Project No: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Models**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

**Future and Emerging Technologies**

**Deliverable D1.3**

**Analysis**

Due date of deliverable: (T0+42)

Actual submission date: 1. September 2012

Start date of the project: **1st March 2009**

Duration: **48 months**

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

Final version

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
<th>Public</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP</td>
<td>Restricted to other programme participants (including Commission Services)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including Commission Services)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including Commission Services)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary:
Analysis

This document summarizes deliverable D1.3 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at http://www.hats-project.eu.

The deliverable reports on the results achieved in Task 1.3 “Analysis” which was concerned with the development and documentation of a workflow driven, coherent user interface as well as the design of parametrizable analyzes. The document is composed of the user interface guideline documentation which is designed as a living document to be updated regularly and the presentation of a pluggable type checking extension that introduces location types and features an innovative user interface solution to present inferred types. Finally, we present a framework that provides a common fundament for the development of ABS language extensions, analyses and compilers. The framework allows in particular rapid prototyping of tools.

List of Authors

Richard Bubel (TUD)
Frank de Boer (CWI)
Dilian Gurov (KTH)
Stijn de Gouw (CWI)
Yannick Welsch (UKL)
# Contents

1 Introduction  
1.1 General Background .................................................. 4  
1.2 List of Papers ......................................................... 5  

2 A Workflow Driven User Interface  
2.1 Motivation ............................................................... 6  
2.2 General User Interface Elements .................................... 7  
2.2.1 Style Elements .................................................... 7  
2.3 Workflow Driven User Interface ................................... 7  
2.3.1 Tool Preferences and Configuration ........................... 7  
2.3.2 Guided Configuration and Element Creation: Wizards .... 10  
2.3.3 Views and Perspectives .......................................... 11  
2.4 Providing User-Feedback ............................................ 15  
2.4.1 Reporting Errors and Warnings ................................. 16  
2.4.2 Reporting Additional Information ............................. 17  
2.5 Conclusions .............................................................. 18  

3 Location Types  
3.1 Introduction ............................................................. 19  
3.2 Example ....................................................................... 20  
3.2.1 Location types ....................................................... 20  
3.2.2 Concurrent object groups ........................................ 20  
3.2.3 Using location types ............................................. 21  
3.3 Results ...................................................................... 23  

4 Meta-Programming  
4.1 Code Analysis ........................................................... 24  
4.2 Code Transformation .................................................. 27  
4.3 Code Generation ....................................................... 27  

5 Sound Control-Flow Graph Extraction ................................ 29  

6 Conclusion ................................................................. 33  

Bibliography ................................................................. 33  

Glossary ........................................................................... 36
Chapter 1

Introduction

1.1 General Background

In this deliverable we report on the main results of Task 1.3 “Analysis”. The task was concerned with the development of a workflow driven user interface and the documentation of user interface guidelines. The intention was to achieve a seamless integration of the plethora of different analyses into the ABS Integrated Development Environment (ABS IDE). As well as to collect, consolidate and document the user interface integration efforts for the different tools. The second task objective was to investigate techniques that allow to realize pluggable, parametrizable analyses which can be combined in a modular manner.

Concerning the first objective: Characteristic for a workflow driven user interface is to support the ABS developer during the whole development process by adapting to the current phase in the development within the same IDE. The development workflow of ABS (and also other modeling/programming languages) consists of a number of distinguishable phases which can be fundamentally different in nature. Usually these phases share only a few commonalities, while requiring all their own set of specialized capabilities. The requirements imposed on the ABS IDE are quite different, for instance: if the user actually develops an ABS model, then the IDE should be optimized for text editing, lookup of library functions and similar, while e.g., if the ABS model is currently debugged, tested or undergoes advanced static analyses other aspects like a variable value inspector should be enabled and emphasized. Each of these phases demand a specialized user interface providing optimal support for the workflow required in the current context.

Designing a workflow driven interface requires also to classify the tools contribution to the overall development methodology in conceptual terms that relate to the covered domain. This allows to organize the environment around task related notions. For instance, the developer simply configures the analyses she wants to be performed, but does not need to care about the underlying tools that execute the analyses.

A major part of the work in this task was distributed among several participants and put into the actual realization of the user interfaces of the tools developed by the respective partner. This work is reflected in the user guidelines chapter (Chapter 2) of the deliverable. The user interface guidelines summarize the made design decisions and explain the different graphical user interface elements.

In fact, the user guidelines serve several purposes: They introduce the different user interface elements and explain their usage by example. This allows developers to get an overview about the possibilities of the ABS IDE and ideas how to realize best the user interface of their own tool. A systematic and thought through design of a tools’ user interface also helps to achieve a better understanding where to place the tool within the HATS methodology and for which phases of the workflow the usage of (certain configurations of) the tool is sensible and can make the most impact. In addition, one has to clarify how to present the results of the tool to the user and which graphical user interface components may serve best for this purpose. Knowing which components already exist and which can possibly be reused, reduces the required amount of development effort significantly, leading to a vastly improved presentation of the tools’ results which goes beyond a simple textual console output. A different purpose of the guidelines is to serve as a reference manual that can be used to lookup the name and place of graphical style elements like icons. Beyond the
obviously saved work in creating new icons for each tool (or using overly generic ones) by encouraging reuse of available icons, they also provide an insight about the composition principles used to create new icon variants from already existing style elements. This contributes directly to a coherent appearance of the ABS IDE and avoids, in particular, the use of different icons for same conceptual elements.

The second objective of the task concerning modular, parametrizable analyses techniques is addressed by the remaining chapters.

In Chapter 3 we describe a modular extension to the pluggable type checking framework developed for the ABS tool suite. The presented extension to the ABS type system allows to annotate variable declarations with location types. Location types restrict the set of objects that can be referenced by the declared variable by the location in which they live, for instance, they require that objects referenced by a certain variable are always member of the same concurrent object group (COG). The presented type system for location types is a showcase for and realized as a plugin for the exiting ABS type checker. The location type checking plugin allows also to infer location types for not annotated variables and presents the inferred location type information non-intrusively as an overlay directly attached to the variable declaration in the ABS source code editor.

In Chapter 4 we present a development framework that allows to extend ABS easily with domain-specific language extensions incl. built-in support for syntax-highlighting and error reporting. The framework supports further the implementation of complex analyses with a minimum of code and facilitates in particular rapid prototyping of new analyses or backends for ABS. It is even flexible enough to allow to extend and parametrize existing analyses in an aspect-oriented manner by transparently weaving additional code into the ABS model. This is achieved by deep and thorough integration of ABS support into the meta-programming language Rascal. The presented development framework will provide a common technological basis for the implementation of ABS backends and analysis tools.

In Chapter 5 we present a call graph extraction algorithm that has been proven sound. The algorithm itself is for Java and can be directly applied to ABS using the ABS Java backend. I can also be implemented for ABS using the Rascal framework presented in Chapter 4.

1.2 List of Papers

This working task was mostly concerned with the user interface part of the ABS tools and the documentation of the design decisions as well as with the implementation of a meta-programming framework for ABS. The technical papers on the tools explaining the background of the analyses techniques are described in other deliverables of the HATS project and, thus, not listed here.

Nevertheless and as requested by the reviewers, the papers are not directly attached to Deliverable D1.3, but are made available on the HATS web site at the following url: http://www.hats-project.eu/sites/default/files/D1.3. Direct links are also provided for each paper listed below.

Paper 1: Location Types for Safe Distributed Object-Oriented Programming

This paper presents location types, which statically distinguish far from near references. It presents a formal type system for a minimal core language and, in addition, a type inference system that gives optimal solutions. The location types are implemented as a pluggable type system for the ABS language, an object-oriented language with a concurrency model based on concurrent object groups. An important contribution of this paper is the combination of the type system with the flexible inference system and a novel integration into an Eclipse-based IDE by presenting the inference results as overlays. This drastically reduces the annotation overhead while providing full static type information to the user. The IDE integration is a general approach of its own and can be applied to many other type system extensions. This paper was written by Yannick Welsch and Jan Schäfer, and was published in the proceedings of TOOLS 2011.

