to Java directly are not viable. In fact, porting the common control interface to Java would force us to include the Java virtual machine into the robots, with negative effects on performance and memory usage.

Therefore, we will make ARGoS into a remotely controllable service in the style of a web service, but with ad-hoc, optimized communication mechanisms that are currently under study.

4.4 D-Finder

On exploring the current state of the art in formal verification, it becomes clear that a formal verification method needs to address the following problems:

- **Scalability** that avoids the state space explosion problem and therefore allows increasing the size of systems to be verified.
- **Effectiveness** that permits to detect errors of systems' models in the design phase as early as possible.
- **Incrementality** that integrates verification into the design process.
- **Compositionality** that allows inferring global properties of a system from the known local properties of its sub-systems.

We have implemented the compositional and incremental methods in *D-Finder*.

4.4.1 Functionality

D-Finder is a tool for verifying safety properties, especially for checking deadlock-freedom of component-based systems described in the BIP language encompassing multi-party interaction. For deadlock

[8] ARGoS website: `http://iridia.ulb.ac.be/argos/`

detection, D-Finder applies proof strategies to eliminate potential deadlocks by computing increasingly stronger invariants.

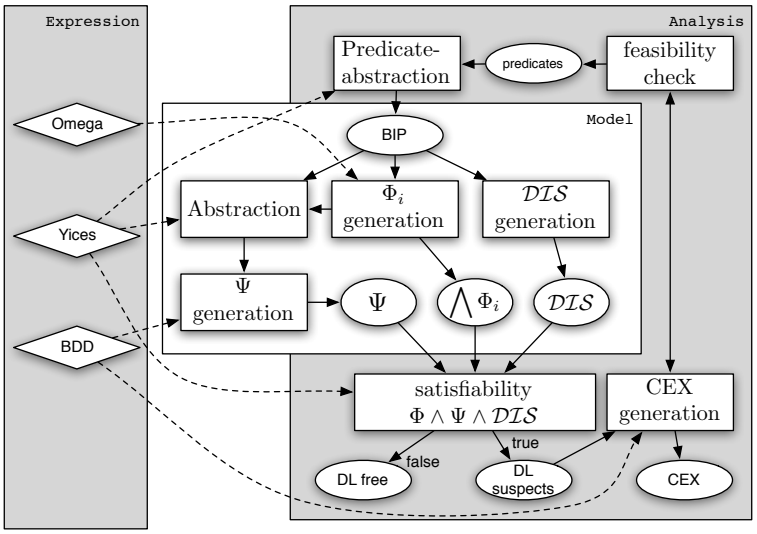


Figure 11: Structure of the D-Finder tool

D-Finder is written in Java and uses external tools and native code via the Java Native Interface (JNI) for computations. Figure 11 gives an overview of the main modules of the tool. The *Model* block handles the parsing of the *BIP* code into an internal model and provides the means to compute component invariants Φ, interaction invariants Ψ, and a set of deadlock states $\mathcal{DIS}$. The results from *Model* are used from various implementations of the *Analysis* block, which perform further steps like the generation of possible deadlocks and, most recently, the generation of counter examples for Boolean systems (CEX). The *Expression* block is used by both *Model* and *Analysis*. Its main purpose is to provide a uniform interface for different back-ends that store the actual expressions. For algorithms on Boolean variables, like computation of interaction invariants Ψ, a more succinct implementation with BDDs as back-end is used, while large systems that incorporate non-Boolean data require to directly create and maintain input files for an SMT solver on disk. The main features of *D-Finder* are as follows:

- **Compositional verification**: For a given safety property, *D-Finder* iteratively conjoins the predicate characterizing violations of the property (set of bad states) with an over-approximation of the set of reachable states. If the conjunction is false, then the property is guaranteed. Otherwise, there is a (bad) state within the approximation that violates the property.

- **Efficient computation of invariants**: The over-approximation is the conjunction of two kinds of invariants, component invariants (Φ_i) and interaction invariants (Ψ). We provide efficient methods implemented in *D-Finder* for computing the two kinds of invariants. Φ_i express local constraints of atomic components. Ψ characterize constraints on the global state space induced by synchronizations between components.

- **Checking reachability**: To eliminate remaining false positives, a compositional abstraction refinement approach is developed. It is based on an abstraction of the components to Boolean systems and subsequent generation of inductive invariants and pre-image computation. Result of this final step of verification is an error trace that proves and demonstrates the reachability of a found bad state.

- **Incremental verification**: an incremental verification method has also been implemented in *D-Finder*. We study rules which allow preserving established invariants during the incremental construction. For the general case where a system might not satisfy these rules, we propose methods for computing incrementally invariants of the entire system from the established invariants of its constituents.

The experimental results on large-scale and complex systems show the efficiency of *D-Finder*. More details can be found at `http://www-verimag.imag.fr/dfinder`.

4.4.2 SCE Workbench Integration

D-Finder is developed in Java using the Eclipse IDE. Consequently, its integration into the SCE Workbench will be straightforward.

5 First Year Resume and Outlook

In the first year, we have focused on selecting a foundation for our work of integrating ASCENS tools into an IDE and providing the partners in the project with information about how they can perform the integration. We based our selection of the SDE from the SENSORIA project on a collection of requirements we assembled in internal discussions about the features expected from an IDE and the associated integration processes. In Section 3.4.1 we described how the SDE fulfills them and argued that it will serve as a good foundation for the *SCE Workbench*, our ASCENS tool integration platform.