Download Paper 1
Chapter 2

A Workflow Driven User Interface

2.1 Motivation

In this section we present a set of guidelines to ensure a coherent and workflow driven user interface for the HATS tool suite. We describe the different elements of the HATS user interface and provide advice where and how to integrate the different components of an ABS tool in an organized and systematic manner. We discuss also different possibilities to provide feedback for the user within the GUI non-intrusively, and yet, meaningful and task supporting.

HATS features currently three different types of user interfaces:

• command line interface
• integration into the GNU Emacs editor, and
• integration into the Eclipse environment

Each tool of the HATS tool suite should have a command line (or standalone) user interface. This decision was taken deliberately. The intention has been to enforce a strict separation of functionality and graphical user interface from the start on. This ensured that virtually all ABS tools can be fully integrated in any integrated development environment and that the tool’s functionality can be reused by other tools. An example for the latter are the tools for parsing and compilation of Core ABS models or those responsible for product selection and product generation where Full ABS models are flattened to pure Core ABS models by delta-application.

To accommodate developers used to the GNU Emacs editor there exists an integration of (some of the) ABS tools into Emacs. The integration has been mainly used for work related to the Maude backend in particular during the development and integration of ABS deployment components. The GNU Emacs integration features syntax highlighting of ABS source code and allows experienced Emacs users to quickly edit, compile and simulate the ABS model without having to start up a full fledged IDE like Eclipse.

The actual ABS Integrated Development Environment (ABS IDE) is build upon the Eclipse rich client platform and realized as a feature project encompassing a collection of plugins. The ABS IDE is the main graphical user interface presented to users of the ABS language and toolkit. For the majority of all developers the ABS IDE establishes the first contact to the ABS language and the accompanying tool support. Using Eclipse allows to leverage the powerful capabilities of an industrial strength development environment to support the development of ABS models and lower the burden of getting used to yet another IDE.

The tools have been (and still are) produced in different tasks of the HATS project starting with Task 1.1 which provided a first version of the ABS tool suite. In this section we report on the efforts to achieve a uniform user interface and to enable a smooth and consistent, user friendly experience when working in the ABS world.

The user interface guidelines categorize systematically the developed user interface elements and their usage. Different concepts are explained by example using one or more selected representatives of the ABS
tool suite to highlight and discuss the respective user interface feature. These explanations shall provide a concrete orientation for tool developers on how to integrate new tools or extensions to existing tools in the ABS tool suite in a coherent, workflow driven and user friendly manner.

The main objective is to avoid ending up with an overloaded and counter-intuitive user interface working against the user and not for the user. This is in particular important to ensure a good user experience amid an ever increasing suite of available ABS tools. As discussed above the Eclipse-based ABS IDE is the main development environment for ABS and the user interface guidelines are targeted to the integration of tools within the ABS IDE.

The structure of this Chapter is as follows: In Section 2.2 we describe briefly the current style elements of the GUI like icons and their meaning. In Section 2.3 we describe the possibilities Eclipse offers to organize the workflow and how they should be used by ABS tools. Finally, we describe in Section 2.4 the different available mechanisms to provide feedback to users in a timely and non-intrusive manner.

### 2.2 General User Interface Elements

In this section we list all style elements (icons, graphics etc.) used in the current ABS tool suite and describe briefly their intended meaning and usage. These style elements should be reused by all HATS user interfaces as far as possible. As a rule of thumb, new graphical elements should only be added if

- a category is not yet covered by a given element or
- a new subcategory containing more than one element should be introduced. If the subcategory is a specialization of the same element type (e.g., COG class instead of a not annotated class) of the parent category, the new icon should resemble the parent category icon.

The styling and color code used in new graphical elements should follow the HATS logo and HATS web colors as shown in Fig. 2.1.

#### 2.2.1 Style Elements

Table 2.1 lists all icons that depict concrete ABS source elements. These icons should be reused by all views needing to present an ABS language element. Table 2.2 shows all icons that are used by the ABS Model Explorer (similar to the Eclipse Package Explorer) a central view allowing to navigate within and between ABS projects. Finally, the icons used in the global Eclipse toolbar are explained in Table 2.3.

One additional remark on the composition of items: Introduction of new icons (or icon variants) for closely related concepts should be done with care. For instance, Table 2.1 lists three icons for classes: normal classes, classes with the type annotation [COG] as well as for classes with the type annotation [Plain]. As COGs are an elementary concept for ABS the usage of specialized icons for these purposes is justified, but note that adding icons for other type annotations can lead to a proliferation of icons as one would have to multiply out all possible combinations. This restriction holds only for the global icon set, specialized views might instead focus on different type annotations and bring their own icon set.

### 2.3 Workflow Driven User Interface

#### 2.3.1 Tool Preferences and Configuration

ABS tools are configurable and the question arises where to place their different configurations. Eclipse provides basically three default places where the user expects to find configuration options. These are:

**Eclipse Preferences Page** The preferences page provides configuration options that effect the global Eclipse setup and appearance. Tool settings that are not project specific should be configurable here. These settings are then used as default settings when creating a new ABS project. They might
Figure 2.1: HATS project color code

(a) ABS Preferences

(b) ABS Project Properties

Figure 2.2: ABS Preferences & ABS Project Properties
### Table 2.1: Icons: ABS Source Code Elements

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Package</td>
<td>Depicts a single ABS package</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Packages</td>
<td>Depicts a collection of ABS packages</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Module</td>
<td>Depicts a single module containing elements</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Module Empty</td>
<td>Depicts a single empty module</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Import</td>
<td>Depicts a single ABS import declaration</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Imports</td>
<td>Depicts a group of ABS import declarations</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Export</td>
<td>Depicts a single export declaration</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Exports</td>
<td>Depicts a group of ABS export declarations</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Datatype</td>
<td>Depicts an ABS datatype declaration</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Constructor</td>
<td>Depicts a constructor of an ABS datatype</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Function</td>
<td>Depicts an ABS function declaration/definition</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Interface</td>
<td>Depicts an ABS interface</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Class</td>
<td>Depicts an unannotated/normal ABS class</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Plain Class</td>
<td>Depicts a [Plain] annotated ABS class, i.e., a class that is always instantiated in the current COG</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Cog Class (Annotated)</td>
<td>Depicts a [COG] annotated class, i.e., a class which is always instantiated in a new COG</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Field</td>
<td>Depicts an ABS object field</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Main Block</td>
<td>Depicts the main block of an ABS module</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Method</td>
<td>Depicts a method declaration/implementation</td>
</tr>
<tr>
<td>![Δ]</td>
<td>- (unicode character)</td>
<td>Depicts an ABS delta definition</td>
</tr>
<tr>
<td>![PL]</td>
<td>- (characters)</td>
<td>Depicts an ABS product line description</td>
</tr>
<tr>
<td>![Π]</td>
<td>- (unicode character)</td>
<td>Depicts an ABS product configuration</td>
</tr>
</tbody>
</table>

Table 2.2: Icons: ABS Navigator Elements

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Navigator</td>
<td>Depicts the ABS Model Explorer view</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Navigator Folder</td>
<td>Depicts a folder in the ABS Model Explorer view</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS project</td>
<td>Depicts an ABS Project</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>ABS Logo</td>
<td>General Use ABS Logo</td>
</tr>
</tbody>
</table>

Table 2.2: Icons: ABS Navigator Elements
HATS Deliverable D1.3

<table>
<thead>
<tr>
<th>Icon</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="icon" /></td>
<td>Duke HATS (small)</td>
<td>Compile &amp; Run (using Java Backend)</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>Duke HATS UML (small)</td>
<td>Compile, Run &amp; Create Sequence Diagram (Using Java Backend &amp; SEdit)</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>Duke HATS Debug (small)</td>
<td>Compile &amp; Debug (using Java Backend)</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>Maude Compile</td>
<td>Compile &amp; Run (using Maude Backend)</td>
</tr>
<tr>
<td><img src="image" alt="icon" /></td>
<td>Maude Debug</td>
<td>Compile &amp; Debug (using Maude Backend)</td>
</tr>
</tbody>
</table>

Table 2.3: Icons: ABS Toolbar (global)

be overwritten by project specific settings which are configurable in the project properties dialog. Figure 2.2(a) shows the Eclipse preferences page. It contains a specific category ABS where all ABS specific tool configurations should be placed.