In the course of the first year, we provided project partners with the necessary information for the integration of their tools into the *SCE Workbench*, e.g. in the form of written documentation, screencasts, but also live-demos and talks at the kick-off meeting held in Munich and the general meetings held in Pisa and Grenoble.

As of now, the project partners, which are actively working on an integration of their tools or are looking at the *SCE Workbench* for tool integration in the next months, have not requested any additional functionality. For the time being, we will therefore continue to focus on disseminating the necessary information about tool integration and use of the *SCE Workbench* and providing support for tool integration projects that are underway.

Once the integration of tools reaches a level where orchestration of tools and the building of tool chains is feasible, we will shift our focus towards tighter integration of the tools.

References

[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.

[BBGL08] Antonio Bucchiarone, Roberto Bruni, Stefania Gnesi, and Alberto Lluch Lafuente. Graph-based design and analysis of dynamic software architectures. In *Concurrency, Graph and Models. Festschrift in honor of Ugo Montanari*, volume 5065 of *LNCS*, pages 37–56. Springer Verlag, 2008.

[BGL10] Roberto Bruni, Fabio Gadducci, and Alberto Lluch Lafuente. An algebra of hierarchical graphs and its application to structural encoding. *Sci. Ann. Comp. Sci.*, 20:53–96, 2010.

[BK11] Marianne Busch and Nora Koch. NESSoS deliverable D2.2: First release of the SDE for security-related tools, 2011.

[BKH] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. pages 146–162.

[BKLSar] Roberto Bruni, Anne Kersten, Ivan Lanese, and Giorgio Spagnolo. A new strategy for distributed compensations with interruption in long-running transactions. In *Proceedings of the 20th International Workshop on Algebraic Development Techniques (WADT 2010)*, LNCS. Springer Verlag, to appear.

[BLM09] Roberto Bruni, Alberto Lluch Lafuente, and Ugo Montanari. Hierarchical design rewriting with Maude. In Grigore Rosu, editor, *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, volume 238 (3) of *Electronic Notes in Theoretical Computer Science*, pages 45–62. Elsevier, 2009.

[BLM11] Roberto Bruni, Alberto Lluch Lafuente, and Ugo Montanari. On structured model-driven transformations. *International Journal of Software and Informatics*, 5(1-2):185–206, 2011.

[BLar] Roberto Bruni and Alberto Lluch Lafuente. Evaluating the performance of model-transformation styles with Maude. In *Proceedings of the 8th Symposium on Formal Aspects of Component Software (FACS'11)*, LNCS. Springer Verlag, to appear.

[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.

[CLW+07] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. USARSim: a robot simulator for research and education. In *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*, pages 1400–1405. IEEE Press, Piscataway, NJ, 2007.

[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.

[Ecl11] Eclipse Foundation. The Eclipse open source community and Java IDE, 2011.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[Fou09] Eclipse Foundation. Eclipse Equinox - Implementation of the OSGi R4 core framework specification, 2009.

[Gue11] Irena Gueorguieva. Sviluppo di un'interfaccia grafica per modelli gerarchici in maude. Bachelor Thesis, Computer Science Department, University of Pisa, 2011. In Italian.

[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.

[IBM09] IBM. Common Public License (CPL) version 1.0, 2009.

[KH04] Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2149–2154. IEEE Press, Piscataway, NJ, 2004.

[KKZ05] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244. IEEE CS Press, 2005.

[LPT11] Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A WP7 robotics scenario in Klaim. Avaiable in the ASCENS Wiki, 2011.

[LV11] Alberto Lluch Lafuente and Andrea Vandin. Towards a Maude tool for model checking temporal graph properties. In Fabio Gadducci and Leonardo Mariani, editors, *Proceedings of the 10th International Workshop on Graph Transformation and Visual Modelling Languages (GT-VMT'11)*. ECEAAST, 2011. To appear.

[Mic04] Olivier Michel. Cyberbotics Ltd. – Webots: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, March 2004.

[MR10] Philip Mayer and Istvan Rath. D7.4d: Report on the Sensoria Development Environment (SDE), third version. Deliverable for the EU project SENSORIA, reporting period October 2008 - February 2010, SENSORIA Project 016004. 2010.

[MT02] Jos Meseguer and Carolyn Talcott. Semantic models for distributed object reflection. In Boris Magnusson, editor, *16th European Conference on Object-Oriented Programming (ECOOP'02)*, volume 2374 of *LNCS*, pages 1–36. Springer Verlag, 2002.

[PTO$^+$] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Timothy Stirling, Álvaro Gutiérrez, Luca Maria Gambardella, and Marco Dorigo. ARGoS: a modular, multi-engine simulator for heterogeneous swarm robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*. In press.

[QS10] Paola Quaglia and Stefano Schivo. Approximate model checking of stochastic COWS. In *Proc. of TGC 2010*. To appear., 2010.

[RAR07] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 1–20, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[The07] The OSGi Alliance. OSGi service platform core specification, release 4.1, 2007.

[Vau08] Richard Vaughan. Massively multi-robot simulation in Stage. *Swarm Intelligence*, 2(2):189–208, 2008.

[WBF+08] Martin Wirsing, Laura Bocchi, Jose Luiz Fiadeiro, Stephen Gilmore, Matthias Hoelzl, Nora Koch, Philip Mayer, Rosario Pugliese, and Andreas Schroeder. *Sensoria: Engineering for Service-Oriented Overlay Computers*. MIT Press 2008, 2008.

[WCL+05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more*. Prentice Hall PTR, march 2005.