**Eclipse Project Properties** The project properties dialog allows to overwrite the default tool settings made in the preferences page for a specific project. It allows also to configure project specific settings like build paths and similar. Figure 2.2(b) shows the project properties dialog with the ABS category opened.

**Launch Configurations** Launch configurations (see Figure 2.3) can be accessed by the previous two configuration pages via the Run/Debug settings category. They allow to configure the runtime/simulation environment for an ABS model. Launch configurations allow to configure how and where an ABS model is run/simulated. For instance, there exist launch configurations for each backend (Java, Maude & Scala). These general launch configurations serve as a template and are instantiated further to serve different purposes, e.g. for running, debugging or testing the ABS model. Project specific instances of these configurations are created automatically and by the user via the project properties settings.

When adding a new tool or new configuration options for existing tools and analyses, the existing categories should be reused as far as possible. To facilitate reuse of configuration categories it should be avoided to name them after the underlying tool. Instead notions describing their conceptual commonalities should be used. This adds towards a workflow driven user interface which puts the users domain knowledge in the center.

For instance, instead of introducing configuration categories called COSTA and KeY (as top level categories), a parent category Analyses should be introduced which can be subdivided into Resources, Security, Dead-Lock and Functional Correctness. This allows to configure the kind of analyses to be performed without the need for the user to map them to specific tools. Configuration categories named after tools should only be used for truly tool specific options that do not make sense in any other context.

### 2.3.2 Guided Configuration and Element Creation: Wizards

Wizards guide the user through complex configuration processes by presenting a sequence of configuration pages presented in a dialog window. The dialog queries the user step-by-step for the necessary information to achieve the desired outcome.

The typical structure of a wizard consists of a dialog window featuring two main compartments:

- the configuration page compartment takes most of the dialog’s layout space and gathers the necessary information to setup an analysis or create a new element; a wizard can have any number of configuration pages which are displayed in a fixed, given order.
the button compartment contains the buttons Back, Next, Cancel and Finish (the order is fixed w.r.t. the user interface guidelines of the underlying platform) and is placed at the bottom of the dialog.

The buttons are enabled or disabled depending on which steps are possible at the present situation. The Cancel button has to be enabled all the time. The Finish button may already be active before the last step is reached, e.g., when the additional information to be filled in on the next configuration pages, is not strictly necessary or reasonable default values exist.

Tools added to the ABS environment should offer wizards for all actions that require or offer a complex configuration. The different configuration pages should be clearly structured and ordered with respect to importance. In particular, optional configuration pages should come last to avoid forcing the user to click through all pages and instead to be able to activate the Finish button already at earlier stages.

ABS provides a collection of wizards most notably the wizard for creating a new ABS project or specific ABS source elements like classes or interfaces. Figure 2.4(a) shows the list of all currently existing new element creation wizards supporting the user in the creation of ABS projects (Figure 2.4(b)) and ABS classes (Figure 2.4(c)).

2.3.3 Views and Perspectives

In Eclipse (and, hence, the ABS IDE) additional functionality is provided by plugins which add new controls, editors and views to the platform. As the name suggests, views provide a specialized view on certain aspects of the ABS model. Typical representatives of views are the ABS Model Explorer, the ABS Outline view or the Problems view. Views can be enabled or disabled at any time by the user and freely placed in the IDE. While the latter feature allows the developer to adapt the IDE according to her own taste and needs, it is not enough to achieve a seamless developing experience and workflow.

Consider a typical workflow scenario like the following:

1. design and implementation of a (part of) an ABS model,
(a) List of available wizards for new element creation

(b) Project Creation Wizard

(c) Class Creation Wizard

Figure 2.4: Guided creation and configuration: Wizards
2. testing/simulating the ABS model and, possibly,
3. debugging the ABS model.

The three phases of this small workflow scenario are quite different in nature and there is no common setup of the ABS IDE (or any IDE for that matter) that can support the different phases simultaneously. Depending on the current task at hand, the developer is either mostly editing the ABS model and, hence, the ABS source code editor and editing controls should be emphasized by the IDE. In contrast, for the debugging phase easy access to the program execution controls of the debugger and a possibility to inspect the current state of the ABS simulation gain importance.

To address this issue the ABS IDE makes use of the possibility to define perspectives in Eclipse which allow to configure the appearance of the IDE with respect to and specifically tailored towards the current task. Basically, a perspective consists of a preconfigured collection of views, editors and controls as well as their placement on the screen. The selection of enabled views and their placement within the IDE is tailored towards a specific purpose like modeling, debugging or simulation emphasizing important aspects and hiding unnecessary details.

Views, editors and controls can be (and often are) part of more than one perspective. Although perspectives can be created and customized by the users themselves, it is important to provide well-structured and thought through default perspectives for the different tasks.

To give an idea on how to design useful perspectives we describe selected examples in brief. We show an example of each of an ABS perspective for development and debugging of ABS models.

The ABS Modeling Perspective

The ABS modeling perspective (see Figure 2.5) is intended to be used in workflow phases that are mainly concerned with the actual development of the ABS model. In these phases editing the source code files makes up most of the actual workload. Hence, the ABS modeling perspective places the ABS source code editor as central element in the middle of the IDE and assigns it the largest amount of space. When writing ABS source code, typical other actions needed to be carried out by the developer involve the creation of and navigation between different ABS modules, classes etc., available in different source files of the same or even other projects. The ABS Model Explorer view supports the user in performing the necessary actions by listing all ABS projects, ABS modules and classes in a hierarchical tree structure. The tree structure allows for easy access and navigation between the different elements and can be linked to the editor such that selecting an item in the tree opens the corresponding file in the editor. The ABS modeling perspective places the Model Explorer view therefore to the left of the source code editor. This allows for short mouse movements and provides quick and easy access for cultures using a left-to-right (LTR) text direction (for cultures with a right-to-left text direction the placement needs to be changed).

By default, the ABS Outline view is placed in the right column of the ABS IDE next to the source code editor. The Outline view provides the user with a lean and structured overview of all ABS source elements declared within the currently opened file. The Outline view features a hierarchical list of the declared datatypes, functions, interfaces, classes, fields and methods. Selecting an element in the Outline view brings the selected element in the center of the ABS editor for closer inspection.

The bottom compartment of the IDE contains a collection of views that are generally useful and needed regularly in the given context, but not as frequently as the others. The arguably most important view resting in the bottom part is the Problems view (described later in this document) which lists all errors or warnings present in the currently opened ABS projects.

The ABS Debug Perspective

In contrast to the ABS Debug perspective adds a number of new views, hides and rearranges the remaining views. We explain a few more details along Figure 2.6 which shows the default layout of the ABS Debug perspective.
As can be seen immediately the ABS source code editor is also part of the ABS Debug perspective, although its role is less prominent than in the previous ABS modeling perspective. The editor is now placed below a new view—the ABS Variable Inspector. The variable inspector allows to examine the state of all concurrent object groups (COGs) and their current members during the ABS simulation.

The second new view is the debugger’s control view (named ABS Debug identical to the perspective’s name). It occupies the whole left column of the ABS IDE and provides the necessary means to interact with the ABS debugger. In the view’s own toolbar, it offers access to the debugging controls for step next, step into or step-to-return. The icon at the left end of the view’s toolbar allows to change the scheduling strategy of the simulation environment.

Additionally, the ABS Debug view provides a concise overview of the current simulation state. In contrast to the variable inspector, it does not provide a detailed view of the values of variables or fields. Instead it allows to browse between all created COGs and their (active or waiting) threads. In other words, the ABS Debug view provides a high level view on the running system.

Some details about the visual representation: The visualization of the active COGs and threads uses a hierarchical tree list where the root node represents the ABS model under simulation. The direct children of the root node stand for the created and active COGs of the ABS model at the current moment in time. Each of these COG nodes can be expanded further to a list of their respective active and waiting threads. Clicking on one of the COGs or threads opens the corresponding source file in the editor highlighting the currently executed statement for the thread. The currently executed statement is highlighted by changing its background color to green and by placing a green arrow marker next to it on the left border of the editor.

The ABS Outline view remains unchanged, but is placed to the left of the ABS source code editor also below the variable inspector.

Another example for a perspective which is developed as part of this task and Task 1.5 (Tool Integration) is the ABS Verification Perspective for the KeY ABS DL prover which integrates the GUI into Eclipse and
provides a proof tree browser, Goal outline as well as a view on the current proof obligation.

In summary, perspectives are a basic and yet powerful tool to tailor the appearance of the IDE for the current workflow. Views on the ABS model that are relevant for the task at hand are emphasized why unnecessary details are hidden. Perspectives can also introduce additional toolbars (or toolbar elements) with shortcuts and controls useful in the specific context.

2.4 Providing User-Feedback

This section gives some advice about providing non-intrusive feedback to the user in a timely manner and close to the point of origin (if applicable). Here, we are mainly concerned with feedback stemming from analyses that are run online while editing or very briefly upon saving of the edited file. More elaborate analyses that need to run for a longer time should usually be triggered by a user action and provide their results in specialized views. However, analyses run in the background might in addition use some of the presented mechanisms to report warnings or errors to the user. A strict side condition is that the reporting is done non-intrusively and does not hinder the workflow of the user. According to [20] the following holds:

- a delay below 100 msec remains unnoticed by the user and provides the experience of immediate response;

- a delay between 200 msec up to one second causes the impression on the user that the system is working on the demand, but gives her/him still the impression to be able to navigate fluidly.

These rules entail that analyses run online or in the background taken together might not delay the response to a key stroke by more than 100 msec. As otherwise the user experiences a delay between her typing and the characters displayed on-screen. Analyses that are triggered by certain events like saving may take up to one second. Analyses that require more time should not be triggered during the normal coding workflow,
but either explicitly by the user or implicitly as part of the response to a certain (longer taking) action which might depend on the analyses outcome (e.g., compilation). In these cases the user should be notified by a progress bar and a waiting cursor that the system is currently working and not able to respond immediately.

2.4.1 Reporting Errors and Warnings

The ABS IDE provides the following standard means to report errors and warnings about the opened ABS models:

Problems View The Problems view is a hierarchical list view that displays warnings and errors. The display is organized in different sections named Errors, Warnings, etc., which contain a list of error/warning reports. These reports consist of a single line structured as follows:

Description: A short textual message explaining the encountered problem
Resource: Usually the source code filename where the encountered problem is located
Path: Usually the directory where to find the resource
Location: Location of the problem as precise as possibly, e.g., line number
Type: The kind of problem, e.g., the name/kind of the analysis which encountered it

Basically all analyses whose results can be interpreted as errors and warnings should report them also in the problem view. This provides the developer with a central list of problematic or erroneous parts of the model and allows to work through them systematically.

Problem Markers Problem markers are part of the editor pane and shown at its left and right borders (see Figure 2.7). They highlight the position where a certain problem is located. The bar on the right side displays all problem markers present in the whole source code file. The developer can directly jump to a the actual line in the source file by clicking on one of these markers. While the bar on the righthand side presents basically a thumbnail view, the actual markers are attached to the bar at the lefthand side. Hovering over them displays a tooltip with further information about the kind of problem. If several problems are detected at the same position, the tooltip contains one explaining line for each of them. Problem markers should be attached closest to the element causing the error. If a problem can be traced back to a single statement, the marker should be at the line where the statement begins, otherwise at the beginning of the smallest enclosing unit, e.g., block, method or class.

Editor Markers The editor itself provides means to visually highlight warnings or problems by underlining the corresponding source code with zigzag lines or highlight an area in a different background color.

Figure 2.7 demonstrates the different means of problem reporting as used by the ABS modeling perspective. Here, the actual error is caused by a misspelled function name and reported in the

1. ABS Problems view, where a short description of the encountered error together with its location is shown and that it has been detected by the ABS type checker.

2. ABS source code editor which has the erroneous ABS file opened. The ABS source code editor uses all three possibilities to highlight the erroneous code:

- in the opened ABS file, one can quickly spot and jump to the location of the error by looking and clicking on the filled red rectangle placed within the gray bar at the right border of the editor. Clicking on the red rectangle causes the editor to jump to the corresponding line of code; leading to the shown situation.
- at the actual error location (line of code) a red circle with a white cross is shown, hovering over this marker displays a tooltip with further information about the error.
• the erroneous statement itself is underlined with a zigzag line.

The color red is used to highlight errors while yellow is used for warnings. These colors should not be used for other purposes like displaying inferred type information or similar.

### 2.4.2 Reporting Additional Information

Analyses performed on the ABS model need not result in a warning or an error. For instance, resource analyses compute the asymptotic runtime for a certain method (and maybe except for non-termination) should not be judged along the categories warning or error. Typically these analyses provide their own view allowing the user to browse the results in a convenient manner.

In cases where the reporting is attached to a certain source code location, and is of general interest for the developer, the analysis results may be reported via the Problem view in the section Others which co-exists next to Errors and Warnings, and as information markers in the editor or as information displayed as tooltip.
when hovering over the corresponding source element (e.g., hovering over a method or loop could show its asymptotic runtime behavior). An additional means to display derived information is to use overlays as introduced and done by the location type checker for inferred types.\footnote{The location type checker and inference engine was developed as part of this task and is reported in detail in Chapter 3 of this deliverable.} The inferred location type is displayed as a superscript (either “N”, “F” or “S”) attached to the type reference of a field or variable declaration (see Figure 2.8)

### 2.5 Conclusions

In this chapter we described and consolidated the design choices for the ABS IDE, which is realized on top of the Eclipse rich client platform. It summarizes the work of several partners on the user interface of the ABS IDE and is not meant to explain the technical principles on which these tools are based. For information about the technical background see the according deliverables available at the HATS project website (http://www.hats-project.eu).

This chapter is also separated out as a living document, which is updated regularly to address new design challenges and to describe the recommended solutions as they arise. It should primarily be consulted when integrating an ABS tool into the ABS IDE, so that existing elements can be reused as far as possible and to achieve a smooth integration.
Chapter 3

Location Types

3.1 Introduction

In the puristic view of object-oriented programming, objects live in an unstructured space (or heap) and communicate via messages. This view is elegant and simple, but fails to address important software engineering principles like decomposition and encapsulation. That is why many researchers work on structuring techniques for object-oriented programming. Structuring the object space provides clear boundaries between different parts of the system. The resulting partitioning can be used to formulate and check important properties, e.g., that an object in one part does not reference an object in another part. Many ownership techniques [6, 7, 24] realize a hierarchical structuring of the objects into so-called ownership contexts and control access to a context from the surrounding context. The goal of our work is to support a partitioning of the object space as in distributed object-oriented programming where each object belongs to exactly one location. Thus, we can analyze whether two objects are at the same location or at different locations.

A location can take many different forms; it may refer to a physical computation node, some process, or it can be a concept supported by a programming language. This versatile notion of locality is not only useful for distributed programming, but also for programs running on a single computer. For example, in object-oriented languages with concurrency models based on communicating groups of objects such as E [19], AmbientTalk/2 [27], JCoBox [25], or ABS [8], the location of an object can be considered as the group it belongs to. In these scenarios it often makes a difference whether a reference points to an object at the current location, i.e., the location of the current executing object (in the following called a near reference), or to an object at a different location (a far reference). For example, in the E programming language [19], a far reference can only be used for asynchronous method calls (named eventual sends in E), but not for synchronous method calls. In Java Remote Method Invocation (RMI) [23] accessing a remote reference may throw a RemoteException, where accessing a normal reference cannot throw such an exception. It is thus desirable to be able to statically distinguish these two kinds of references. In particular, this distinction is useful for documentation purposes, to reason about the code, and to statically prevent runtime errors.

We present location types which statically distinguish far from near references. Location types can be considered as a lightweight form of ownership types [7, 24] with the following characteristics. The first is that location types only describe a flat set of locations instead of a hierarchy of ownership contexts. The second is that ownership types typically support different roles of objects. Location types only classify objects as belonging to the current location or some other location. Furthermore, location types are not used to enforce encapsulation, which is the main goal of many ownership type systems.

As with any type system extension, writing down the extended types can become tiresome for programmers. Furthermore, type annotations may clutter the code and reduce its readability, especially when several of such pluggable type systems [4, 11] are used together. This reduces the acceptance of pluggable type systems in practice. The first issue can be solved by automatically inferring the type annotations and inserting them into the code. But this results again in cluttered code with potentially many annotations. Our solution is to leverage the power of an integrated development environment (IDE) and present the
inferred types to the programmer by using visual overlays. The overlays give the programmer full static type information without cluttering the code with annotations nor reducing readability. Furthermore, the overlays can be turned on and off according to the programmer’s need. Type annotations can still be used to make the type checking and inference modular, where the degree of modularity just depends on the interfaces where type annotations appear. This way of integrating type inference into the IDE simplifies the usage of the proposed type system and is applicable to similar type system extensions.

3.2 Example

In this section, we illustrate the use of location types. After a short introduction to the type system, we explain the language setting based on concurrent object groups for which we developed our implementation and show the benefits of location types using an example.

3.2.1 Location types.

Location types statically distinguish far from near references. To do so, standard types are extended with additional type annotations, namely location types. There are three different location types: Near, Far, and Somewhere. Location types are always interpreted relatively to the current object. A variable typed as Near means that it may only refer to objects that belong to the same location as the current object. Accordingly, a Far typed variable may only refer to objects that belong to a different location than the current object. Somewhere is the super-type of Far and Near and it means that the referred object may either be Near or Far. Note that only Near precisely describes a certain location. A Far annotation only states that the location of the referred object is not Near. This means that a Far typed variable may over time refer to different locations which are not further defined, except that they are not the location of the current object. What a location actually means is irrelevant to the type system. So whether the location of an object represents a specific Java Virtual Machine (JVM) on which the object is running or some other form of object grouping does not matter. Note, however, that the type system relies on the assumption that the location of an object does not change over time.

3.2.2 Concurrent object groups.

We use location types to distinguish near and far references in languages with a concurrency model based on groups of objects. Concurrent object groups (COGs) follow the actor paradigm [13] and were developed to avoid data races and the complexity of multithreading and to simplify reasoning about concurrent programs.

The concurrency model of COGs is used in the abstract behavioral specification language ABS [8] and in JCoBox [25], a Java-based realization of COGs. Groups are created dynamically (cf. [5]) and form the units of concurrency and distribution. Execution within a single group is sequential but groups are running concurrently with other groups. Communication between groups is asynchronous. In a program using COGs, each object belongs to a group for its entire lifetime. This is similar to the Java RMI setting where objects belong to certain JVMs, which may run distributed on different machines.

We recapture here COGs briefly. Each ABS objects lives in a COG where a COG may contain several objects. A COG is created when its first object is created. The creation expression specifies whether the object is created in the current COG (using the standard new expression) or is created in a fresh COG (using the new cog expression). Communication in ABS between different COGs happen via asynchronous method calls which are indicated by an exclamation mark (!). A reference in ABS is far when it targets an object of a different COG, otherwise it is a near reference. Similar to the E programming language [19], ABS has the restriction that synchronous method calls (indicated by the standard dot notation) are only allowed on near references. Using a synchronous method call on a far reference results in a runtime exception. Our location type system can be used to statically guarantee the absence of these runtime exceptions.
interface Server {
    [Near] Session connect([Far] Client c, String name);
}

interface Session {
    Unit receive(ClientMsg m);
    Unit send(ServerMsg m);
}

interface Client {
    Unit connectTo([Far] Server s);
    Unit receive(ServerMsg m);
}

Figure 3.1: The annotated interfaces of the chat application.

Figure 3.2: Runtime structure of the chat application.

3.2.3 Using location types.

As an example, we model an IRC-like chat application, which consists of a single server and multiple clients. For simplicity, there is only a single chat room, so all clients actually broadcast their messages to all other clients. The basic interfaces of the chat application in the ABS language are given in Figure 3.1. Note that only Server, Client, and Session are reference types, the types Unit, ClientMsg, and ServerMsg are data types and represent immutable data.

Figure 3.2 shows a possible runtime structure of the chat application. As the clients and the server run independently of each other, they live in their own COGs. This means that all references between clients and the server are far references. The Session objects that handle the different connections with the clients live in the same COG as the Server object. This means that references between Session and Server are near references. In a typical scenario, the client calls the connect method of the server and passes a reference to itself and a user name as arguments. The server then returns a reference to a Session object, which is used by the client to send messages to the server. The interfaces of Figure 3.1 are annotated accordingly, e.g., the connect method of the server returns a reference to a Session object that is Near to the server.

Figure 3.3 shows the ClientImpl class, an implementation of the Client interface. It has a field session which stores a reference to the Session object which is obtained by the client when it connects to the server. Lines 3-5 show the connectTo method. As specified in the interface, the Server parameter has type Far. On line 4, the client asynchronously (using the ! operator) calls the connect method of the server. The declared result type of the connect method is [Near] Session (see Figure 3.1). The crucial fact is that the type system now has to apply a viewpoint adaptation [10]: As the target of the call (server) has location type Far from the viewpoint of the caller ClientImpl, the return type of connect (which is Near from the viewpoint of Server) is adapted to the viewpoint of ClientImpl, namely to Far. Furthermore, as the call is an asynchronous one, a future is returned, i.e., a placeholder for the value to be computed. The ABS type system uses the built-in
class ClientImpl(String name) implements Client {
    [Far] Session session;
    Unit connectTo([Far] Server server) {
        Fut<[Far] Session> f = server!connect(this, name);
        session = f.get;
    }
}

Figure 3.3: Fully annotated implementation of the ClientImpl class.

class ServerImpl implements Server {
    List<[Near] Session> sessions = Nil;
    [Near] Session connect([Far] Client c, String name) {
        [Near] Session s = new SessionImpl(this, c, name);
        sessions = Cons(s,sessions);
        this.publish(Connected(name));
        return s;
    }
    Unit publish(ServerMsg m) {
        List<[Near] Session> sess = sessions;
        while (~isEmpty(sess)) {
            [Near] Session s = head(sess);
            sess = tail(sess);
            s.send(m);
        }
    }
}

Figure 3.4: Fully annotated implementation of the ServerImpl class.

polymorphic data type Fut to type futures. The type parameter of Fut is instantiated with the type of the value that it is a placeholder for. The variable f on line 4 is thus of type Fut<[Far] Session>. On line 5, the client waits for the future to be resolved and stores the value in the session field. The built-in get operator is used to retrieve the value of the future, blocking if necessary until the value is ready.

Figure 3.4 shows the ServerImpl class, an implementation of the Server interface. It has an internal field sessions to hold the sessions of the connected clients. List is a polymorphic data type in ABS whose type parameter is instantiated with [Near] Session, which means that it holds a list of near references to Session objects. When a client connects to the server using the connect method, the server creates a new SessionImpl object in its current COG (using the standard new expression), which means that it is statically clear that this object is Near. It then stores the reference in its internal list, publishes that a new client has connected (Connected(name) yields the corresponding message), and returns a reference to the session object. In the publish method at line 16, the send method is synchronously called. Here, the location type system guarantees that s always refers to a near object so that the synchronous call does not cause a runtime exception.
3.3 Results

We presented a type system for distributed object-oriented programming languages to distinguish near from far references. We applied the type system to the context of the ABS language to guarantee that far references are not used as targets for synchronous method calls. A complete type inference implementation allows the programmer to make use of the type system without making any annotations. The type inference results are visualized as overlay annotations directly in the ABS Eclipse-based development environment. Our evaluation of the type system to several HATS case studies shows that the type system is expressive enough to type realistic code. The type inference implementation is fast enough to provide inference results within fractions of a second, so that interactive use of the system is possible. A more detailed description of our work can be found in our paper that appeared at TOOLS 2011 [28] and the Appendix of this document (see Section 1.2).

We see three directions for future work. First, the type system could be applied to other settings where the location of an object is important, e.g., Java RMI [24]. Second, it would be interesting to investigate the visual overlay technique for other (pluggable) type systems, e.g., the nullness type system [15]. Third, it seems worthwhile to weaken the premise that objects stay at a location for their entire lifetime (for a motivation of object migration see Mycroft [22] and for the treatment of object transfer in relation to ownership see Müller and Rudich [21]).
Chapter 4

Meta-Programming

In this chapter we present a framework that allows rapid development of ABS parser extensions, model analyses and program transformations as well as ABS compiler backends. This has been achieved by adoption of a meta-programming language.

Ordinary programs work on data, while for meta-programs the role of data is played by programs themselves. In other words, meta-programs are programs that analyze, transform or generate other programs. One prominent representative of a state of the art meta-programming language is Rascal (http://www.rascal-mpl.org). Rascal has been used successfully in various application domains like

- creation of domain-specific languages which have been used for forensic investigation of data, gaming and modeling;
- analysis of Java systems (computing metrics, usage of Hibernate);
- refactoring of source code.

Rascal provides powerful parsing and pattern matching techniques and is tailored towards code analysis, code transformation and code generation.

For this task we added support for ABS to the Rascal meta-programming language. We want to emphasize that the integration has been done thoroughly and all analysis, transformation and code generation is performed directly on the ABS model instead of, for instance, on the code generated by the Maude or Java ABS backends. In the following sections we describe our results of adding support for the ABS language to Rascal and how to use the resulting framework to implement domain-specific language extensions, perform code analysis and more.

4.1 Code Analysis

Code analysis forms the basis for all other tasks (Figure 4.1) that are implemented using Rascal. In the analysis phase, relevant information is extracted directly from the source code of an ABS model. The extracted information is stored in an internal representation which can be used as input for code transformation or code generation.

Examples of typical code analyses are the calculation of code metrics, generation of documentation, the construction of an abstract or concrete syntax tree of an ABS program, syntax-highlighting and call graph extraction from ABS programs. In this section we discuss how the latter three analyses have been achieved.

In order to capture the grammatical structure of ABS programs we translated the existing ABS Beaver grammar (which is used in the ABS compiler front-end) to Rascal.

Due to Rascal’s support for general context-free grammars the translation did not cause major problems and has been achieved fairly easy: a large part of the translation was automated, since all Beaver features are also supported by Rascal. The resulting grammar has about 260 non-terminal elements. For comparison:
syntax FunctionDecl
    = Annotation* al DEF Datatypeuse t IDENTIFIER fn 
      LPAREN {ParamDecl ","}∗ params RPAREN ASSIGN BUILTIN SEMICOLON 
    | Annotation* al DEF Datatypeuse t IDENTIFIER fn 
      LPAREN {ParamDecl ","}∗ params RPAREN ASSIGN ExpFunctionDef ef SEMICOLON 
    | ... 

syntax Methodsig 
    = Annotation* l TypeExp returntype IDENTIFIER id LPAREN {ParamDecl ","}∗ params RPAREN;

Figure 4.1: Meta-programming workflow. Source: the Rascal website.

Figure 4.2: Selected non-terminal declarations in the Rascal ABS grammar

A full Rascal 1.4 Java grammar contains around 125 non-terminals. The reason for the relatively huge amount of non-terminal symbols is owed to a large account to the support for variability like delta-oriented programming and product configurations in ABS which are not present in a language like Java.

Another reason is that the ABS Beaver grammar introduces new non-terminal symbols for lists of grammar symbols. For instance, the non-terminal symbol ParamDeclList for a possibly empty comma-separated list of formal parameters is introduced. In Rascal, such lists can be defined directly using the grammar expression {ParamDecl ","}∗ without the need to introduce new non-terminal symbols. Polishing the grammar and rewriting those rules that need to parse lists of elements to use the more elegant direct encoding resulted finally in a grammar with only 234 non-terminal symbols (see Figure 4.2).

In the current ABS Beaver grammar, the grammar is extended with actions to build up the abstract syntax tree of the parsed ABS program. Each internal node in the resulting abstract syntax tree corresponds to a non-terminal symbol while each leaf corresponds to a terminal symbol. Hence, the abstract syntax tree is completely determined by the grammar. The usage of Rascal simplifies the existing grammar further as there is no need to manually define actions for abstract syntax tree construction. Instead the construction is performed automatically via a library function parse, which takes a non-terminal (in our case, the start symbol of the Rascal ABS grammar) and a string (in our case, an ABS program) as input and returns a parse tree. Syntax highlighting is also supported out-of-the-box by another library function which produces a highlighted ABS program (optionally with user-defined color schemes) based on its parse tree (Figure 4.4). In case that an ABS program contains syntax errors, no parse tree is produced, for such programs the
Figure 4.3: Parse tree of the ABS statement `await l2?` rendered by Rascal

location of the parse error is immediately isolated and highlighted.

Figure 4.4: ABS Syntax-Highlighting generated by Rascal

The produced parse tree can be used directly to extract a call graph of the ABS program. The Rascal snippet in Figure 4.5 contains the full code needed to extract a call graph from the ABS programs parse tree. A call graph is a set of tuples representing all asynchronous calls present in a given ABS model. A single asynchronous call is represented as a tuple `<m, n>` expressing that method `m` asynchronously calls method `n`.

To extract the call graph the Rascal program of Figure 4.5 uses an array of techniques, namely, parsing, pattern matching and set comprehension. In more detail, `Method` and `AsyncCall` are non-terminal symbols declared in the ABS grammar and used as abstract syntax tree nodes of the ABS model. The second line in the snippet traverses the syntax tree of the parsed `ABSFile` and pattern-matches any `Method` node (i.e., any node representing an ABS method declaration) and assigns it to variable `m`.

The third line subsequently pattern-matches all asynchronous call-statements in the subtree with `m` as root (i.e., all asynchronous calls that occur in the body of method `m`).

Finally the matches are accumulated in a set comprehension of pairs `<m.ms.id, c.method>`. The first coordinate `m.ms.id` is the name of the declared ‘Method’, the second coordinate `c.method` is the name of the method which was asynchronously called by `m`.

This concludes our section on code analysis using Rascal. The described examples demonstrate the expressiveness of Rascal and the rich support to support writing of complex analyses for ABS programs.
In this section we explain the code transformation support provided by Rascal and illustrate its usage on a few examples.

In general, a code transformation takes an ABS model as input and outputs a different (transformed) ABS model. Code transformations are used by several ABS tools for a number of purposes. One concrete area where code transformation is used is the elimination of syntactic sugaring (and more generally, preprocessors) and to produce a pure Core ABS abstract syntax tree. Refactorings can also be implemented as code transformations. Yet another example for the usage of code transformation is aspect-oriented programming, which can be implemented by instrumenting (also known as weaving) source code at appropriate user-defined places (pointcuts).

In Rascal, code transformations are performed on the parse tree representation of the ABS program. The code transformations are basically realized by only using the following three concepts:

- a high-level visit-statement to walk through the tree,
- pattern matching to identify the relevant nodes to transform in the tree and
- rewrite rules to transform these nodes.

Figure 4.6 shows a complete Rascal program that weaves a skip statement into the beginning of each method body in the parse tree of the given ABSFile. The actual work is done by method insertStmt.

Although this transformation itself is rather useless, one could have instead added logging or tracing statements. The last line in method insertStmt writes the transformed abstract syntax tree newTree as a pretty printed ABS program into an ABS file which can then be compiled by the ABS compiler.

A very useful feature of Rascal is that transformations are ensured to be valid with respect to the ABS grammar. This means it is not possible to produce a syntactically malformed ABS program.

Using the Rascal meta programming language allows to easily describe transformation of ABS models and enable for instance to easily add support for domain-specific language to ABS (e.g., embedding of SQL queries for database intensive models) which are then transformed away into CoreABS (e.g., by mapping the queries to corresponding library calls).

### 4.3 Code Generation

Code generation converts an intermediate representation of source code written in a programming language $\mathcal{L}_{\text{in}}$ (the input language) into code of a programming language $\mathcal{L}_{\text{out}}$ (the output language). In our case, the internal representation consists of a concrete syntax tree. Any language can be used as output, and any language for which a Rascal grammar is available can be used as input. It is not necessary that a grammar is available for the output language, the resulting code can be outputted as a normal string generated using powerful string interpolation techniques. However if a grammar is available, the generated source code is guaranteed to be syntactically correct with respect to that grammar.

We describe now the relevant cases where ABS is the input or output language (or both). In the previously discussed code transformations, both ABS played the role of the input and the output language

---

1 Examples include Java, C, numerous domain-specific languages and now ABS.
import ParseTree;
import ABS;
import IO;
import ToString;
import String;
import util :: IDE;

// Aspect which weaves a skip statement
// at the beginning of each method body
public void insertStmt(loc f) {
    ptree = parse(#start[Goal], f);
    newTree = visit(ptree) {
        case (Method) '<Method m>' =>
            (Method) '<[Methodsig] "<m.ms>" >
            <[Block] "<m.b.al> { skip; <m.b.b.s> }">'
    }
    writeFile(f, newTree);
}

Figure 4.6: An aspect that weaves a skip-statement into each method body

(ABS to ABS). Adding support of domain-specific languages (DSL) to ABS is an example where a different language (namely, the given DSL) is used as input language for which ABS code is actually generated.

Finally, as an example where ABS is translated into another language, we highlight how Rascal is used to implement the code generation part for the various ABS backends in a systematic manner. In other words, we show how to generate a compiler for the ABS language.

In general, code generation requires to first create a Rascal grammar for the source language. In our case, we can use the already existing and previously explained ABS grammar. In a second step, one has to provide for each non-terminal element of the ABS grammar (#ABSNonTerminal) and (overloaded) function strGenerate(#ABSNonTerminal t). The function may have additional parameters to pass extra information from the ABS grammar to the code generator.

The actual parameter of Generate is a parsed fragment of ABS source code of type #ABSNonTerminal. The return value is a source code fragment in the target language, computed by appropriately combining generated source code of child nodes. If a grammar for the target language is available, the return type of Generate is a parse tree in the target language instead of a string. In this case, the resulting target source fragment is guaranteed to be syntactically correct. Any potential syntax errors in the generated source code are caught as early as possible, i.e. at the level of the smallest fragment containing the error. This is desirable for debugging the compiler being constructed. The Generate functions and the Rascal ABS grammar together can be considered to form an attribute grammar. The (optional) additional parameters of Generate are the inherited attributes and the return value is a synthesized attribute.
Chapter 5

Sound Control-Flow Graph Extraction

In program analysis, control-flow graphs (CFGs) are a widely used program model, where only the control-flow information is kept, and all program data is abstracted away. In a CFG, nodes represent the control points of the program, while edges represent the instructions that move control between control points. Figure 5.1 illustrates the concept by showing a simple Java program and the control-flow graph extracted from it.

Numerous techniques have been proposed to extract automatically control-flow graphs from program code, see e.g. [17, 18]. Typically, however, these are not accompanied by a formal correctness argument. In a recent paper [3] we attempt to fill this gap: we define a control-flow graph extraction algorithm for sequential Java bytecode (JBC), and show that the extraction algorithm is sound w.r.t. the behavior (i.e., executions) of the program, in the sense that all behaviors of the extracted CFG are a sound over-approximation of the original program behavior. In particular, the extraction algorithm produces control-flow graphs that are sound for the verification of temporal safety properties.

The extraction algorithm considers all the typical intricacies of Java, such as virtual method call resolution, the differences between dynamic and static object types, and exception handling. The sound analysis of exceptional flows is particularly challenging for two reasons. First, the stack-based nature of the Java Virtual Machine (JVM) makes it hard to statically determine the type of explicitly thrown exceptions, thus making it difficult to decide to which handler (if any) control will be transferred. Second, the JVM can raise (implicit) run-time exceptions, such as NullPointerException and IndexOutOfBoundsException, and to keep track of where such exceptions can be raised requires much care.

**Connection with ABS** The work discussed here targets the verification of temporal safety properties of implementations of ABS models as sequential Java bytecode. Our soundness argument does not rely on any assumptions on the behaviour of the model or the correctness of code generators.

Chapter 4 of this deliverable shows how to implement relatively easily a control-flow graph extractor from ABS models. Proving soundness in this setting will arguably be easier than for Java bytecode. However, to
guarantee that temporal properties verified for an ABS model also hold for an implementation of the model obtained via code generation, one would need additional soundness proofs for the used code generator.

**Extracting Control-Flow Graphs from BIR** We present a two-phase extraction algorithm using the Bytecode Intermediate Representation (BIR) language [9], developed by Demange et al. The use of BIR has several advantages. First of all, BIR provides a stack-less representation of JBC. Thus, all instructions (including the explicit `athrow`) are directly connected with their operands. This allows to determine the static type of explicitly thrown exceptions. In addition, the representation of a program in BIR is smaller, since operations are not stack-based, but represented as expression trees. Second, BIR supports the analysis of implicitly thrown exceptions by generating assertions that indicate when the next instruction might raise a run-time exception. Figure 5.2 presents a summary of the transformation function from JBC into BIR.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pop</code></td>
<td><code>0</code></td>
<td><code>nop</code></td>
<td><code>[nop]</code></td>
</tr>
<tr>
<td><code>push c</code></td>
<td><code>0</code></td>
<td><code>if p</code></td>
<td><code>[if e pc']</code></td>
</tr>
<tr>
<td><code>dup</code></td>
<td><code>0</code></td>
<td><code>goto p</code></td>
<td><code>[goto pc']</code></td>
</tr>
<tr>
<td><code>load x</code></td>
<td><code>0</code></td>
<td><code>return</code></td>
<td><code>[return]</code></td>
</tr>
<tr>
<td><code>add</code></td>
<td><code>0</code></td>
<td><code>vreturn</code></td>
<td><code>[return e]</code></td>
</tr>
</tbody>
</table>

**Figure 5.2:** Transformation function from JBC into BIR

Figure 5.3 presents a summary of the extraction function of CFGs from BIR. The overall CFG extraction algorithm is defined as the functional composition of the two functions.

\[
\mathcal{H}^{pc}_{x} = \begin{cases} 
\{(c_{m}^{pc,x}, handle, c_{m}^{pc'})\} & \text{if } h_{H}(pc, x) = pc' \neq 0 \\
\{(c_{m}^{pc,x}, handle, c_{m}^{pc,r'})\} & \text{if } h_{H}(pc, x) = 0 
\end{cases}
\]

\[
bG(i_{pc}, H) = \begin{cases} 
\{(c_{m}^{pc,x}, c_{m}^{pc+1})\} & \text{if } i \in \text{Assignment} \cup \{[\text{nop}], [\text{mayinit}]\} \\
\{(c_{m}^{pc,x}, c_{m}^{pc+1}), (c_{m}^{pc}, c_{m}^{pc'}, c_{m}^{pc+1})\} & \text{if } i = [\text{if expr pc'}] \\
\{(c_{m}^{pc,x}, c_{m}^{pc'}, c_{m}^{pc+1})\} & \text{if } i = [\text{goto pc'}] \\
\{(c_{m}^{pc}, c_{m}^{pc'+1}), (c_{m}^{pc', c_{m}^{pc'}, c_{m}^{pc'+1}})\} & \text{if } i \text{ is a BIR assertion} \\
\bigcup_{x \in X} \{(c_{m}^{pc,x}, c_{m}^{pc+1}, c_{m}^{pc', c_{m}^{pc'}, c_{m}^{pc'+1}})\} & \text{if } i \text{ is a constructor} \\
\bigcup_{h \in res^{\ast}(ns)} \{(c_{m}^{pc,x}, n, c_{m}^{pc+1})\} & \text{if } i \text{ is a method call} 
\end{cases}
\]

\[
N^{pc}_{n} = \bigcup_{n', x'} \in bG(n) \{(c_{m}^{pc}, handle, c_{m}^{pc,x'})\} \cup \mathcal{H}^{pc}_{x}
\]

**Figure 5.3:** Extraction rules for control-flow graphs from BIR

**Implementation** The two-phase extraction algorithm uses the transformation from JBC to BIR from Sawja [14], a library for static analysis of Java bytecode. It then extracts CFGs from BIR. The algorithm is
implemented as the tool ConFLEx. Since Sawja only provides intra-procedural support for exceptions, to obtain a sound extraction tool we implemented on top of it a fixed-point computation of the inter-procedural exceptional flow caused by uncaught exceptions.

Table 5.1 provides statistics for the CFG extraction of several examples with varying sizes. Methods from the API are not extracted; only classes that are part of the client program are considered. BIR Time is the time spent to transform JBC into BIR. For the transformation from BIR to CFG, we provide statistics for the intra- and the inter-procedural analysis separately.

<table>
<thead>
<tr>
<th>Software</th>
<th># of JBC instr.</th>
<th># of BIR instr.</th>
<th>BIR time (ms)</th>
<th># of nodes</th>
<th># of edges</th>
<th>time (ms)</th>
<th># of nodes</th>
<th># of edges</th>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jasmin</td>
<td>30930</td>
<td>10850</td>
<td>267</td>
<td>19152</td>
<td>19460</td>
<td>320</td>
<td>21651</td>
<td>21966</td>
<td>25</td>
</tr>
<tr>
<td>JFlex</td>
<td>53426</td>
<td>20414</td>
<td>706</td>
<td>38240</td>
<td>38826</td>
<td>859</td>
<td>42442</td>
<td>43072</td>
<td>23</td>
</tr>
<tr>
<td>Groove Ima.</td>
<td>193937</td>
<td>77620</td>
<td>587</td>
<td>159046</td>
<td>158593</td>
<td>4817</td>
<td>193268</td>
<td>192905</td>
<td>1849</td>
</tr>
<tr>
<td>Groove Gen.</td>
<td>328001</td>
<td>128730</td>
<td>926</td>
<td>251762</td>
<td>252102</td>
<td>13609</td>
<td>308164</td>
<td>308638</td>
<td>5541</td>
</tr>
<tr>
<td>Groove Sim.</td>
<td>427845</td>
<td>167882</td>
<td>1072</td>
<td>311008</td>
<td>311836</td>
<td>16067</td>
<td>386553</td>
<td>387556</td>
<td>6886</td>
</tr>
<tr>
<td>Soot</td>
<td>1345574</td>
<td>516404</td>
<td>98692</td>
<td>977946</td>
<td>976212</td>
<td>26490</td>
<td>1209823</td>
<td>1208358</td>
<td>57621</td>
</tr>
</tbody>
</table>

The practical results show that in all cases the number of BIR instructions is less than 40% of the JBC instructions. This indicates that the use of BIR mitigates the blow-up of control-flow graphs, and clearly program analysis benefits from this. The computation time for intra- and inter-procedural analysis grows proportionally with the number of BIR instructions. The intra-procedural analysis is linear w.r.t. to the number of instructions, and the experimental results of the inter-procedural analysis show that it only contributes to a small part of the total extraction time.

**Soundness of CFG Extraction** Proving soundness of the two-phase extraction algorithm directly (e.g., by means of behavioral simulation) is cumbersome. Instead, to simplify the overall correctness argument we use the correctness of an idealized direct extraction algorithm developed by Amighi [1]. We connect the CFGs that are extracted by the idealized algorithm and by the two-phase algorithm via a (structural) simulation relation, and use a previous monotonicity result [12] to infer behavioral simulation. From this, one can conclude that all behaviors of the CFG generated by the indirect algorithm (BIR) are a sound over-approximation of the original program behavior. Thus, the extraction algorithm produces control-flow graphs that are sound for the verification of temporal safety properties. Figure 5.4 summarizes the proof strategy.

![Figure 5.4: Schema for CFG extraction and correctness proof](image-url)
The proof that CFGs extracted with the indirect algorithm simulate the CFGs extracted by the direct algorithm proceeds follows by case analysis on the Java Bytecode instructions set, and can be found in the accompanying technical report [2].

Conclusion  We present an efficient and sound control-flow graph extraction algorithm that takes into account exceptional control flow. The extracted CFGs can be used for various control-flow analyses, in particular model checking. The algorithm is precise because it is based on BIR, an intermediate stackless bytecode representation, which provides precise information about exceptional control-flow and is more compact than the original bytecode.

The algorithm is presented formally as an extraction function. We state and prove its soundness: the behavior of the extracted graphs is shown to over-approximate the behavior of the original programs. To the best of our knowledge, this is the first CFG extraction algorithm that has been proved correct. The proof is non-trivial, relying on several results to obtain a relatively economic correctness argument phrased in terms of structural simulation. We believe that the proposed proof strategy, with the level of detail we provide, paves the ground for a mechanized proof using a standard theorem prover.

The extraction algorithm is implemented as the ConFlEx tool. The experimental results confirm that the algorithm is efficient, and that it produces compact CFGs.

The extraction algorithm has been designed with modularity in mind. Currently, we investigate how to relativize the algorithm on interface specifications of program modules in order to support modular control-flow graph extraction. In particular, we target CVPP (see e.g. [26, 16]), a framework and tool set for compositional verification of control-flow safety properties. In this setting, one typically wishes to produce CFGs from incomplete programs.
Chapter 6

Conclusion

This deliverable reported on the user interface design of the ABS IDE, on a type system extension for location types realized as pluggable type system extension and using a novel approach to present type inference results within the ABS source code editor using overlays. Further, we presented a general framework for the development of ABS language extensions, compilers and analyses.

The results of this deliverable are intended to ensure a viable and coherent platform for ABS beyond the project’s lifetime. It ensures a consistent usable and extensible development environment and provides a well-documented framework for extending the ABS tool suite with additional features. Both aspects contribute to a sustainable technical platform that has the potential to make an impact in research and industrial application long after the end of HATS.
**Bibliography**


**Glossary**

**ABS** Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

**COG** Concurrent Object Group, the unit of parallelism in ABS.

**Core ABS** The behavioral functional and object-oriented core of the ABS modeling language.

**Full ABS** Core ABS plus PCL (Product Configuration Language), PSL (Product Selection Language), DML (Delta Modeling Language) and more.

**Launch Configuration** Configuration of the runtime environment.

**Location Types** Types that restrict the possible objects referenced by a variable not only by interface type but also by the location (concurrent object group) they live in.

**Meta-Programming Language** Programming language specialized to be used for analyzing, transforming and generating programs.

**Perspective** A collection of views, their layout and default IDE configurations tailored towards a specific development task.

**Pluggable Type Checker** A type checker that can be parametrized by (a set of) different independent type systems.

**User Response Time** The time a system needs to respond to a user input.

**Style Element** Graphical element (font, icon or other graphical representation).

**Rascal** A meta programming language tailored towards application development for program analysis, transformation and code generation.

**View** A graphical component that presents a certain aspect of the ABS model to the developer in a concise and optimized manner.